ImageJockey: A Framework for Container Performance Engineering

Takeshi Yoshimura, Rina Nakazawa, Tatsuhiro Chiba
IBM Research - Tokyo
19-21 Nihonbashi Hakozaki, Chuo-ku, Tokyo, Japan
{*tyos, rina, chiba*}*@jp.ibm.com*

***Abstract*—Containerized applications have become widely used in modern software development due to their high flexibility and lightweight deployment. Currently, there exists a large set of publicly available container images with many different operating systems (OSs) and versions. As a result, developers typically select images carefully on the basis of memory footprint versus flexibility with available libraries. However, information regarding performance is insufficient. We have verified that different OS-based images with different versions vary the performance of certain applications running in them. Additionally, minor updates in container images, without changing its versions, also affect application performance. Therefore, to understand application performance, it is important to determine impact on performance in different container images along with the continuous monitoring of the minor changes.**

Since existing performance test frameworks do not encompass the required features to analyze performance regressions in container images, in this paper, we introduce *ImageJockey*, an original test framework to continuously evaluate the performance of a broad range of container images. Our framework enables experiments of container benchmarks with periodic image builds, simple container orchestration, metrics collection, and result visualization. We demonstrate the usefulness of our framework through case studies that analyze the performance characteristics of sixteen container images and nine popular benchmarks. The experimental results show that there is a noticeable performance variation due to the deployed environments and characteristics of Alpine and JDK images.

***Keywords*-Container; Container image; Performance; Analysis; Benchmark; Framework; Test; Docker**

I. Introduction

Container images are crucial software components for modern cloud applications offering lightweight deployment and high reusability. Container images are distributed in public repositories such as Docker Hub, which contain over 100,000 images [1]. The public repositories provide a wide variety of images including Web frameworks, operating system (OS) tools, and programming language runtimes, which are essential to start and run Linux applications. Each image also provides many choices of image versions as tags. For example, a popular Linux image, Ubuntu, provides 223 versions of x86_64 images with 373 tags at the time of this writing.

Developers often reuse and enhance existing images with application logic and create new container images. Container images for OSs and runtimes contain software binaries to support broad usage and rich user-friendly features.

These additional features are useful for development purposes and many use cases but can be harmful to cloud deployment due to security concerns and excess memory overhead for rarely-used features. As a result, developers select container-oriented runtimes such as Alpine Linux as an application base, avoiding unnecessary binaries to initialize and execute Linux applications.

These runtime images have compatible interfaces but performance characteristics can be different due to their minimized binaries. Given this vast range of image selection, developers typically select images on the basis of memory footprint versus flexibility with available libraries. However, information regarding performance is insufficient because memory footprint does not cover all aspects of the complex characteristics of application performance. Therefore, this paper aims to provide a framework to help to answer the research question "What impact does container image type have on application performance?"

Existing performance test tools and frameworks [2] [3] [4] [5] help reduce developer effort required to configure and run complex experiments. ReBench [5] and airspeed velocity (asv) [2] are simple tools to enable continuous performance testing of generic benchmarks. CloudBench [3] and CloudPerf [4] support cloud-based deployments including distributed environments. However, they do not focus on container workloads, which poses the following new challenges for performance test frameworks.

First, different deployment models must be supported due to the isolation mechanisms inherent with containers. Applications based on a server-client architecture require network configurations for multiple containers on different physical/virtual hosts. Even for stand-alone applications, online and post-hoc analysis requires systematic metrics collection for containers. Developers often collect CPU, memory, and other resource metrics to understand performance characteristics, but metrics collection needs to connect to containers behind isolation mechanisms. Runtimes with just-in-time compilers like JDK also require special adaptation of agent tools to support Linux `perf` tools.

Second, public repositories continuously update container tags due to minor fixes for security and bugs but developers cannot retrieve the previous state of overwritten tags from remote repositories. For example, we observed two minor updates of a Python image tagged as '3.8' in February 2020. Thus, post-hoc analysis requires the preservation of past versions of benchmark and base images to precisely reproduce past experiments. Developers also need to test performance regression due to minor or major updates of container images.

