# Chapter Six

# Future Work

In this chapter we discuss future directions that continue this line of research. The first two directions are incremental pieces of work and are more experimental rather than exploratory. Section 6.1 describes a necessary continuation of determining the nature of staleness and how it affects workloads in `/cloud`. Section 6.2 describes the important questions left to answer about `/cloud-history`. Section 6.3 describes a research idea with preliminary validation in the research literature which automatically generates introspection drivers. It would enable rapid, widespread adoption of introspection for any file system if proven scalable and robust. Section 6.4 discusses deriving an introspectable file system from first principles. Research down this path would answer the question, how can the architecture and on-disk layout of a file system be architected to support introspection as a first-class citizen? Ideally, it would not need to compromise on performance, but future work decides whether this is possible or impossible. Finally, Section 6.5 describes creating a filtering language for supporting and implementing file-level policies.

## 6.1 Evaluating the Effect of Staleness in `/cloud`

Remember that `/cloud` is an eventually consistent interface to files. It is at the mercy of the monitored system's file system logic and kernel policies. If the file system batches updates to disk this delays their visibility to `/cloud`. Similarly, if a kernel stores writes in a large page cache for tens of seconds, `/cloud` must wait until the kernel flushes the writes. For example, we have observed during experiments, although not studied in-depth, that Windows and NTFS can take up to 30 seconds and sometimes longer to emit writes to a storage device. Understanding the bound of this staleness, and portion of the file system tree that it affects is important for defining the limitations of `/cloud`. Finding the answer provides a precise meaning to the oft cited, but rarely precisely defined, eventual consistency. While we know `/cloud` is eventually consistent within at most a few minutes from an actual event, what is the lower bound for different operating system and file system combinations? What about the worst case latency? How many files and folders would this affect in practice? Should we tune monitored systems to synchronize more often to persistent storage, or are the eventual consistency bounds reasonable?

Under various workloads, with and without tuning the kernel of the monitored system, we need to collect the following metrics (1) average time from event to `/cloud` visibility, (2) average portion of the file system tree affected by staleness. For the first question, an example worst case answer would be many tens of minutes, rendering `/cloud` less useful. For the second, an example worst case would be the entire file system tree from the root.

## 6.2 Evaluating the Design Decisions of `/cloud-history`

`/cloud-history` has microbenchmarks and early results for all of its major design points. However, a deeper full end-to-end evaluation has not been carried out yet. Here we consider the remaining top questions such an evaluation needs to answer. We came up with he following four top-level questions:

1. How tractable is indexing large amounts of historic data?

2. Using DS-VMI, will ingest rates be enough for modern datacenters?

3. Are temporal-based file versions good enough? Or are content-based versions required to more closely match user expectations?

4. Do users prefer file-level retrieval, or whole-disk restoration when handling archival data? Does the coarse granularity of whole-disk backup render it less useful?

For the first question, we have early validation in the form of the backup study. However, we need to explore more indexing applications to ensure performance remains viable. The second question requires a detailed look at not only the efficiency of the log formats, but also their overall network bandwidth requirements. Many streams may not be scalable, even though one stream is highly amenable to certain optimizations. Per-VM and inter-VM log files are possible—but will they degrade performance past the point of usefulness? We currently have only implemented temporal-based file versioning which, at first glance, seems most intuitive to human users. However, this choice may not match meaningful user-level events such as saves when writes are rapid, reordered, or arbitrarily delayed by the kernel. This requires a study about how well temporal- vs content-based approaches match what the human user expected. However, the capability of rapid indexing may obviate the need for other forms of versioning file-level update logs. Users could use indexing algorithms to define the versions of files which are important to them. In this manner, indexes serve as filters over backup data. Finally, answering the last question—what do users prefer—requires a usability study. Such a study would enable us to make qualified statements about the concrete benefits of file-level indexable backup.

### 6.2.1 Sustained Ingest Bandwidth: Single vs Many Logs

Backup systems have two primary metrics. One is the amount of storage they need to ensure the safety of primary live data. By using compression and deduplication, backup systems can end up using much less storage then they logically contain. The second metric is their ingest bandwidth. In other words, how quickly can they complete a backup? In our discussion of `/cloud-history`, we considered saving a log per file across all monitored file systems. It is currently unclear how much this design decision will affect the ingest bandwidth of `/cloud-history`. Future experiments should tease out the effect of merging file-update streams at various levels. For example, the file-update streams of an entire file system could be merged together into one stream. Merging streams may incur a cost on other operations such as garbage collection.

### 6.2.2 Measuring Time to Index

Preliminary results show that file-level deduplication is a clear win for cutting down the time to index. What remains to be answered is how does storing file data within file update log streams affect indexing time? Do the logs hinder or aid indexers? Presumably, reconstruction time of historic file versions will negatively impact the time to index. File update logs should be compared against other techniques for storing versions of files such as copy-on-write file systems implemented with tree structures such as btrfs [99]. We expect file update logs to perform comparable to a log-structured file system, and they can be directly compared to the NILFS [82] file system.

