OpVis: Extensible, Cross-platform Operational Visibility and Analytics for Cloud

Fabio A Oliveira, Sahil Suneja, Shripad Nadgowda, Priya Nagpurkar, Canturk Isci
IBM T.J. Watson Research
{fabolive,suneja,nadgowda,pnagpurkar,canturk}@us.ibm.com

Abstract

Operational visibility is an important administrative capability and a critical factor in deciding the success or failure of a cloud service. It is becoming increasingly complex along many dimensions including the need to track both persistent and volatile system state across heterogeneous endpoints, as well as provide higher level services such as log analytics, software discovery, anomaly detection, and drift analysis. In this paper we present *OpVis*, our unified monitoring and analytics framework to provide operational visibility, which overcomes the limitations of traditional monitoring solutions and provides a uniform platform as opposed to requiring the configuration, installation, and maintenance of multiple isolated solutions. We highlight our framework's extensibility model, enabling custom data collection and analytics based on the cloud user's requirements, describe its monitoring and analytics capabilities, present performance measurements, and discuss our experiences while supporting operational visibility in our cloud.

CCS Concepts • **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**; *Virtual machines*; • **General and reference →** *Performance*; • **Information systems →** Document collection models; Distributed retrieval; *Information extraction*; *Document filtering*;

1 Introduction

Traditionally, the operational visibility practices have been limited to resource monitoring, collection of metrics and logs, and security compliance checks on the underlying environment. In today's world, better equipped to manipulate massive amounts of data and to extract insights from it using sophisticated analytics algorithms or machine-learning techniques, it becomes natural to broaden the scope of operational visibility to enable, for instance, deep log analytics, software discovery, network/behavioral anomaly detection, configuration drift analysis, to name a few use cases.

To enable these analytics, however, we need to collect data from a broader range of data sources. Logs and metrics no longer suffice. For example, malware analysis is done based on memory and filesystem metadata; vulnerability scanning needs filesystem data; network analysis requires data on network connections; and so on. At the same time, these data sources are potentially very different in nature. Log events are typically continuously streamed, whereas filesystem data changes are less frequent, and configuration changes normally occur when an application is deployed.

Middleware'17, Las Vegas, Nevada USA

DOI: 10.1145/nnnnnnn.nnnnnnn

Yet another source of data for modern operational visibility stems from the diverse and prolific image economy (DockerHub, Amazon Marketplace, IBM Bluemix) that we witness as a result of pervasive virtualization. The more cloud images are used, the more important it becomes to proactively and automatically certify them by performing security and compliance validation, which requires visibility into dormant artifacts, in addition to running cloud instances.

Adding to the complexity of dealing with a multitude of data types for modern operational visibility, cloud environments are becoming larger and increasingly heterogeneous. For instance, it is nowadays common for a cloud provider to support deployments on physical hosts, virtual machines (VMs), containers, and unikernels, all at the same time. As a result, for more effective visibility, operational data from this diverse set of runtimes needs to be properly collected, interpreted, and contextualized.

As if heterogeneity were not enough, the lighter the virtualization unit (e.g., containers and unikernels), the higher the deployment density, which leads to a sharp increase in the number of endpoints to be monitored. Figure 1 summarizes the complexity of modern cloud environments along multiple dimensions, including deployment types and cloud runtimes, as well as some challenges for which operational visibility is needed.

In this paper, we propose a novel approach to operational visibility to tackle the above challenges. To enable increasingly sophisticated analytics that require an ever-growing set of data sources, we implemented *OpVis*, an **extensible** framework for operational visibility and analytics. Importantly, *OpVis* provides a **unified** view of all collected data from multiple data sources and different cloud runtimes/platforms. *OpVis* is extensible with respect to both data collection and analytics.

As opposed to traditional solutions, to scale to the increasing proliferation of ephemeral, short-lived instances in today's high-density clouds, we propose an **agentless**, non-intrusive data collection approach. We further employ a holistic approach to analytics by decoupling it from data collection, thereby uncovering relationships across otherwise separated data silos.

Our implementation of *OpVis* supports multiple data sources and cloud runtimes. We have been using it in a public production cloud environment for over two years to provide operational visibility capabilities and several analytics applications to, among other things, support security-related services to our cloud users.

We evaluate *OpVis* with both controlled experiments and real production data in cases where we were allowed to publicize it. An extended version of this paper is available as a technical report [18].

2 Existing techniques

To gain visibility inside VMs (beyond just black-box metrics), most existing solutions typically require agent installation inside the monitored endpoint's runtime, and thus cause guest intrusion and interference [1], as well as VM specialization [16, 29–31].