Finally, many choices of container images and tags sometimes require enormous amounts of continuous experiments to cover all possible combinations of tags and benchmarks, which easily exceeds human comprehension. None of the existing frameworks provide the necessary tools to satisfy all of these requirements.

In this paper, we propose *ImageJockey*, an original benchmark framework to reduce the developer engineering effort required to evaluate the performance of container images. Our framework enables us to run a diverse range of popular benchmarks for programming language runtimes, databases, Web servers/applications, and OSs. It supports single-container experiments such as DaCapo [6] and UnixBench, and multi-container ones such as YCSB [7] and DayTrader. Our framework uses a periodic image builder, benchmark driver, metrics collection, and visualization tools to enable to analyze a broad range of container workloads. The periodic image builder enables us to track and preserve the latest state of a tagged container image, which is essential for advanced post-hoc analysis with Linux `perf`. The benchmark driver contains Python class libraries to easily define and run new benchmark experiments including distributed ones with rich metrics collection for containers. Visualization tools enable developers to continuously monitor daily changes of enormous amounts of continuous benchmark results.

To demonstrate the usefulness of our framework, we analyze sixteen container images and nine benchmarks. Our analysis shows the following implications for container application development and testing.

- The performance variation of selected container images is non-negligible. For example, `nginx` showed three times better throughput than `httpd` under ApacheBench. Our experimental results indicate that performance testing of container images is essential for the development of performance-sensitive applications. Note that the selection of container images is often based on not only performance but also software security, license, and other practical factors. Our framework enables engineers to select better container images in terms of performance.
- A benchmark from UnixBench showed that Alpine processed 50% more operations per second than CentOS. Alpine also had lower CPU usage than CentOS. This indicates that "fat" images potentially change performance characteristics compared with "slim" images not only due to reduced container image size but also due to the unique system libraries of the Alpine image.

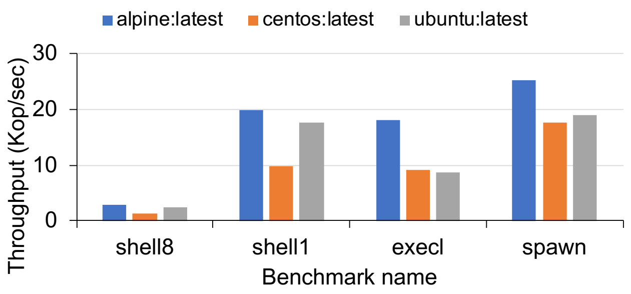


Figure 1. **UnixBench workloads impacted by OS image**. This figure shows UnixBench workloads that had more than a 50% throughput variation caused by changing the OS image.

- The latest images for IBM JDK and AdoptOpenJDK showed different performance characteristics regarding synchronization primitives, e.g., object monitors in Java. They showed different CPU usage patterns despite running the same benchmark on the same cloud environment. This result indicates that many container runtimes using Java such as Open Liberty, Cassandra, and Elasticsearch potentially show similar performance characteristics on concurrent workloads.

The unique contributions of this paper are threefold: 1) design and implementation of our continuous performance testing framework for container workloads with a periodic image builder, benchmark driver, metrics collection, and visualization tool, 2) demonstration of our framework with case studies of nine popular benchmarks and sixteen container images, and 3) implications of container development/testing from our performance analysis of containers.

II. Motivation

In this section, we briefly show the performance variation due to 1) selected container images and 2) experimental environments. The experimental results of varying container images highlight the necessity of performance testing and careful selection of base container images and tags. The experiments in which the environment is varied show that even carefully selected images can have different performances due to external events and continuous testing enables us to quantitatively understand these effects.

