Chapter Three

`cloud-inotify`: Cloud-wide File Events

Monitoring for file system changes comes in two flavors: polling by scanning directories, or registering for changes via event-driven callbacks. In this chapter we focus on the latter—events—and describe an interface built on top of DS-VMI, the mechanism developed in the previous chapter. In the context of file systems, polling equates to batch-style scans through directory trees. Often, tools such as `rsync` [73] check the modified timestamp on files to determine whether or not new changes need processing. However, for tools such as Dropbox [36]—a file-level versioning and synchronization tool—rapidly scanning directory trees as they grow in size to tens of thousands of files becomes increasingly costly. Thus, the capability of registering for state changes and receiving notification when they occur is a critical capability for certain file-level monitoring workloads.

`inotify` [70] is a Linux kernel notification interface for local file system events. It provides an API to userspace applications that lets them register for events such as directory updates, file modifications, and metadata changes. Via callbacks, the userspace applications act on these events. `inotify` has been used to implement desktop search utilities, backup programs, synchronization tools, log file monitors, and many more file-level event-driven applications. Using DS-VMI, we extend the concept of `inotify` into what we call `cloud-inotify`. There are three major differences between `cloud-inotify` and `inotify`. First, in the actual interface: `cloud-inotify` is a publish-subscribe network-based messaging system, whereas `inotify` is a Linux system call API. Second, `cloud-inotify` lets users subscribe to updates across many remote file systems, but `inotify` only works for local file systems. Third, `cloud-inotify` is, as much as possible, OS agnostic, but `inotify` only works on Linux systems. Naturally, other operating systems such as Microsoft's Windows and Apple's OS X have their own versions of file-level events called FileSystemWatcher [77] and FSEvents [8] respectively. But, each interface is vendor-specific and requires specialization. `cloud-inotify` requires writing file-level event logic once, and instantly works across different OS types.

`cloud-inotify` is the first of three interfaces built on top of DS-VMI that this dissertation describes. It is designed for event-driven workloads such as log analytics, intrusion detection, or file synchronization. For example, a common use case is registering for updates to a directory and its subtree in the overall file system tree. Any updates to that directory, its sub-

directories, and files contained therein gets replicated via messages over the network to the registered monitoring application. Using such technology, one could rapidly build folder-level synchronization and backup. Another use case is registering for events on critical system files such as `/etc/passwd` which contains account credentials on UNIX-based systems. Or following any changes to `C:/Windows/system32/config/SAM` which contains account credentials on Windows.

The rest of this chapter is organized as follows. Section 3.1 describes the design and implementation of the `cloud-inotify` interface. Section 3.2 goes into more detail about the types of workloads we envision using `cloud-inotify`. Section 3.3 describes potential sources of latency which could delay notification of file-level events to registered monitors. Section 3.4 provides an evaluation of the sources of latency in the `cloud-inotify` system. Latency is the primary metric by which we evaluate `cloud-inotify`'s performance. The final Section 3.5 provides an example real-world end-to-end test using a web browser and a research OpenStack [85] cloud cluster.

3.1 `cloud-inotify`'s Design and Implementation

`cloud-inotify` provides a network-accessible, publish-subscribe channel abstraction enabling selective monitoring of file-level update streams. Applications "register" for events by connecting to a network socket and subscribing to channels of interest. Via published messages, they are notified of individual events and take action. We call applications implemented using file-level events *monitors*.

Channel	Description
`gs9671:test:/var/log/*`	Monitor all files in subtree `/var/log` in VM instance test on host gs9671
`*:*:/var/log/*`	Monitor all files in subtree `/var/log` in all VM instances on all hosts
`gs9671:*:/var/log/auth.log`	Monitor `auth.log` on all VM instances on host gs9671
`gs9671:test:/var/log/syslog`	Monitor `syslog` on VM instance test on host gs9671

Table 3.1: Examples of filter specifications, demonstrating the use of pattern matching. The `cloud-inotify` channel implementation supports glob-style pattern matching.

`cloud-inotify` channel names are a combination of three components: the cloud-internal hostname of a compute node hosting VMs, a VM or name referencing a group of VMs, and the full path of interest in the guest file system of the targeted VM instances. Example channels are shown in Table 3.1. Any updates not associated with a specific pathname, such as superblock or MBR updates, are emitted without a path component. `cloud-inotify` allows wildcard characters when subscribing to channels. Monitors can easily subscribe to a variety of events without exposing themselves to a firehose of irrelevant notifications. This improves scalability by reducing the volume of frivolous data transmitted across a cloud and potentially externally across the WAN. When monitors subscribe from outside of a cloud, they do not specify the first component—a hostname—of the channel.

```
import bson, gray
name = 'gs9671:test:/var/log/auth.log'
channel = gray.subscribe('gray.example.com',
                         name)
for msg in channel:
    msg = bson.deserialize(msg.data)
    if msg['type'] == 'data':
        print ('New write from %d to %d' + \
              'for file %s') % (msg['start'],
                                msg['end'],
                                msg['path'])
        print 'Data: %s' % (msg['data'])
```

Figure 3.1: An example monitor in Python.

