Sanity: The Less Server Architecture for Cloud functions

Shripad Nadgowda, Nilton Bila, Canturk Isci, Ricardo Koller
IBM TJ Watson Research Center, NY USA

Abstract

Serverless execution model is becoming an increasingly prominent choice to host data processing applications. In this event-based model, predefined stateless functions are triggered for execution on an event when new data becomes available. We make two critical observations in this framework – *first*, data sources in the serverless framework are largely-consistent, causing lot of **data duplication**, and *second*, stateless functions are **idempotent**, i.e. they compute same result on multiple applications of duplicate data. Leveraging these two insights in this work, we introduce Sanity[1], a novel storage de-duplication framework for serverless platform that not only performs data de-duplication but also de-duplicates events to avoid redundant execution of stateless functions. This helps improve overall throughput of the serverless platform through execution of mostly unique functions. We present the design of Sanity for continuous container vulnerability check use-case and discuss its performance implications.

1 Introduction

Serverless platform has made distributed computing model extremely accessible to users, wherein users can simply submit a stateless function for execution without worrying about complex cluster management or configurations [13]. We like to consider it as a commodity application of disaggregated computing wherein general-purpose cloud storage is connected to shared compute platform through standard network fabric. Such disaggregation has enabled a clear separation of concerns between management of data at storage systems and execution of functions at servers. These two systems are then connected through an explicit relationship of *events*.

Different cloud providers have made such platform commercially available in form of Amazon Lambda [2], IBM Cloud Functions [12], Google Cloud Functions [11], Azure Functions [17]. Fig 1 (A) shows simplistic design for most general use case of serverless platform. Data generated from various external end-points like IoT devices, cloud systems monitors, weather sensors, social media, mobile devices, etc, are stored in a common storage system viz. object store (Amazon S3), key-value store (IBM Cloudant, DynamoDB) or messaging service (kinesis, Kafka). These storage systems generate events on arrival of new data, which then trigger function executions on the servers. The functions read new data from remote storage system, perform their computation on the data and generate result data which is stored back on the remote storage system. Every communication (event trigger, data read/write) between storage systems and servers is performed over a standard commodity network.

From a programmer's point-of-view, serverless systems are essentially a very granular and economical way of performing their computational tasks for event-driven applications. The serverless model is becoming a prominent choice for hosting data analytics and scientific computing workloads [13] as well as IoT applications. Serverless functions in such applications read data in bulk from remote storage, perform analytic tasks and store back transformed data or output to the storage system. Thus, the rate of execution of these functions is limited by the available bandwidth between the disaggregated storage and the servers.

Let us consider two important properties about data and functions together in the context of serverless applications. First, **data has lot of duplication** and second, **functions are typically stateless or idempotent**. Thus, a given function can be applied multiple times over the same data without changing the result from the initial application. In this paper we introduce Sanity, a system that performs *inline* data de-duplication at the storage system to avoid repeated execution of idempotent functions on the servers, as shown in Fig. 1(B). This primarily helps save computation cycles from redundant function executions, reduces network overhead of transferring duplicate data between servers and storage systems, and optimizes storage space at the storage systems. Also, as most platforms execute these functions inside containers, by eliminating redundant activation of functions, Sanity avoids container startup latency. Sanity can also be applied to sequencing of idempotent functions as shown in Fig. 1(C).

In Sanity systems, we generally argue that for serverless platforms it is performance-efficient and cost-effective to perform data de-duplication close to the event source to avoid the redundant activation of idempotent

[1]Duly addressing and acknowledging Einstein's quote- "Insanity: doing the same thing over and over again and expecting different results."