A. Performance characteristics of runtime containers

First, we evaluate the performance characteristics of runtime container images, which may affect the performance characteristics of application containers. We test the performance of UnixBench and DaCapo, as application containers in this section. For UnixBench, we used Alpine, CentOS, and Ubuntu images as the base images. UnixBench consists of system-level benchmarks to measure the performance of basic system operations such as system calls, shell scripts, and file operations. DaCapo is a benchmark collection used

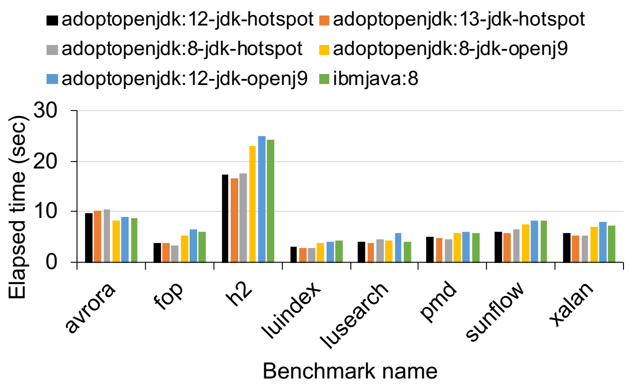


Figure 2. **DaCapo performance with different JDK images**.

to analyze and optimize Java applications, runtimes, and compilers. We ran experiments on an `m4.xlarge` instance on AWS EC2 with four virtual CPUs, 16 GB RAM, and a 100 GB attached volume. Docker containers for this experiment ran on the same Linux kernel version 4.15. Hence system-level benchmarks like UnixBench should show similar performance characteristics for different base OS images. However, as shown in Figure 1, the Alpine image showed a 50% to 100% higher throughput for a number of the workloads in UnixBench compared with Ubuntu and CentOS. Other benchmarks such as fsbuffer and dhry2reg did not show significant differences.

We also evaluated DaCapo with different JDK images and tags on the same node as the UnixBench experiment. Figure 2 shows the performance summary of the DaCapo images we built with multiple versions of AdoptOpenJDK and IBM JDK 8. Dacapo avrora showed OpenJ9-based runtimes (adoptopenjdk:8-jdk-openj9, adoptopenjdk:12-jdk-openj9, and ibmjava:8) performed well, while in other benchmarks, HotSpot-based runtimes (adoptopenjdk:8-jdk-hotspot, adoptopenjdk:12-jdk-hotspot, adoptopenjdk:13-jdk-hotspot) showed better results. DaCapo avrora is a set of simulation and analysis tools in a framework for AVR microcontrollers, which exhibit a great deal of fine-grained concurrency. HotSpot-based AdoptOpenJDK 8 showed better performance than later versions with fop, luindex, pmd, and xalan tests.

These two experimental results show that the selection of base images and tags have a large effect on the performance of application containers. Developers sometimes need to apply upgrades to the applications and OS images they depend on due to security and feature enhancements, however, the latest image does not always provide the best performance.

B. Performance portability of containers

We tested the performance variation of DaCapo avrora and UnixBench shell1 from February 14 to March 25, 2020 in Figure 3. Shell1 consists of multiple iterations of a simple sort program written in a Unix shell script, which launches multiple processes with inter-process communication via a Unix pipe. Note that we reconfigured the storage of the experimental node on March 13th. As shown in the figure, DaCapo avrora showed temporary performance degradation due to the reconfiguration. In contrast, UnixBench shell1 did not show significant performance changes in the event. Our experiments indicate that the functional portability of container images is not equivalent to performance portability, and containers do not eliminate the need for performance engineering. The performance of container images can vary depending on external factors such as deployed environments and device configuration. Developers still need continuous performance evaluation to avoid the performance regression of container applications.