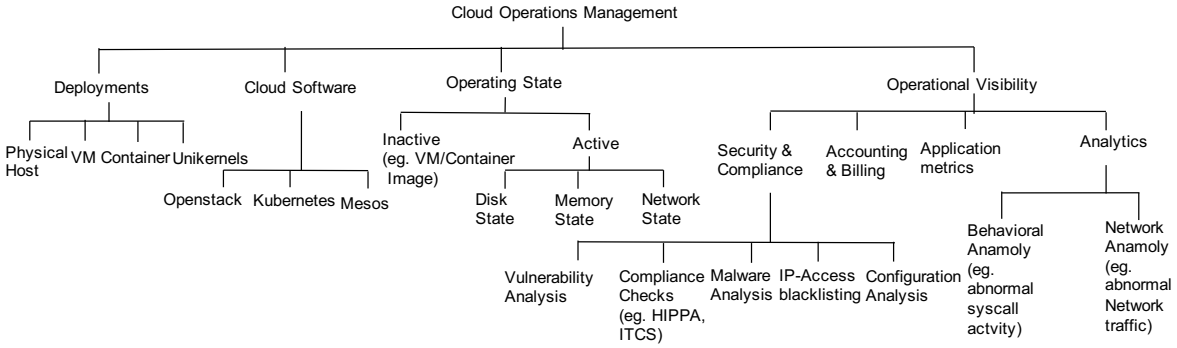

Figure 1. Cloud operation management.

The landscape is a bit different with containers, since the semantic gap between the guest environment and the management layer (host) is greatly reduced. In addition to container-image scanning [2, 6, 22, 23, 28], some solutions are able to provide some level of agentless, out-of-band container inspection by talking to the Docker daemon [9, 24], querying kernel's cgroups stats [5], monitoring containers' *rootfs* [14], or tracing system calls [26]. Although these solutions can provide basic metrics, for deep inspection most resort to installing agents, plugins, scripts, instrumentation libraries, or custom exporters [8, 21, 24] inside the guests.

Furthermore, most solutions only partially address operational visibility. The ones that seem to cover all aspects among image scanning, out-of-band basic metrics, and deep inspection [14, 22, 28] are not open-sourced or extensible.

To the best of our knowledge, no existing solution provides all of OpVis' capabilities of a unified, agentless, decoupled, extensible, and open-sourced operational-visibility framework, which does not enforce guest cooperation or cause guest intrusion, interference, and modification to cover the entire operational visibility spectrum depicted by Figure 1.

3 Design and implementation

In this section we describe the design and implementation of *OpVis*, our unified operational visibility and analytics framework. The overall architecture of *OpVis* is depicted by Figure 2, which clearly separates three layers: data collection, data service, and analytics. We refer to *OpVis* data collectors as *crawlers* (see top of Figure 2). They monitor cloud instances and images to take periodic memory-state and persistent-state snapshots, which are then encoded into an extensible data format we refer to as the *frame*. In addition to discrete state snapshots, the crawlers track log files of interest from cloud instances. Snapshots, in the form of *frames*, and streaming log events enter the data service through a scalable data bus from which they are fetched and then indexed on a data store for persistence. A search service makes all collected data available and queriable, enabling a variety of analytics applications, for instance, to diagnose problems experienced by a cloud application, to discover relationships among application components, and to detect security vulnerabilities. The rest of this Section describes the *OpVis* data collectors (§3.1), data format (*frame*) for discretized state snapshots (§3.2), and backend data service (§3.3). We also present a few analytics applications that take advantage of *OpVis* (§4).

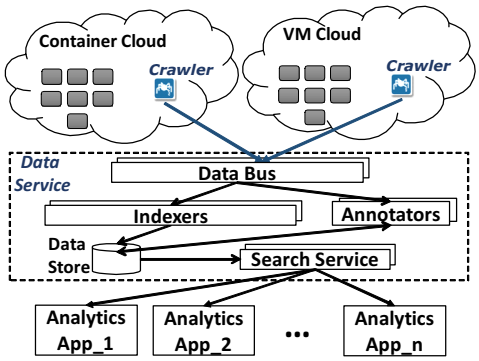

Figure 2. *OpVis* overview.

3.1 Data collectors: agentless *crawlers*

We take an *agentless* approach to data collection, that is, *OpVis* crawlers collect data in an *out-of-band*, *non-intrusive* manner. Critically, an important role of the crawlers is to enable operational visibility with a *unified* view across different cloud-runtime types and for different forms of application and system state. We implement out-of-band visibility into container runtimes, e.g., plain Docker host, Kubernetes, and Mesos, and into VM runtimes, e.g., OpenStack.

We observe that monitoring live cloud instances (containers and VMs) is important for reactive analytics; however, to enable proactive analytics applications it is equally important to also scan cloud images (Docker images and VM disks). For this reason, *OpVis* crawlers provide visibility into these dormant artifacts as well.

To enable semantically-rich end-to-end visibility and analytics, the crawlers collect in-memory, live system state, e.g., resource usage and running-processes information, as well as persistent system state, e.g., filesystem data and logs. This broad range of data types requires proper manipulation of continuously-streaming data, such logs, as well as state that needs to be taken at discrete snapshots, such as process, network, and filesystem information.