Monitors are typically application-specific, and each monitor is typically interested only in a small subset of file update activity in a VM instance. In some cases, a single monitor may receive file update streams from many VM instances and thus perform cloud-wide monitoring for a specific application. In other cases, a monitor may be dedicated to a single VM instance. Monitors interact with `cloud-inotify` via a cloud-provided WebSocket API. After DS-VMI is initiated for a VM instance, `cloud-inotify` channels become available via WebSockets. The WebSockets are provided by a front-end cloud proxy which translates subscription requests from individual WebSockets into Redis subscriptions on back-end compute hosts running VMs under the purview of DS-VMI. Access control is gated by the cloud, which also has the ability to deny subscription requests. This means cloud users can create filters allowing monitors access to subsets of file-level events originating from their VM instances. The use of a front-end proxy frees the back-end cloud to implement `cloud-inotify` using any optimizations it needs. For example, subscriptions could be filtered and aggregated at multiple levels across a cloud datacenter which improves scalability.

We call this fine-grained access control a *data firewall*. Strong guarantees are possible precisely because the cloud infrastructure acts as a mediator between cloud customers and monitor providers. Cloud customers could declaratively configure access to their data at a file-level exactly the same way they configure network firewall rules today and apply them to VM instances. Though we did not implement such enforcement and considered it out of scope for this dissertation, it is still a part of the overall vision and the hooks are in place for implementing enforcement across the DS-VMI stack. On the other side of the equation, engineering monitors becomes a simpler process. By programming to a unified publish-subscribe API, monitor vendors no longer have to implement support for several versions of operating system environments. They implement just once targeting a REST API and WebSockets channels.

We use the publish-subscribe capability of Redis to implement cloud-inotify channels. A monitor connects to Redis' well known TCP port and subscribes to channels using filters similar to those shown in Table 3.1. The monitor then receives BSON-serialized messages relevant to its filter specification, each containing the changed metadata fields in the case of a metadata update, or the corresponding file data in the case of a data update. BSON libraries are available in a number of programming languages; Figure 3.1 shows an example of a simple agent written in Python for monitoring /var/log/auth.log. The code has no OS-specific dependencies, no file-system-specific logic, and is easily re-targeted based on the channel subscription expression.

Libraries for consuming such messages exist in most modern programming languages. We selected Python to form a proxy bridge between OpenStack compute nodes in a research cloud, and monitors. Python fits well within the OpenStack ecosystem because OpenStack is a pure Python-based open source cloud software. Based on OpenStack credentials, access control filters can be placed in this front-end proxy. These filters could provide strong guarantees on the type and extent of data made available to external third-party agents. Such a capability is impossible to achieve with in-VM agents because they often require root access and it is very difficult to bound their access to information. By enabling fine-grained, strong access control guarantees, cloud-inotify enables new possibilities in interactions between cloud customers and file-based monitoring services.

3.2 Event-driven File System Workloads

Event-driven workloads are characterized by a tight bound on the time between an event occurring, and notification of that occurrence propagating. This near-real-time constraint makes cloud-inotify challenging to implement because introspection may not have a lot of time to complete, otherwise it risks introducing latency. Many types of workloads are subsumed in this category including file-based alerts, log analytics, intrusion detection, and events act as triggers to longer running batch jobs as described in Chapter 4.

In this section we demonstrate the usefulness of cloud-inotify and how event-driven workloads are ubiquitous. We demonstrate this via two applications for cloud-inotify. We currently expose cloud-inotify via a WebSocket proxy as an OpenStack front end. This proxy translates the BSON-serialized file-level update stream into a JSON-serialized WebSocket message stream for web browsers or other clients consumption. Using this interface we can perform fine-grained access-control via OpenStack on individual update messages, although this is currently not implemented. Two use cases which are implemented already via WebSockets are described below.

3.2.1 Continuous Compliance Monitoring

Auditing file-level changes is useful for enforcing policy, monitoring for misconfigurations, and watching for intruders. For example, many businesses monitor employee computers for properly licensed software. Using cloud-inotify, enterprises gain visibility into all systems across their fleet without worrying about OS-specific implementations or deployments of monitoring software.