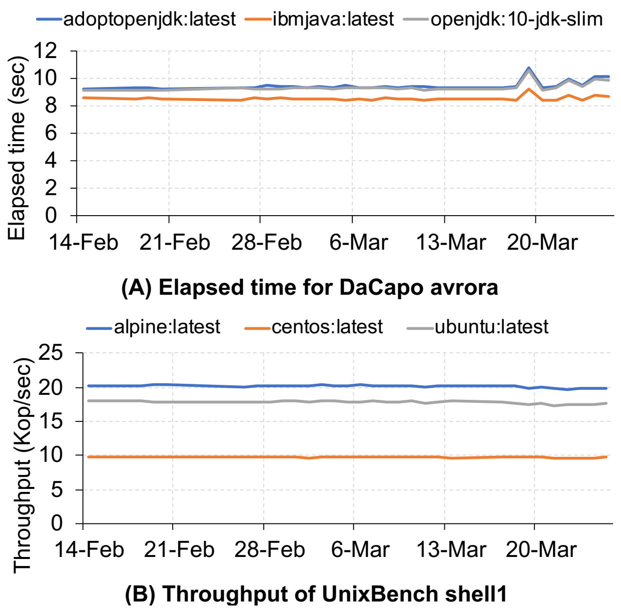


Figure 3. **Daily performance variation of DaCapo avrora (A) and UnixBench fsbuffer (B) with different images**.

C. Summary

In summary, our experiments in this section show the following problems of container performance engineering.

- Latest image does not always provide the best performance (Section II-A).
- Functional portability of container image is not equivalent to performance portability (Section II-B).

Containerization is regarded as a lightweight virtualization technology to show similar performance to bare-metal environments. However, the above problems indicate that containerization does not solve all performance problems and we still need performance testing of containerized workloads. In this work, we aim to reduce engineers' efforts to continuously evaluate their container applications.

Category	Image name	Benchmark
Java	adoptopenjdk, openjdk ibmjava	DaCapo [6], SPECjbb2015 [8]
Python	python	PyPerformance [9]
DB	redis, memcached, cassandra, elasticsearch	YCSB [7], Rally [10]
Web	httpd, nginx, tomcat, open-liberty	ApacheBench [11] DayTrader [12]
OS	alpine, ubuntu, centos	UnixBench [13]
Misc.	hello-world	Time for start up

Figure 4. **Example benchmark images** We use popular benchmarks to evaluate the performance of container images in Docker Hub. For hello-world, we measured the elapsed time to start a container and display a "hello world" message on the console. We also examined multiple versions of images. For example, we used three different versions of Python 3.

III. Benchmark images

This section describes workloads we examined to understand container workload performance implications. The example workloads are listed by category in Figure 4. Application developers often reuse container images for language runtimes such as Java and Python. Many modern microservices require database (DB) container images such as Redis and Memcached. Service providers often use Web (or Web application) container images `nginx` and `httpd` as front-end servers. We also add OS images such as Alpine and Ubuntu, which are commonly used as base images.

Each benchmark in Figure 4 represents typical workloads for evaluated programs. They are distributed in different forms such as Java archive (`.jar`) files, Python packages, and C source code. Thus, running a benchmark in a container requires building a workload image with benchmark files and existing container images. For example, we enhanced Java images by copying `.jar` files to compose the DaCapo and the SPECjbb2015 container images. The PyPerformance image is built by installing a Python package manager and the pyperformance package to the Python image. The YCSB client uses an AdoptOpenJDK image with pre-built `.jar` files from official releases. DayTrader and UnixBench required additional Docker build stages for building source code and configuration. The Rally image requires to contain more than a gigabyte of test data to prevent downloads during each benchmark run.

IV. Framework Design and Implementation

In this section, we describe the concept and overall design of our continuous containerized workload evaluation framework. Then we explain the framework internals in detail.

A. *Concept and overall design of framework*

As discussed in Section II, our fundamental motivation is to validate containerized workload performance on cloud environments while tracking changes of a container image and cloud infrastructure in real-time. We wanted to enable application developers and cloud users to identify performance regression quickly and visualize detailed metrics so they could easily understand the root cause. Those motivations share a number of similarities with CI/CD tools in terms of periodical container image evaluation because those tools are also continuously running test suites to verify the correctness of code or its functionality. However, unlike CI/CD, our focus is on the performance of production or hybrid cloud environments rather than simply a test environment. Unfortunately, there is no framework to satisfy our motivations, so we proposed and built a framework to support continuous performance evaluation for containers.