Exposing and interpreting persistent and volatile state of VMs and containers requires techniques specific to each runtime and state type. Broadly, we apply introspection and namespace-mapping techniques for VMs and containers, respectively. Despite slight differences in the techniques' nuances, the key tenet of our approach remains unchanged: to provide deep operational visibility in near real time and out of band, with no intrusion or side effects on the cloud instances. Next, we delve into our implemented techniques.

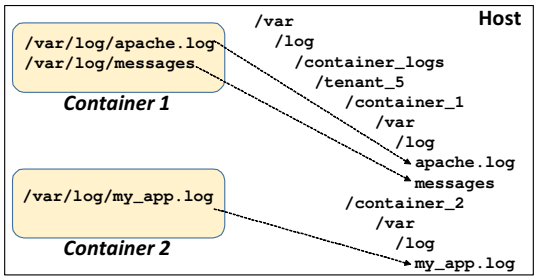

Figure 3. Mapping log files from containers to the host.

3.1.1 Container crawling

As container processes are simply host processes with a different view of the system, they are visible from the host. We use two OS virtualization techniques within the Linux kernel to collect containers' memory-state information: (1) the cgroups accounting interface for resource utilization, and (2) the namespace APIs for details on processes and network connections in the container context.

To collect persistent-state data, our crawlers identify the location of each Docker container's root filesystem in the host filesystem and then extract containers' filesystem data and metadata, information on configuration files, and installed packages. The corresponding state from dormant images is extracted similarly, by creating a dummy container when a new image is pushed to the cloud or an existing one is modified. Another technique is to mount the image on the crawler host and perform the inspection offline.

To deal with log events, all log files of interest in the guest containers are mapped onto a special location in the host filesystem, from which all log events are continuously tracked and streamed to the data service in near real time via Logstash [11]. Figure 3 illustrates this process. The top directory of all monitored log files in the host is `/var/log/container_logs`, where one subdirectory per cloud tenant (user) gets created. Under a tenant subdirectory, one directory per container is created when a newly-created container is discovered. New log files to be monitored are independently discovered by watching recursively the contents of `/var/log/container_logs/**/*`, which is populated by a separate process that identifies the log files to be tracked through a per-container *LOG_LOCATIONS* environment variable and creates symbolic links to them from the container filesystem.

These techniques provide us with the non-intrusiveness character we seek, by not requiring any special agent or library inside the containers. The approach works even if the container is unresponsive or compromised, since the crawler still gets the overall system view from outside the container.

3.1.2 VM crawling

Since VMs have their own OS kernel and therefore keep their internal memory state hidden from the host, they are more difficult to monitor than containers. We use and extend VM introspection (VMI) techniques [15] to gain an out-of-band view of VM runtime state. We have developed solutions to expose VMs' live memory with negligible overheads for KVM VMs. Since KVM is part of a standard Linux environment, we leverage Linux memory management primitives and access VM memory via QEMU process' `/proc/<pid>/mem` pseudo-file, indexed by the virtual address space backing the VM's memory from `/proc/<pid>/maps`. The logical interpretation of this raw, byte-array memory view is achieved via in-memory kernel data structure traversal. Further details can be found in our previous work [25].

Exposing and collecting a VM's persistent state non-intrusively requires VM disk introspection. Our design follows certain key principles: (1) the persistent-state collection of offline and live VMs must be identical; (2) it must have negligible impact on the VM's runtime; and (3) it must be done from outside the VM and not lead to any runtime or persistent-state change.

First, we use OpenStack and QEMU APIs to determine disk layers for running VMs. Next, as the VMs are running while the crawler accesses the disks out of band, we expose all identified disk layers as *read-only* pseudo-devices to ensure that no action can alter the device state at the physical level. Moreover, because the disks being accessed out of band are live and hence inherently *dirty*, we use Linux device mapper reverse snapshots to wrap each pseudo-device with a separate Copy-on-Write (CoW) layer. Then, this new device view can be exposed as either a local storage device or a network-attached one (e.g., iSCSI). Finally, the exposed device is mounted by the crawler process so that it can access the target VM's filesystem over the entire device view to collect the exposed persistent state, such as configuration files, installed packages, etc.

This technique works for both raw and QCoW2 images, and can be applied uniformly to offline and live VMs. After a VM's filesystem is exposed by the above actions, log streaming follows the same approach as in the container case.

3.1.3 Crawler extensibility

Extensibility is a key design principle for the OpVis crawler. We envisioned various sources of heterogeneity (Figure 1) in the operating environment, configurations, and monitored endpoints. As a result, we adopted the plugin architecture shown in Figure 4.

Crawl plugins. Logic for every data type collected by the crawler is implemented as a separate crawl plugin. This allows flexibility in configuring the crawler to selectively collect different data for different environments. Crawl plugins can extract both application and system state. A well-defined interface makes it easy to extend *OpVis* with a new data type and to implement a crawl plugin responsible for its collection. A similar extensibility model is supported for emitters, filters, and environment plugins.