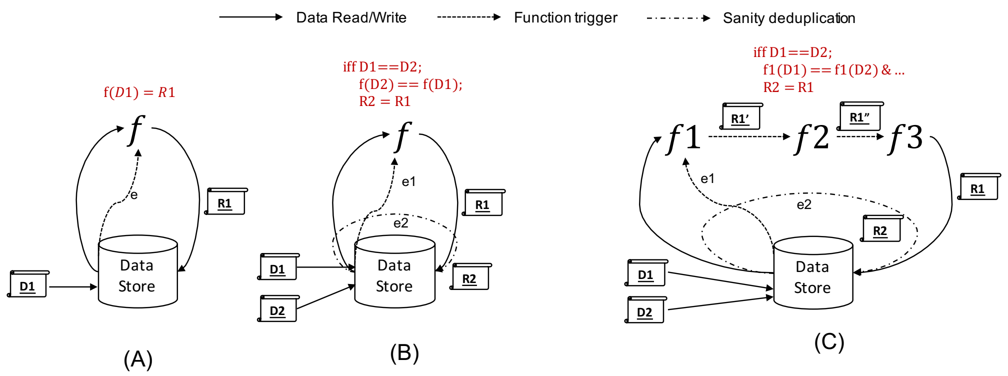

Figure 1: Sanity Serverless Platform

functions. We measured effectiveness of Sanity system for a continuous container vulnerability scanning use-case and report that up to 200x performance benefits can be achieved with modest overhead.

2 Motivation

Sanity is primarily based on the two principles i.e duplicate data and idempotent functions, both being true at the same time in the context of serverless platforms. In this section we will reason about how to validate these principles.

On most serverless platforms function code is dynamically inserted and executed inside a container. And typically these containers are reused across multiple invocations of the same function to mask the container startup latency. But no platform guarantees that and state maintained locally by a function might not be available across invocations. Therefore, by design all serverless platforms implement stateless function semantics that enables them to scale at the line rate of incoming events. This statelessness programming model of functions can also be identified as idempotent.

Common data sources for serverless applications includes IoT/sensor data (e.g. weather), social media data(e.g. tweets), user activity data(e.g. click-stream), system state monitoring data(e.g. agentless-system-crawler [14] for VMs or containers). These data exhibits duplication over both spatial and temporal dimensions. For example, weather data typically has a small range of bounded values (e.g. -20C <temperature <50C). And data collected from a sensors fixed at a particular location might typically be same for most part of the days, months or seasons. Also data collected from multiple geo-distributed sensors might also be same. Such duplicate data characterization is common for many big data analytic applications, one such application is exemplified in our use-case in next section.

Finally, the disaggregation of storage and servers in serverless systems presents an opportunity to build a specialized data de-duplication service on top of general-purpose storage without any disruption in core component of serverless systems, viz. orchestration and management of activation of functions on servers.

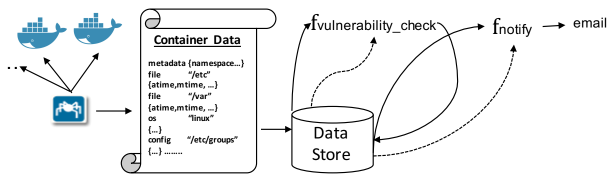

Figure 2: *Vulnerability Analysis on Serverless Platform*

3 System Design

Sanity is essentially a data deduplication system designed not to save the storage space for duplicate data, but to avoid redundant invocation of functions on servers for duplicate input data. In this section we will discuss some salient features of Sanity systems, in particular, specialized data curation techniques and optimized deduplication data structure and indexing. For brevity in the discussion we will consider one specific use-case of serverless application. But, Sanity is generally applicable to most serverless applications.

3.1 Use case

Consider a Vulnerability Analysis use-case for containers in cloud as shown in Fig 2 In this scenario a data collection agent like crawler [14] periodically introspects every live running container to capture their state (metrics, files, packages, configurations etc.) in a blob called *frame*. The crawler generates these frames typically every 1 hour. These frames are then pushed to the elasticsearch and an event is triggered for serverless function $f_{vulnerability-check}$. On invocation this function queries ES to read new frame, performs vulnerability analysis and writes vulnerability report back to elasticsearch. This write then triggers another function f_{notify} to send email alert about new findings.

According to our previous study [23], during 2 weeks, only 3-5% of containers in production cloud platform showed drift in their state since their startup. The drift was measured in terms of change of files/packages/-configurations. Thus, clearly many frames produced for same containers were duplicates during this time. Also, in general in cloud, large number of containers are instantiated from relatively small number of images, thus frames are potentially duplicate across containers as well.