Figure 5 shows the overall design and architecture we proposed in this paper. Our framework consists of a periodic image builder, metrics collection, benchmark driver, and visualization dashboard to track metrics and performance variation. We define an image update event as the trigger to initiate a full evaluation on our framework because this event often represents an entry point to perform DevOps in a cloud and is a natural point to integrate performance evaluation with the CI/CD cycle. To simplify the evaluation cycle, the framework checks for image updates periodically. Existing monitoring systems for containers such as DataDog [14] and Prometheus [15] enables a similar metrics collection to ours. However, our benchmark driver additionally provides easy orchestration among benchmark containers and metrics collection. The following sections explain each component in detail.

B. *Periodic image builder*

Our periodic build system reuses Docker tags and an external container repository to store a series of image snapshots for post-hoc analysis. We generate a unique ID for each benchmark day and append it to the tags of every container image. The tagged container images are pushed to a repository and used when a user needs to reproduce an experiment on a specific day. The Docker registry handles the deduplication of identical disk images with different tags to avoid consuming excessive disk space for large images such as Rally. We assume a container image update does not change greatly from the previous update.

For the example workloads, we re-build and track workload images every day using the Linux CRON daemon. Rebuilding images takes less time than the first build because of the Docker build cache system. We do not update benchmark binaries. However, base container images occasionally need to be updated for security and bug fixes. Currently, we do not observe performance bottlenecks with the image repositories but we may need to adopt efficient architectures to enhance container image repositories such as Bolt [16] depending on the size of image updates.

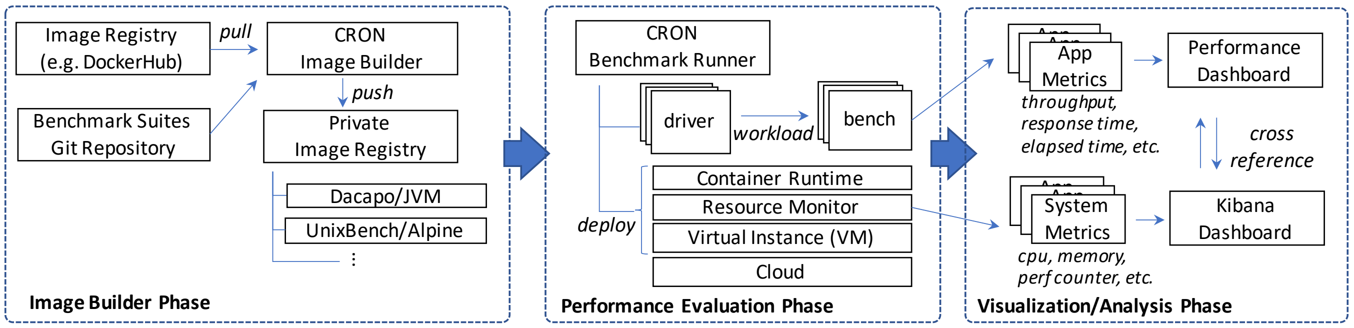


Figure 5. **Framework overview**

C. Metrics collection container

Our framework provides a special container image for metrics collection. The metrics collection container uses Docker APIs to retrieve performance metrics and provide output in JSON format. The Docker API interface is often exposed as a special file such as `/var/run/docker.sock` in Linux. The collected metrics include commonly used information ranging from a container's CPU, memory, disk, and network usage to system-level metrics such as the number of page faults.

Users can also explicitly launch the collector container for a specific monitored container by placing the image ID in an environmental variable. The collector container transfers the metrics to standard output or external loggers such as Fluentd. Currently, our framework places the collector container on the same host as the evaluated container, but our collector image also supports remote hosts if the Docker API interface of the evaluated container is exposed.

Our framework also contains special wrapper tools for the Linux `perf` recorder and reporter for containers. Docker enables us to capture Linux `perf` metrics specifying the `perf cgroups` ID for a Docker container. The reporter tool mounts a container image with the overlay filesystem (as performed by Docker) to enable Linux `perf` to traverse the binaries and debug information inside a container image.