Emitter plugins. The *OpVis* crawler supports combinations of different output data formats (e.g., CSV, JSON, Graphite) and target endpoint types (e.g., stdout, file, HTTP, Fluentd, Kafka), allowing it to cater to various types of specialized data stores for analytics and monitoring.

Environment plugins. Environment information is an orthogonal dimension for data collection but is important to establish the context for the entity being crawled. For example, multiple containers from different subnets could have the same IP address; therefore, in addition to collecting network state of a container, the tenancy information and network topology is important to resolve and further analyze the collected data. Environment plugins provide this contextual information by adding structured metadata to each state snapshot containing, among other things, environment-specific elements such as container labels and pod ids in the case of a Kubernetes environment.

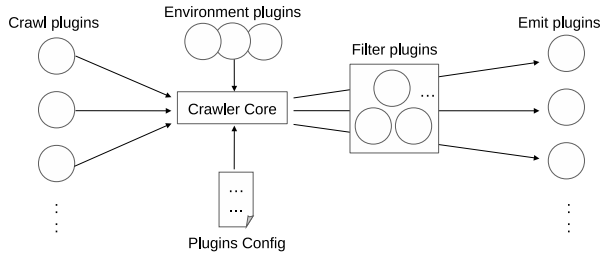

Figure 4. OpVis crawler's plugin architecture

Filter plugins. These plugins provide data aggregation and filtering capabilities atop the extracted data. Examples include the metrics aggregator plugins (min, max, average), as well as `diff` plugins to send only deltas to the backend.

3.2 Frame: state snapshot

A *frame* is a structured representation of a system snapshot, encompassing memory and persistent state, taken by the crawler using the data collection techniques previously presented. Log events are not part of a frame, as they are streamed rather than discretized.

We refer to each element in a frame as a *feature*, which in turn embodies a collection of key-value pairs where the keys are *feature attributes*. The set of attributes of a feature depends on the *feature type*. Each feature type corresponds to a type of memory or persistent state collected by the crawler. The feature types we define include: `os`, representing general information on the operating system, `process`, corresponding to OS processes, `connection`, encapsulating information on network connections, `file`, associated with metadata of filesystem objects, `config`, representing the contents of filesystem objects identified as configuration files, and `package`, for metadata on OS-level and programming-language-level packages. We also define types for discretizing resource-usage metrics, e.g., for CPU and memory. Our framework can be extended with a new type declaration and the corresponding crawl plugin.

In addition to features, a frame has metadata to capture important aspects of the snapshot. In particular, a *timestamp* indicates when the snapshot was taken, allowing analytics applications to run temporal queries to reason about state evolution. Also, associated with a frame is a *namespace*, which is used to identify the cloud instance in question, typically as a combination of a cloud-assigned id and a user-provided string with a name and version of the cloud application/service. Other pieces of metadata provide provenance information to identify image versions and cloud users.

3.3 Data service backend

The *OpVis* data service is illustrated in the middle part of Figure 2. It comprises a data pipeline whose entry point is a scalable, replicated, and fault-tolerant data bus. To realize our data-bus cluster we use Apache Kafka [3]. One key function of the data bus is to provide buffering, which is critical when the data-ingestion rate exceeds the data-consumption rate. Kafka allows data producers to *publish* data to different *topics*. Thus, frames and log events emitted by the crawler enter the data pipeline through two different Kafka topics.

In the next stage of the data pipeline, clusters of indexers fetch data from Kafka. Frame indexers *subscribe* to the frame topic and store the incoming frames on the Elasticsearch [10] store. Once indexed on Elasticsearch, a frame becomes a searchable document. We refer to this general management paradigm as *state as documents*. Using the Elasticsearch query language, users, operators, or analytics applications can execute semantically-rich queries to find frames and frame features. Every attribute of every feature of indexed frames can be used as a query key, and so can the frame's metadata fields. This notion of applying search to manipulate operational data (system state) made visible, derived from the *state-as-documents* paradigm, is extremely powerful, as the analytics applications presented in §4 demonstrate.

We used Logstash [11] to implement our frame indexers. In particular, we relied on two Logstash plugins: Kafka *input plugin* and Elasticsearch *output plugin*, while adding a new *filter plugin* to process crawler frames.

Backend extensibility. The *annotators* (see Figure 2) are also an important element of our data pipeline. An annotator's key goal is to read frames of interest to create and index a different type of document. This is done to support certain analytics applications that might need to search for these special, curated documents.

We distinguish between two categories of annotators: privileged and regular. Privileged annotators typically originate from the cloud provider to support analytics applications that have general applicability, e.g., to detect vulnerabilities in all cloud-user images (see §4.1). These annotators fetch frames directly from Kafka, like the indexers. Regular annotators, on the other hand, can be provided by users for creating document types not supported by other annotators. User-provided regular annotators are deployed on a sandbox environment provided by our serverless cloud infrastructure, and they read frames from Elasticsearch, not Kafka.