3.2 Classifying idempotent functions

Both functions described in the use case above are idempotent. We categorize all serverless functions into two classes, namely *storage closed-loop* and *external stimuli*. Functions that read input data and write their result back to the data store are termed as *storage closed-loop functions*, e.g. $f_{vulnerability-check}$. And functions that execute to stimulate external events like sending email, slack, SMS etc. are termed as external stimuli functions, e.g. f_{notify}. In Sanity our objective is to de-duplicate execution of functions but provide the same effect if function was executed. And since external stimuli can not easily be duplicated, currently Sanity is primarily applicable for storage closed-loop functions.

In some cases, users tend to embed storage closed-loop function with external stimuli, e.g. consider $f_{vulnerability-check-notify}$ that analyze vulnerabilities, stores report in ES and then notify through email. Sanity enforces *pure* storage closed-loop semantics and such tainted functions are not considered for de-duplication. As discussed earlier, most data processing applications like big data analytics, map-reduce tasks, scientific workloads are pure storage closed-loop applications.

3.3 De-duplicated storage system

In Sanity we design data de-duplication primarily to save the associated function computation on the duplicate data, while we may or may not get the performance and capacity optimization benefits at storage system.

All existing de-duplication techniques uses exact content-matching using MD5 or SHA1 on the whole original data to identify duplicate data. Many existing big data filesystems like HDFS [6], object storage systems [3] [24] already computes and stores checksum along with the data, that are later used for integrity checking, data scrubbing, auditing, bit-error analysis, which can be used in Sanity right out-of the box. Such content-hash indicators are very generic and application-agnostic.

On the other hand, data used for any computation has some metadata to describe its properties (e.g. size, format...) or context(e.g. timestamp, namespace ...). And most common data exchange formats (e.g. json) tends to include them in the data itself, which introduces lot of variablities in data. Therefore, in Sanity we consider a notion of *semantically* equivalent data, wherein instead of performing content-match on the original data, we transform the data aided with knowledge of function's use of that data, which is then used for comparison. Such application-aware data de-duplication technique helps improve accuracy of data matching. Two such methods are discussed below:

PoV based duplication Consider a sample frame from our use-case as shown in Listing 1. In the example, "metadata" field captures the meta information about particular run of a crawler, while remaining fields describes state of the container in terms of available files, packages etc. during that run.

```
{
metadata:{
namespace: "dev/mysql",
crawl-time: "2017-03-11T17:04:42"...
},
file:{
name: "/etc/hosts",
atime: "1459243509",
mtime: "1459243509",...
}...
packages:{
name: "coreutils",
version: "0.5.8-2.1ubuntu2",...
}...
}
```

Listing 1: Sample container frame

Metadata, although important, usually is not used during the computation. In Sanity we allow annotation of such Points of Variabilities or *PoVs* in the data, so that those can be ignored during content-matching. For example, *namespace* describes name of the container, while $crawl-time$ is a timestamp when crawler introspected that container. In our use-case both $crawl-time$, *namespace* are clearly PoVs since they are going to change across multiple frames collected over time and different containers. Further, specific to our $f_{vulnerability-check}$, since only a change in file and/or package can influence a result, we can safely annotate *atime* for files, which is again high susceptible to change across frame. We replace the annotated values for PoVs as shown in Listing 2. Although we could automatically identify few common PoVs [4] like timestamp, IPAddress etc., Currently we allow user to define these PoVs over data schema. The PoV-annotated frame is then