To display Java-level code information, developers need to explicitly use an agent library for Java processes inside a container and obtain the mapping between symbol addresses and JITed methods. Our reporter tool automatically copies the mapping information for JITed code to `/tmp/perf-{PID}.map`, from which the Linux `perf` reporter looks up unresolved symbols. The `PID` number must be a host-level process ID, so our recorder tool automatically collects the mapping of container-level PIDs to host-level by querying `PID cgroups`.

Note that workload images need to contain debug information to obtain code-level metrics. From our observation, most of the container repositories in Docker Hub also provide such debug-friendly images as well as "slim" images. For example, JDK images are used for debugging JVM runtimes rather than JRE images. Benchmarks that require source builds like UnixBench need to modify build

```
"mynode": {
  "CPE_FLUENTD_ADDR": "192.168.13.2:24224",
  "CPE_REG_HOST": "www.repo.url",
  "CPE_DOCKER_HOSTS": {
     "localhost": "unix:///var/run/docker.sock",
     "192.168.13.3": "tcp://172.17.0.1:23750"
  }
}
```

Figure 6. **Example node configuration** This JSON string contains the host for Fluentd (CPE_FLUENTD_ADDR), Docker registry (CPE_REG_HOST), and benchmark runners (CPE_DOCKER_HOSTS). Developers can modify and pass this JSON string to the framework to manage the benchmark environment.

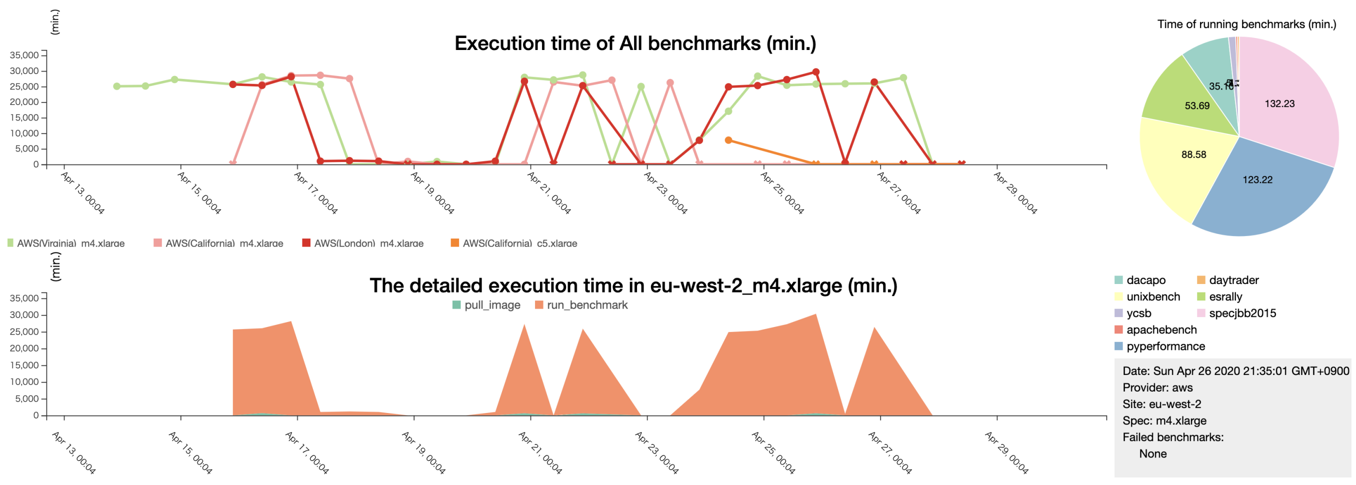
```
ports ={'9080/tcp': '9080', '9443/tcp': '9443'}
dt_img = conf.get_image('daytrader')
bd = BenchmarkDescription("daytrader", ...)
dt = DockerMon(ContainerRunner(dt_img, bd, ports=ports))
time.sleep(10)
bd2 = BenchmarkDescription('client_stdout', ...)
jm_img = conf.get_image('jmeter')
args =  ['-JHOST=' + dt.co.ip(), '-JTHREADS=4']
jmeter = ContainerRunner(jm_img, bd2, args, locallog=True)
dm = DockerMon(jmeter)
jmeter.join()
log = jmeter.log()
dm.join()
dt.stop()
... # parse log
```

Figure 7. **Example benchmark code for DayTrader** DayTrader requires server and client containers. The class library helps configure two containers through the ContainerRunner class and metrics collection with the DockerMon class. The BenchmarkDescription class enables users to add additional records for metrics collection to better comprehend voluminous logs.

options to include debug symbols. The technique introduced by Thalheim et al. [17] can enable to reduce the size of such "fat" container images, but a reduced image footprint potentially affects performance characteristics.