Log indexing. Log indexers subscribe to the log topic and store incoming log events on Elasticsearch. Like the frame indexers, we used Logstash to implement them. Unlike frames, which have a well-defined structure, cloud application logs can have any format, since many of them are application-specific. This implies that performing semantically-rich Elasticsearch queries on logs (using attributes in log records as keys) might not be possible, unless the log indexers know what log format to expect. Before emitting logs, the crawler tries to apply the Logstash JSON filter plugin; thus, if logs coming from cloud applications are in the JSON format, our log indexers will guarantee that log records can be queried by log attributes, whatever they might be. Note that free-text queries are still possible, even if logs are emitted in an unknown format.

4 Analytics applications over *OpVis*

Our operational visibility pipeline has been running in our production cloud for over two years. It has been the foundation for analytics applications providing security-oriented insights to our clients. We highlight four such applications in this Section, focusing specifically on Docker images and containers, although the analytics presented are independent of the target runtime.

4.1 Vulnerability analyzer (VA)

VA discovers package vulnerabilities in users' Docker images and containers in our public cloud, and points them to known fixes in terms of relevant distribution-specific upgrades. Once a frame enters the *OpVis* pipeline, the VA annotator extracts the package list from it and compares it against publicly-available vulnerability databases (e.g. NVD) to report vulnerable packages with their corresponding CVE ids. We currently target Ubuntu security notices, while support for other distributions is in the process. VA has been integrated as part of a DevOps pipeline where, based upon

user-specified policies, images tagged as vulnerable by VA cannot be deployed as containers.

Recently, we have also added support for application-runtime-specific packages such as Ruby gems and Python pip packages. An interesting security dimension this feature addresses is defense against typo-squatting attacks on application libraries [27]. In this case, malicious packages similar in names to legitimate ones make their way into a user's system when the user inadvertently makes a typo while executing an installation command, for example, `pip install reqeusts` instead of `requests`. By comparing installed packages against permutations of white-listed ones, such malicious packages can be detected and protected against.

4.2 Security configuration (SecConfig)

Software misconfiguration has been a major source of availability, performance, and security problems. For an application developer using third-party components shipped as standard Docker images, it is non-trivial to ensure optimal values for various configuration settings spread across different components. To aid in this process, SecConfig scans applications and system configuration settings to test for compliance and best practices adherence from a security perspective. SecConfig gives container developers a view into their runtimes' security properties, and provides guidance as to how they should be improved to meet best practices in accordance with certain industry guidelines like HIPAA, OWASP, PCI, and CIS [19], taking into account internal deployment standards. For details, we direct the reader to our SecConfig paper [13].

4.3 Drift analysis

One key tenet of DevOps automation is enforcing container immutability. Once a container goes into production, the expectation is that it would never be accessed manually (login or ssh) and thus its contents or behavior should be the same thereafter. However, it has been observed that systems inevitably "drift" [12], i.e., deployed containers change over time and show unexpected behavior; changes can be either persistent or reside in memory.

With our *OpVis* pipeline, we can detect such drifts by tracking the evolution of all monitored systems over time. Specifically, by computing the differences of collected container frames over time, we can discover which systems violate the immutability principle, and then investigate the deviation origin by uncovering potential causes ranging from package-level changes to even fine-grain ones that manifest as altered application configuration settings.

For instance, an analysis of our production cloud over a 2-week period showed that almost 5% of the containers exhibited unexpected deviations from their images in terms of detected vulnerabilities and compliance violations. Although changes in the vulnerability database itself could explain a few cases, our analysis uncovered that "in-place" updates to the containers, both benign and undesirable, were taking place via ssh, docker exec, automated software updates, and software reconfiguration via web front-ends.

4.4 Malware analysis

For containers, the scope of vulnerability exposure by rootkits is smaller than that for VMs or physical servers. For example, certain kernel-mode rootkits are developed as loadable kernel modules in Linux, but application containers do not generally have privileges to load kernel modules. Therefore, for malware detection we focused on detecting file-based malware in container images and instances. To that end, we maintain a repository of known malwares and their corresponding offending file paths by pulling their definitions from available open-source targets[20][4]. Our malware analysis application then checks the file paths crawled from images and containers against this repository to identify potential malwares.

5 Experiences

In this section we discuss a few experiences and lessons we learned while running *OpVis* in production for over two years.

Choosing the right APIs for monitoring. A critical monitoring aspect is choosing the stack layer to get data from. Different stack layers may pose different challenges, one of which is API stability.

One of the crawler functions is to collect data from containers. Doing so can be achieved at different layers and with different methods. For example, to collect the CPU usage of a container, we can use either Docker APIs or *cgroups* at the host level. We observed that the Docker APIs have been changing rapidly, compared to the *cgroup* APIs, which are provided by the Linux kernel. Using more stable monitoring APIs will require fewer changes to *OpVis* as a result of updates to the underlying systems.