### 6.2.3 Effect of Hashing Choice

We proposed using incremental hashing for computing whole file hashes which theoretically outperforms both traditional hashing and Merkle trees for random updates. This performance edge on handling random updates makes incremental hashing an ideal candidate for computing a hash on file update log streams with low latency. This low latency directly translates into higher achievable bandwidth over streams of random updates, even when performed offline. Although we explored incremental hashing for its perfect computational fit, it actually should exhibit a higher ingest rate—one of our key metrics—of file update logs. Head-to-head experiments should be done comparing the various hashing methods performance on recomputing a hash with random updates. The key metric is bandwidth in terms of bytes processed per unit time. A secondary metric is the space used to store the resulting hash. For example, a Merkle tree needs space to store a tree of hashes.

## 6.3 Generating Introspection Code from File System Drivers

Early validation by Virtuoso [34] shows that introspection code can be generated with a minimal helper application inside the monitored system during a training phase. This thread is very promising, because it hints at a mechanized process for the current very manual engineering task of writing introspection code. Indeed, the ability to generate such code deterministically implies that the file system is sound and consistent. This introspection code serves as a run-time check or hypervisor-verifiable proof that the file system is running correctly.

Generating introspection code by hand is a major roadblock impeding the progress of general introspection across modern clouds today—especially for closed-source kernels. An approach that is machine-performed, and thereby scalable, would completely change the introspection landscape and provide a new wave of tooling for clouds to leverage. As we have shown in this dissertation, scalable new interfaces to cloud-wide persistent state then become possible, and mechanized generation of introspection code makes scalability across OS versions and types trivial.

Introspection code generation works presently via two main techniques. The first technique could be considered supervised with access to ground truth in the form of kernel source code. It has so far worked with fair success [41]. The second technique is unsupervised without access to kernel source code. Generally, this method works with a "helper" application [34] running inside the guest writing known patterns to virtual storage. The introspection generator watches for these patterns and then infers the relationship between blocks on disk. This technique has also met with limited success in the past. A breakthrough maturing one of these techniques is necessary to make introspection a status quo reality in the future.

## 6.4 Designing an Introspectable File System

As far as we know, no file system has ever been developed within the context of virtual computing with introspection as a supported feature. Developing an introspectable file system is compelling for debugging purposes. Introspection enables run-time verification of writes as they are flushed from a kernel to storage devices. Thus, an introspectable file system becomes easier to develop, debug, and externally verify. The closest known work in the research literature is ReconFS [67]. ReconFS embeds inverted pointers with every written page. These pointers are written in the scratch space of flash pages. Using these pointers, the file system can be reconstructed even if core datastructures are damaged.

Reverse pointers as in ReconFS, especially with every single page, would immensely ease the implementation of introspection. It seems possible for a small augmentation which could be retrofitted to legacy file systems making them introspection-friendly. We need just enough information to understand the type of a block and how it fits into the overall file system. However, the absolute minimum mapping we want is from block to file. Thus, the only true pointers we need are from a block to a byte stream within a file, and a notion of the full path of a file. Both of which could be embedded into the notional "scratch" space of a page on a flash device, or added as a header to every block on the storage device. This reverse mapping, as in ReconFS, also makes the file system resilient to failures even those which wipe out core file system data structures.

The minimal data structure for a data block is (1) file byte offset start, (2) file final block flag, (3) file offset finish which would only be present for final blocks, and the file's name or a pointer into a table storing the absolute path. Ideally the path information would be embedded as part of the reverse mapping data structure as well. Directory blocks have a flag set denoting their status as being a list of path information. Directories could be spread throughout the disk as other files are, or centrally located in a large table like NTFS's MFT.

## 6.5 A Storage Introspection Language for Policy Enforcement

This dissertation was entirely concerned with the mechanism of file-level data capture, its implementation with multiple interfaces on top, and its evaluation. In Chapter 1 we described what was out of scope for this dissertation and one such topic was enforcement of policy via introspection. However, paralleling networking research, development of a filter language and mechanism over storage traffic seems inevitable for enforcement of policies. Also, having such a language makes it easier to implement introspection tasks. It would be desirable if the language was flexible enough to implement introspection. Even better if the language itself could make implementing introspection for a new file system faster than without it.

There are unique challenges which distinguish creating such a language, and enforcing it on a write stream, from the creation of the network community's packet filtering languages. First, many writes often need to be buffered before the overall operation within the file system can be understood. This is in contrast to network protocols such as TCP which have sequence numbers clearly defining the number of outstanding writes and order of a stream. Second, there are many more file systems in use than network protocols. Predominantly, the standard protocols in networking are less than a handful: IP, TCP, UDP, and today some would add HTTP. However, the file systems in use today are more than a handful and include FAT32, NTFS, ext4, ZFS, ReiserFS, XFS, NILFS, Btrfs, GPFS2, and many more. Each one has its own datastructures, on-disk layout, and algorithms for sending data to disk. In addition, writes to file systems are wrapped within a hardware-specific protocol such as SATA or IDE. These factors combine and force a rethink of packet filter and the design of policy languages when applied to storage.