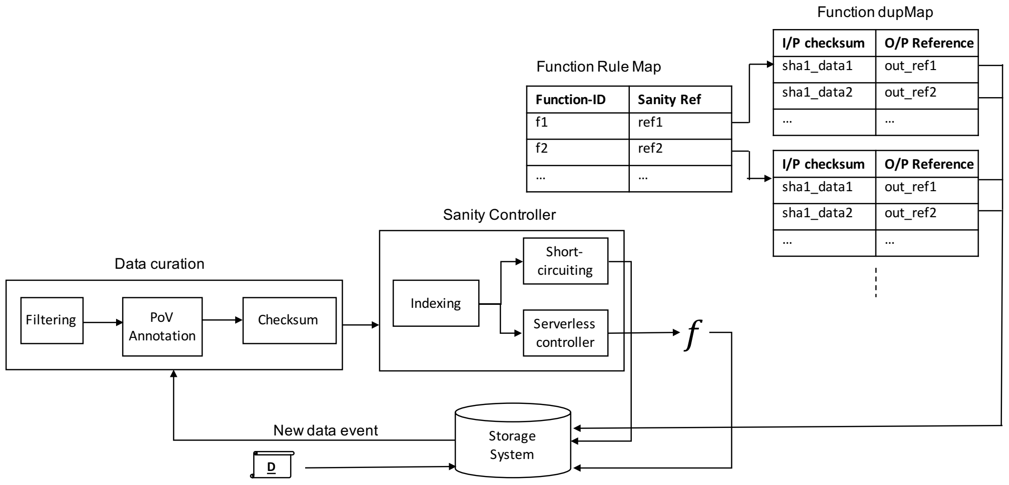

Figure 3: Sanity System Design

chesksum using SHA1 and lookup in the hash-table. This PoV annotations are done equally for input and output data of a function.

```
{
metadata:{
namespace:  "$namespace$",
crawl-time:  "$crawl-time$"...
},
file:{
name:  "/etc/hosts",
atime:  "$atime$",
mtime:  "1459243509",...
}...
packages:{
name:  "coreutils",
version:  "0.5.8-2.1ubuntu2",...
}...
}
```

Listing 2: PoV annotated frame

Filter based duplication This is slightly coarse-grained and specialized case of PoV annotation. In this case, input data is filtered to identify the actual segment of the data being used by the function and the rest of data is ignored during content-match. Following our use-case, container frame contains data about processes, disks , metric data, which is not used by $f_{vulnerability-check}$. So any changes in those data values are not relevant to our function execution, so those can be easily ignored. So checksum can be computed only on the metadata, file and package sections of the frame.

Data Storage It is important to note that, PoV annotations and filtering is done in-memory temporarily for checksum computation only. While storing in the database, the actual data without any annotations or filtering is stored. For PoV annotated data we also maintain the annotated PoVs and their real values only during the lifetime of that request, so that they can be mapped in the result data, as discussed in the next section. Thereofore, it is important to note, Sanity data de-duplication does not help save the storage space, if the data is found duplicate after filtering and annotations, since the original data will get stored multiple times. It primarily helps de-duplicate the execution of associated functions. In other design, data can be stored with annotations and PoVs mapping separately allowing opportunity de-duplicate annotated data to space saving on media. But, the cloud storage used in serverless systems are general purpose cloud storages used for applications beyond serverless. And since such annotations are typically application specific, in general they can not mutate the original data.

3.4 Sanity architecture

Fig. 3 shows the overall architecture for Sanity system. For every new frame, first data is curated and checksumed. The event and chechsum is communicated to the Sanity controller. The controller internally maintains 2-level hashmap for de-duplication management. It periodically queries serverless platform and store rules in *Function rule map* to associate data events with respective function(s). As alluded before, only storage closed loop functions are considered and stored in this map. For each function a separate *Function dedup* map is stored which maintains checksum of all (unique) input data processed by that function and reference to output data in the data store for respective input data. For key-value stores, the reference to output data is *key*, for object stores this reference is $object-path$. The size of function rule map is limited to $1KB$, which can potentially store $2K$ entries. Further, size of each function dedup map is limited to $10MB$ which can host deduplication metadata for $40K$ unique data for each function. Total memory overhead of de-duplication metadata is $2GB$. Beyond this capac-

Computational task	Time
PoV annotation and filtering	4.7 ms
SHA1 computation	0.05 ms
Indexing and output de-duplication	0.02 ms
Total	5 ms

Table 1: Sanity de-duplication overhead

ity entries in the maps are replaced using standard LRU policy.