Burst and sampling bias. The number of crawler processes created by the main thread can be one or more and is provided as an option. When multiple threads are created, their cumulative activity may result in spikes for CPU utilization. By staggering the activity of the threads, the overall consumption can be amortized leading to an average lowering of CPU activity.

Watching out for starvation. To monitor the crawler's behavior with respect to log collection, one of the things we did was to deploy on each cloud host a test container emitting log events at a low frequency: 2 log events per minute. We created a dashboard to verify whether and when the logs from our test containers were being indexed on the data store. We noticed that, occasionally, some containers generating logs at an extremely high frequency were deployed to a few hosts and, when that happened, our low-frequency test logs from those hosts were significantly delayed.

Our investigation revealed that the root cause was not in the data service; the indexers were working properly and there was no backlog on the data bus. To corroborate our suspicion of starvation caused by the crawler behavior, we experimented with throttling high-frequency logs, which indeed mitigated the problem.

Operational visibility systems need global, distributed admission-control policies in place to allow fair and timely visibility into all systems across all monitored cloud runtimes.

6 Contribution

We made both internal (production cloud) and external (GitHub) contributions. *OpVis*' crawler, with container inspection and VM introspection capabilities, has been open-sourced [7] and has thus far seen contributions from 19 developers internationally.

The *OpVis* backend is closed-sourced and offers analytics services to customers of our container clouds. The full *OpVis* pipeline has been in production for over two years. Our VM monitoring capabilities were previously part of our OpenStack cloud that supported over 1000 OS versions. Multiple instances of the crawler alone have been deployed as data collectors for internal departments.

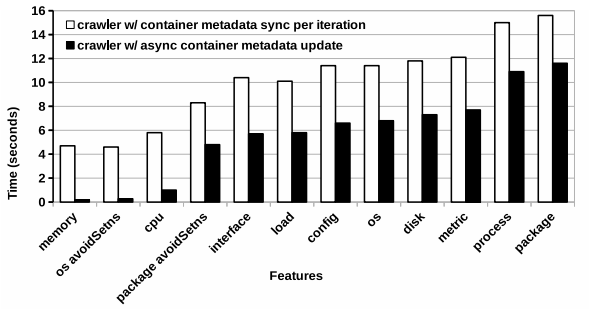

Figure 5. Time to crawl different features for 200 containers.

7 Evaluation

To evaluate *OpVis*' efficiency and scalability we first assess the monitoring frequency achievable by the out-of-band crawler. Then, we contrast the performance of out-of-band and in-band monitoring, measure the frames' space overhead, and present numbers for production-scale log-event streaming. The experiments focus on container clouds; the corresponding benefits for VM clouds have been demonstrated in our previous work [25]. In the experiments, the hosts have 16 Intel Xeon E5520 (2.27GHz) cores and 64 GB of RAM, and run CentOS 7, Linux kernel 3.10.0-514.6.1.el7.x86_64, and Docker version 1.13.1. The guest containers are created from the Apache httpd-2.4.25 DockerHub image.

7.1 Monitoring latency and frequency

We measured the maximum frequency with which the crawler can extract state from containers. Figure 5 shows the time it takes to extract different runtime-state elements (features) from 200 Web server containers. Shown are two sets of bars for each feature. The left bar represents the case where the crawler synchronizes with the Docker daemon on every monitoring iteration to get metadata for each container, whereas the right bar shows the optimization where the crawler caches the container metadata after the first iteration and subscribes to Docker events to update its cache asynchronously for container creation or deletion. The optimization yields an average improvement of 4.4s to crawl 200 containers.

Another point to note is the crawl-time reduction when namespace jumps are avoided. Figure 5 shows 11.6s vs 4.8s to crawl packages with and without namespace jumping, respectively. Finally, although we show the latencies for extracting individual features, for feature combinations one can just add the individual times.

We observed that the crawler can easily support over 10 Hz of monitoring frequency per container, and can go beyond 100 Hz for certain feature types. This observation considers a single crawler process consuming a single CPU core, although for many feature plugins only 70% of a core is consumed, with time spent waiting for either the disk while reading containers' *rootfs*, or the kernel while reading cgroups stats and/or during namespace jumping.

7.2 Performance impact

Next, we measure the *OpVis* agentless crawler performance impact and compare it with that of agent-based in-guest monitoring. We run 200 Web server containers on the host, and configured the crawler to extract CPU, memory, and package data from each container. The first two plugins provide metrics data, whereas the third allows periodic vulnerability analysis (see Section 4.1).