`cloud-inotify` has a centralized design focused on aggregating the updates of file systems across a cloud. In addition to the aforementioned benefits, `cloud-inotify` cannot be turned off, tampered with, or misconfigured by guests unlike agent-based solutions. Centrally-managed auditing ensures that all VMs are checked for the most recent security updates and best practices as well. Example checks include: proper permission bits on important folders and files, and monitoring `/etc/passwd` to detect new users or modifications to existing users. Google [27] reported an outage in early 2011 that affected 15-20% of its production fleet. The root cause was a permissions change to a folder on the path to the dynamic loader. Localizing such problems within a short amount of time can be difficult, but an auditor built using `cloud-inotify` would have detected the misconfiguration almost instantaneously. Google found troubleshooting difficult because logging into affected servers was impossible. `cloud-inotify` does not depend on guest health; thus, cloud customers never have to "fly blind" even if they cannot log into VM instances.

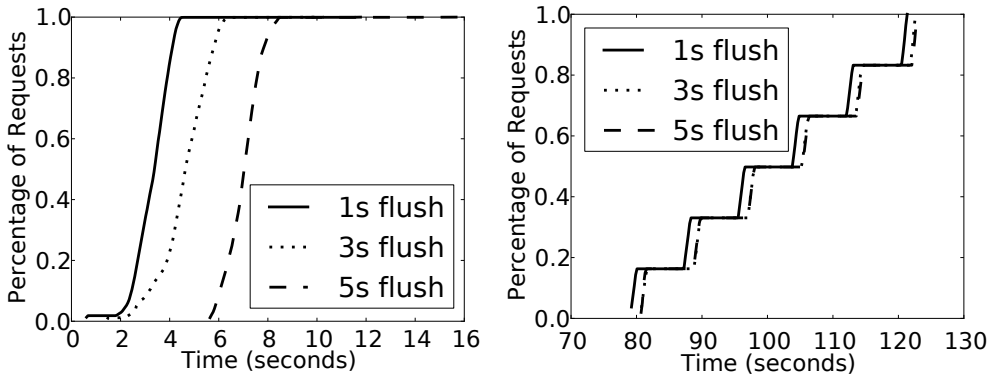
3.2.2 Real-time Log Analytics

Log files contain insights into the health of systems and the responsiveness of their applications. An example of such an insight is response time derived from web application log files. In this example we consider not just engineering monitoring solutions from the perspective of the cloud user or a monitor vendor, but also the usefulness of `cloud-inotify` for directly implementing cloud infrastructure features. One straightforward intuition is if the infrastructure has an understanding of application performance, it can more intelligently assign resources across the cloud. Sangpetch et al. [102] show that an application-performance-aware cloud can double its consolidation while lowering response time variance for customer applications. They measured response time based on network traffic; however, encrypted flows and applications not tied to network flows cannot benefit from this feedback loop. Normally, the opacity of a VM requires resorting to indirect measures such as inspecting network packets to measure response time. With `cloud-inotify`, the same metric can be derived from accurate information directly from application logs.

3.3 Sources of Latency

The sources of latency are primarily parameters inside the guest kernel and outside of our control. DS-VMI does its best, but it only receives writes when the guest kernel emits them to the virtual storage layer and not before. Thus, the fundamental limits on latency are gust kernel flush interval, DS-VMI processing time, and message propagation time. We assume the guest kernel is outside of our control, although as discussed in Chapter 2 future guests will likely cooperate with cloud infrastructure.

The guest kernel introduces latency via two primary mechanisms: the page cache, and write reordering. Page caches are used to batch updates in fast memory destined for slow persistent storage. Page caches primarily speedup small write workloads. Page caches also serve as fast access for reads, and they rely on temporal locality to maintain high hit ratios. Write reordering occurs



(a) Sync every second in the guest VM.

(b) No in-guest syncing.

Figure 3.2: Latency CDFs demonstrating feasibility of near-real-time event-driven agentless monitoring using `cloud-inotify`.

when the kernel takes license to optimally order writes without impacting correctness. A famous example of write reordering is the Elevator algorithm [62] which minimizes parameters such as seek time, by taking the liberty to reorder recent writes queued for storage. An even simpler policy of just writing the next sector with shortest seek time clearly leads to reordering of writes. A kernel has the undesirable job of balancing recoverability, correctness, and performance. Though we can not blame the kernel for optimizing where it finds opportunity, tighter cooperation between a kernel and its underlying hypervisor would greatly aid introspection. However, the theme of this dissertation is to require zero guest cooperation—thus, we must live with the reordering of the guest kernel and related latencies.