For every new event, Sanity controller first indexes into the hashmaps to identify if the event is duplicate for a function. If the event is identified as duplicate, then gets the output reference for the result from earlier invocation. Using the reference it queries result data from storage system and PoV annotates them. PoVs then either replaced with their corresponding values from input data (e.g. namespace, atime etc.) or computed dynamically (e.g. current-time). The new result is then stored back in the storage system. Thus, the invocation of a function is avoided still producing the same effect. For example, one sample output for vulnerability report is shown in Listing 3. For duplicate input data *namespace* is populated with PoV value from input data and *timestamp* is evaluated and set accordingly.

```
  {
  check-id:  "vulnerability-1.1.0a",
  namespace:  "$namespace$",
  timestamp:  "$eval(time)$",
  vulnerability-status:  "false",
  reason:  "No known vulnerability
found"
  }
```

Listing 2: Sample PoV Annotated output

If Sanity controller do not find a duplicate match, then the event is identified as unique and it triggers execution of a function on serverless platform. On completion, function stores result data in the data store. The controller asynchronously keeps updating the function dedup hasmap for every unique functions execution.

4 Evaluation

We conducted micro-experiments to evaluate effectiveness of Sanity system for our vulnerability-check usecase. We used IBM Cloud functions for serverless platform and hosted our Sanity (de-duplicated) data service over elasticsearch in the same region. Containers are crawled periodically and *frames* are sent to our data service. Although *size* of the frames are different for every container, average size was noted to be 2.5MB. For every frame, first an *inline* curation, checksum and de-duplication is performed. For duplicate frame, previous result is fetched from elasticsearch and new result is stored after corresponding PoVs replacements. Table 1 shows the overhead for every de-duplication task in Sanity. Average overhead was recorded to be $\tilde{5}$ milliseconds. This overhead is equally applicable for duplicate as well as unique data. Since data curation is performed inline with the data stream, all its data accesses happens in-memory, avoiding any disk IO penalties. Even for duplicate data, output data retrieval and storage is done locally to a data store avoiding any network latency. Moreover, duplicate data typically exhibits temporal colocation, thus output data access benefits from caching at data stores.

Computational task	Time
Input data retrieval	772 ms
Function computation	333 ms
Output data storage	59 ms
Total	1165 ms

Table 2: Function execution stats

For unique data, this overhead essentially contributes towards overall latency of function execution. Therefore, the rule of thumb is to maintain data curation overhead to minimum. This can be achieved by hosting deduplication service close to the store, performing data filtering first, then applying only few annotations.

Next, in our experiments we also profile the runtime performance characteristics of our serverless function $f_{vulnerability-check}$, as shown in Table 2. Input data retrieval time accounts for *two* data access query to elasticsearch, namely *frame data* and *thread intelligence*. For frame data it access the whole frame of 2.5MB over network. As described in previous section, input and output data are always persisted in the data store in original form without annotation or filtering. Secondly, *thread intelligence* it accesses data of $\tilde{3}$00KB, which is information about package vulnerabilities collected from standard repository like CVE [8], NVD [18]. Then, during computation it matches any known vulnerabilities from threat intelligence with available packages from container frame. Finally, it stores a vulnerability verdict of $\tilde{2}$00B as output. Overall execution latency for $f_{vulnerability-check}$ is observed to be 1.1 seconds. Important to note that, this latency does not include for container start time before actual function execution is started. According to study [15] container start latency is about 100ms. Although, on most serverless platforms these latencies are obstructed (re)using cache containers.

To summarize, for unique function activations of unique data events Sanity adds an overhead of 5 ms, while for duplicate data events it performs de-duplication of function activations to save redundant execution time of 1 seconds. Therefore, for serverless applications where multiple data events are expected to be duplicate, Sanity can be really effective.

5 Related Work

Data de-duplication is a common practice in large data centers, primarily applied to secondary storage for backup or achieve data to optimize storage capacity [10] [7]. More recently, inline de-duplication is being performed [22] [9] [5] to increase IO performance and reduce capacity requirement for primary storage. All these techniques use content-hash to identify duplicate data. In Sanity, we implement application-aware semantic equivalence to identify duplicate data and our primary objective is to suppress the activation of serverless functions associated with that data.

Another related field of work is incremental computation and memoization [19] [16] [20] that avoid re-execution of functions by caching their results from previous invocations. Sanity practices variant of such memoization techniques wherein for semantically equivalent data, the result from previous execution is retrieved, curated to replace any PoVs from current input data and then used. Thus, content-hash of the output data for multiple invocations of memoized functions are typically different.