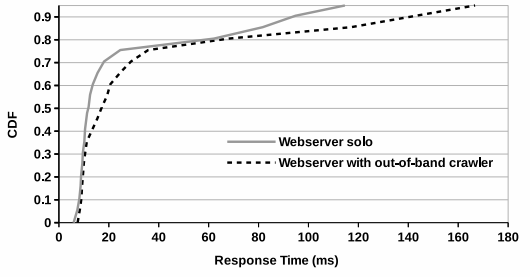

Figure 6. CDF of webserver response times.

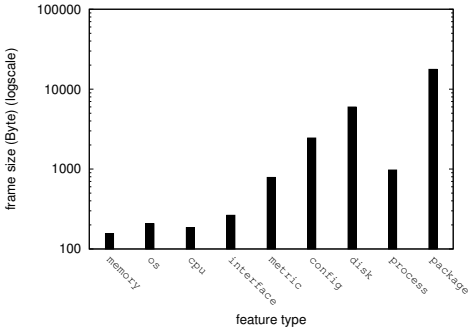

Figure 7. Size accounting for each crawled feature.

Each container was given equal CPU shares and 256 MB of memory. To minimize interference, we pinned the containers to 10 cores and the Docker daemon and its helper processes to the remaining 6 cores. Further, various Web server and kernel parameters were tuned appropriately to enable high throughput operations. The workload consisted of random 64-byte static files, requests for which were made to each container from another host via httperf [17] workload generators. We carefully selected the file sizes to prevent network saturation, and recorded throughput and response time after a warm-up phase to ensure all data was served from memory and thus impose maximum stress on the system.

The maximum sustainable throughput of the Web server aggregated across all 200 containers was 28000 requests/s (140 requests/s per container), with an average response time of 30ms per request. With continuous out-of-band crawling, we observed no impact on the throughput. However, the average response time degraded by 50% to 45ms. Interestingly, not all containers were affected by the increase in response time, as can be seen in Figure 6 which plots the CDF of response times for the 200 containers with and without the *OpVis* crawler. The crawl latency itself increases from 6s to 8s.

To mimic an agent-based monitoring methodology, we next ran the crawler process inside each container, configured to crawl every 8s as per the above out-of-band crawling experiment. With such in-guest monitoring, the aggregate sustainable throughput decreased 14%, and response times increased 65%. The in-guest monitor process competes for resources with the user workload, whereas the out-of-band crawler has its own dedicated core.

For completeness, we evaluated the out-of-band crawler restricting it to use only the cores that were running the Web server containers. In this case we still saw no impact on throughput, but with a 75% response-time increase (up from 50% with a dedicated core). Alternatively, baseline response times could be achieved but with a 7% throughput decrease (still better than a 14% degradation incurred by in-guest monitoring, which goes to 21% to meet the baseline response times). *Note that being able to use a separate core is*

in fact desirable and indeed a benefit of OpVis*' decoupled execution–monitoring framework, which enables offloading monitoring tasks outside the critical workflow path, thus minimizing interference.*

7.3 Space and network overhead

Disaggregated delivery of analytics functions is a key feature of *OpVis*, enabled by the separation between *data collection* (crawler) and *data curation* (backend annotators). Consequently, there is the need to transfer data between crawlers and annotators, which are typically separated by low-latency, high-bandwidth local-area network connections. We measured the overhead of transferring and storing crawled data. Figure 7 shows the size of crawled data for each individual feature type from a single Web server container. During this experiment we used a popular crawler output format (JSON); it is important to note that the space overhead differs for different emitter formats. Such overhead can be reduced by using data compression, which has its own performance implications by adding data-curation latency during decompression.

7.4 Performance of annotators

From the cloud users' perspective, it is critical that violations of security policies on their containers be discovered quickly. In certain cases it depends on external factors, e.g., how quickly a newly-discovered package vulnerability is added to standard CVEs. In the *OpVis* context, we measure the time taken by each annotator to produce a verdict. We measured the average latencies of 3 *OpVis* security-related annotators. *Remote Login Check* checks all account passwords inside containers for weaknesses. For containers with weak passwords the security report is produced quickly (within a second), and for those with strong passwords it can take up to 15 seconds. *Compliance Check* validates 21 standard security rules within 7 seconds. Finally, *Vulnerability Analyzer* checks packages against any known vulnerabilities within 1 second.

7.5 Log streaming in production

We now present data on the crawler's log-streaming behavior observed during 1 month, while it was exercised by external users and internal core services of our production cloud. Over this period, the crawler was tracking several hundred containers per host and thousands of containers, collecting approximately 250, 000 log events per minute per host on average. Figure 8 shows the data for one of our cloud hosts. The top plot shows the rate of new log files created per minute, exhibiting the level of dynamism and live activity in the cloud, with many new log files discovered every minute as instances come and go. It also shows a trend of increasing adoption and scale at the macro level as the month progresses. The bottom plot shows the number of log events processed per second.