3.4 Evaluation of Latency

Here we evaluate latency on real-world workloads as observed via monitoring log files. We monitored `/var/log/httpd/access.log` and found negligible delay at the time granularity we cared about—on the order of one second. That is, DS-VMI introduces negligible delay from the emitting of a storage write, to its introspected form.

Figure 3.2 shows the results of $10,000$ requests during the microbenchmark. Figure 3.2(a) shows the best case when the guest frequently syncs data to disk with a latency of 1, 3, or 5 seconds on average. Figure 3.2(b) shows an untuned guest where latency is at the mercy of guest kernel I/O algorithms which flush at much lower frequency than the latencies we tuned. The step-like nature is because many updates appear at once—many log file lines fit inside a single file system block. A representative monitoring system such as Akamai's [24] is an example that would tolerate these latencies without any tuning of guests, but low latency performance monitoring [102] may require tuning of guest flush algorithms.

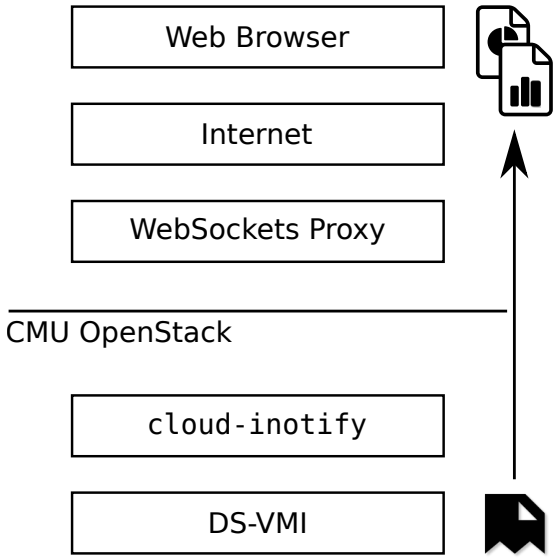


Figure 3.3: Writes are introspected by DS-VMI. DS-VMI was activated by a cloud user using a standard OpenStack command-line utility extended to support our DS-VMI cloud API. Emitted file-level updates are sent to the user via a front-end WebSockets [38] proxy over the Internet to a web browser.

This is a positive result overall for applying DS-VMI technology to implement a near-real-time system such as `cloud-inotify` on top. With zero tuning and zero guest configuration the latencies are tolerable. In addition, the DS-VMI framework shows negligible overhead in terms of latency. Especially for an untuned guest—the timeouts inside the guest kernel dominate the latency equation in this case.

3.5 Using `cloud-inotify` in a Research Cloud

As laid out in Chapter 2, we implemented DS-VMI within an OpenStack cloud computing cluster at Carnegie Mellon University (CMU). This cluster had between 15-20 healthy compute hosts over the course of the DS-VMI deployment. Each host had a modified hypervisor capable of duplicating writes to our DS-VMI framework. In this section, we demonstrate a working end-to-end application using this research cloud.

Figure 3.3, shows the flow of file-level updates as they traverse the cloud boundary. Users activate DS-VMI using a modified OpenStack command-line utility. The modified utility supports our DS-VMI API extensions residing inside the OpenStack API server. Once activated, the compute host responsible for this VM sends a command to the KVM hypervisor activating duplication of virtual disk writes. The writes are duplicated to DS-VMI which transforms them into file-level updates and emits them on channels for listening subscribers. The virtual machine guest in this

Figure 3.4: Textual display of file-level updates affecting `/var/log/syslog` within an unmodified executing Ubuntu 14.04 Server. Red text denotes the affected byte range, blue text is the file contents being written, and the bottom black text is a metadata update.

case is an unmodified Ubuntu 14.04 Server 64-bit cloud computing image [17]. Our user's web browser has subscribed to some number of channels and receives updates over a WebSocket [38]. Currently, only a textual display is supported in the browser. However, by using web technologies we enable the development of GUIs utilizing the full power of the modern web.

Figure 3.4, shows the textual view presented to the user in their browser. Google Chrome was used in this demo as the web browser. In this demo, we subscribed to the `/var/log/syslog` log file inside the unmodified Ubuntu Server guest virtual machine. The red text denotes the bytes in the file being updated. The blue text represents the data being written into the file. The output respects the recorded file size, and does not print extra bytes even if the actual data written exceeds the size of the file. The black text at the bottom represents a metadata update. Note, that the file system, `ext4`, safely updates metadata *after* writing data to disk. This safe ordering of operations led to the temporal gap challenge as mentioned in Chapter 1. Also note, that metadata updates are assigned a transaction number. This is because block-level writes with granularity larger than a single file system data structure often contain many file-level metadata updates.