Discovering Points of Variabilities or *PoVs* in the data is exercised in the prior work [4] [1] [21] primarily for the configuration data, which are then exploited for understanding semantic state and migration of virtual machines (VMs). In Sanity we are applying *PoVs* to the user data for de-duplication.

6 Conclusion

In this paper, we presented Sanity framework that exploits two fundamental characteristics of a serverless platform, viz. disaggregation of storage and servers, and idempotent constraint on serverless functions, to build a data de-duplication service. De-duplication service leverage application-aware semantic-equivalence to identify duplicate data at storage system and avoids redundant invocation of functions on servers. We also presented design for Sanity system and evaluated it's performance and resource overhead for one real-world use-case of continuous container vulnerability analysis. With modest overhead of 2GB memory it can help improve the latency of function for duplicate data by about 200x.

References

[1] ALBRECHT, J. R., BRAUD, R., DAO, D., TOPILSKI, N., TUTTLE, C., SNOEREN, A. C., AND VAHDAT, A. Remote control: Distributed application configuration, management, and visualization with plush. In *LISA* (2007), vol. 7, pp. 1–19.

[2] AMAZON, INC. `https://aws.amazon.com/lambda/`.

[3] ARNOLD, J. *Openstack swift: Using, administering, and developing for swift object storage.* " O'Reilly Media, Inc.", 2014.

[4] BALANI, R., JESWANI, D., BANERJEE, D., AND VERMA, A. Columbus: Configuration discovery for clouds. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 328–337.

[5] BASU, G., NADGOWDA, S., AND VERMA, A. Lvd: lean virtual disks. In *Proceedings of the 15th International Middleware Conference* (2014), ACM, pp. 25–36.

[6] BORTHAKUR, D., ET AL. Hdfs architecture guide. *Hadoop Apache Project 53* (2008).

[7] CHAPA, D., AND MOFFITT, N. Choosing the right backup technology. *NetApp Tech OnTap* (2008).

[8] COMMON VULNERABILITIES AND EXPOSURES. `https://cve.mitre.org/`.

[9] DOUGLIS, F. A case for end-to-end deduplication. In *Hot Topics in Web Systems and Technologies (HotWeb), 2016 Fourth IEEE Workshop on* (2016), IEEE, pp. 7–13.

[10] DUBOIS, L., AND AMATRUDA, R. Backup and recovery: Accelerating efficiency and driving down it costs using data deduplication. *White Paper, IDC, Retrieved from the Internet:, Accessed on Dec 29*, 2013 (2010), 16.

[11] GOOGLE, INC. `https://cloud.google.com/functions/`.

[12] IBM, INC. `https://console.bluemix.net/openwhisk/`.

[13] JONAS, E., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99%. *arXiv preprint arXiv:1702.04024* (2017).

[14] KOLLER, R., ISCI, C., SUNEJA, S., AND DE LARA, E. Unified monitoring and analytics in the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, 2015), USENIX Association.

[15] KOLLER, R., AND WILLIAMS, D. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), ACM, pp. 169–173.

[16] MICHIE, D. Memo functions and machine learning. *Nature 218*, 5136 (1968), 19–22.

[17] MICROSOFT, INC. `https://functions.azure.com/`.

[18] NATIONAL VULNERABILITY DATABASE. `https://nvd.nist.gov/`.

[19] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. Dryadinc: Reusing work in large-scale computations. In *HotCloud* (2009).

[20] PUGH, W., AND TEITELBAUM, T. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), ACM, pp. 315–328.

[21] SATYANARAYANAN, M., RICHTER, W., AMMONS, G., HARKES, J., AND GOODE, A. The case for content search of vm clouds. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual* (2010), IEEE, pp. 382–387.

[22] SRINIVASAN, K., BISSON, T., GOODSON, G. R., AND VORUGANTI, K. idedup: latency-aware, inline data deduplication for primary storage. In *FAST* (2012), vol. 12, pp. 1–14.

[23] TAK, B., ISCI, C., DURI, S., BILA, N., NADGOWDA, S., AND DORAN, J. Understanding security implications of using containers in the cloud. In *USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX Association, pp. 313–319.

[24] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.