8 Conclusion

In this paper we presented *OpVis*, our unified monitoring and analytics framework to achieve operational visibility across the cloud. We described the various techniques employed to enable agentless, non-intrusive extraction of volatile and persistent state from VM and container guests. We emphasized the extensible nature of our framework, enabling custom data collection as well as analysis. We described four of the analytics applications we built atop the *OpVis* pipeline, which have been in our public cloud for over 2 years. We experimentally highlighted *OpVis*' high monitoring efficiency and low impact on target guests.

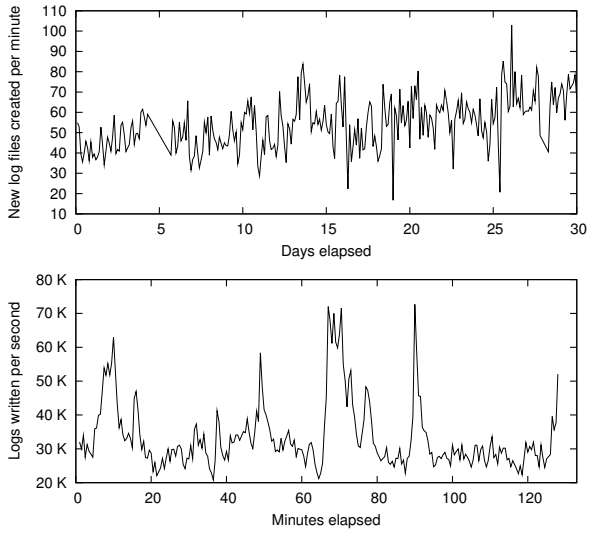

Figure 8. New log files per minute (top) and logs processed per second (bottom).

References

[1] Amazon. Summary of the october 22,2012 aws service event in the us-east region. https://aws.amazon.com/message/680342/, 2012.
[2] Anchore. Open source tools for container security and compliance. https://anchore.com/, 2017.
[3] Apache Kafka. https://kafka.apache.org, 2017.
[4] Michael Boelen and John Horne. The rootkit hunter project. http://rkhunter.sourceforge.net/, 2017.
[5] Google cadvisor. Analyzes resource usage and performance characteristics of running containers. https://github.com/google/cadvisor, 2017.
[6] Clair. Automatic container vulnerability and security scanning for appc and docker. https://coreos.com/clair/docs/latest/, 2017.
[7] cloudviz. agentless-system-crawler: A tool to crawl systems like crawlers for the web. https://github.com/cloudviz/agentless-system-crawler, 2017.
[8] Collectd. The system statistics collection daemon. https://collectd.org/, 2017.
[9] Datadog. Modern monitoring and analytics. https://www.datadoghq.com/, 2017.
[10] Elastic. Elasticsearch. https://www.elastic.co/products/elasticsearch, 2017.
[11] Elastic. Logstash. https://www.elastic.co/products/logstash, 2017.
[12] Byungchul Tak et al. Understanding security implications of using containers in the cloud. In *USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.
[13] Salman Baset et al. Usable declarative configuration specification and validation for applications, system, and cloud. In *Industrial Track of the 18th International Middleware Conference (Middleware'17)*, 2017.
[14] Tenable Flawcheck. Expanding vulnerability management to container security. https://www.flawcheck.com/, 2017.
[15] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Network and Distributed Systems Security Symposium*, 2003.
[16] VMWare Inc. Vmware vmsafe security technology. http://www.vmware.com/company/news/releases/vmsafe_vmworld.html, 2017.
[17] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3), 1998.
[18] Fábio Oliveira, Sahil Suneja, Shripad Nadgowda, Priya Nagpurkar, and Canturk Isci. A cloud-native monitoring and analytics framework. Technical Report RC25669, IBM Research, 2017.
[19] OpenSCAP. Security compliance. https://scap.nist.gov/validation/, 2017.
[20] OSSEC. Open source hids security. https://ossec.github.io/, 2017.
[21] Prometheus. Monitoring system & time series database. https://prometheus.io/, 2017.
[22] Aqua Security. Container security - docker, kubernetes, openshift, mesos. https://www.aquasec.com/, 2017.
[23] Tenable Network Security. Nessus vulnerability scanner. http://www.tenable.com/products/nessus-vulnerability-scanner, 2017.
[24] Sensu. Full-stack monitoring for today's business. https://sensuapp.org/, 2017.
[25] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal de Lara, and Todd Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, 2014.
[26] Sysdig. Docker monitoring, kubernetes monitoring & more. https://sysdig.com/, 2017.
[27] Nikolai Tschacher. Typosquatting programming language package managers. http://incolumitas.com/2016/06/08/typosquatting-package-managers/, 2016.

[28] Twistlock. Docker security & container security platform. https://www.twistlock.com/, 2017.
[29] VMware. Vmci overview. http://pubs.vmware.com/vmci-sdk/, 2017.
[30] VMware. Vmware tools. http://kb.vmware.com/kb/340, 2017.
[31] VMware. vshield endpoint. http://www.vmware.com/products/vsphere/features-endpoint, 2017.