D. Benchmark driver container

We created another special container image as the benchmark driver to provide simple container orchestration. The benchmark driver allocates hosts for containers of benchmarks and the metrics collector. Users must pass a benchmark script for minimal, single-node experiments, but can also pass a list of available hosts and corresponding Docker API interfaces to run multi-node experiments. Our frame-



Date: Sun Apr 26 2020 21:35:01 GMT+0900
Provider: aws
Site: eu-west-2
Spec: m4.xlarge
Failed benchmarks:
None

Figure 8. **End-to-end view**. Developers can check the anomaly behavior of a benchmark harness and overview an overall trend before deep-dive analysis like that described in Section V.

work also enable users to specify a logger host through a JSON string in a configuration file (Figure 6).

For multi-node experiments, the container allocation policy uses a simple greedy algorithm. We randomly choose unused hosts from the given host list for each benchmark container. The benchmark fails if the benchmark script attempts to launch more containers than available hosts. The driver container records Docker commands to easily reproduce and customize past experiments in post-hoc analysis.

The benchmark scripts can reuse the asv library [2]. This library enables the easy collection of elapsed time and benchmark scores by parsing benchmark results from the standard output. Our framework also provides a helper class library to define inter-container communication for benchmark runs. Overlay networks are sometimes a bottleneck for certain workloads [18], so the framework also enables binding to host network ports. Figure 7 shows example Python code for DayTrader.

E. Visualization of performance and resource metrics

Our framework extends asv [2] to provide the visualization of performance metrics. We implemented the dashboard with D3.js [19] to display the execution time of all the benchmarks as end-to-end views (see the top of Figure 8). When a user clicks a data point on a line chart, this view shows the stacked area chart that represents the detailed performance of the clicked cloud environment. Users can also filter lines by one or multiple items such as providers and locations of containers. In the case that our framework is not able to collect the performance of all of the benchmarks, the view draws the data point of a line chart for a benchmark execution as a cross instead of a dot to represent the failure. Since these line charts include the execution time of all benchmarks, this view includes the time ratio of benchmarks using pie charts when the user clicks a data point or cross when viewing the second type of line chart.

By clicking a benchmark name on each view, the dashboard switches its view to the asv detailed view that visualizes the performance of the selected benchmark by line charts or bar charts. This view can filter performance data by image tag, version, and host node of a container. We visualize the detailed resource usage using Kibana [20] to assist with detailed performance analysis of computing resource usages and decision making for containerized applications. We add a link to this Kibana dashboard to the detailed performance view, which enables users to analyze performance bottlenecks with resource usage.

F. Summary and extension for new workloads

Developers can add new workloads by registering a Dockerfile to the periodic image builder (Section IV-B) and benchmark scripts to the benchmark driver (Section IV-D). Our framework provides a JSON file for benchmark registration. The periodic image builder and benchmark driver look up the file and automatically execute all the image builds and benchmark runs sequentially. Developers can reuse the metrics collection container and class libraries (Section IV-C and IV-D) to write benchmark scripts. After the benchmark finishes, the driver container collects benchmark results such as elapsed time and custom benchmark metrics. Finally, our framework visualizes the collected results as described in Section IV-E. We follow this development flow to build our example workloads in Section III.

V. Case Studies

In this section, we describe a performance analysis of container images with our framework. To show the effectiveness of our framework, our experiments focus on answering the following questions. 1) How did base container images modify the performance characteristics of benchmarks? 2)

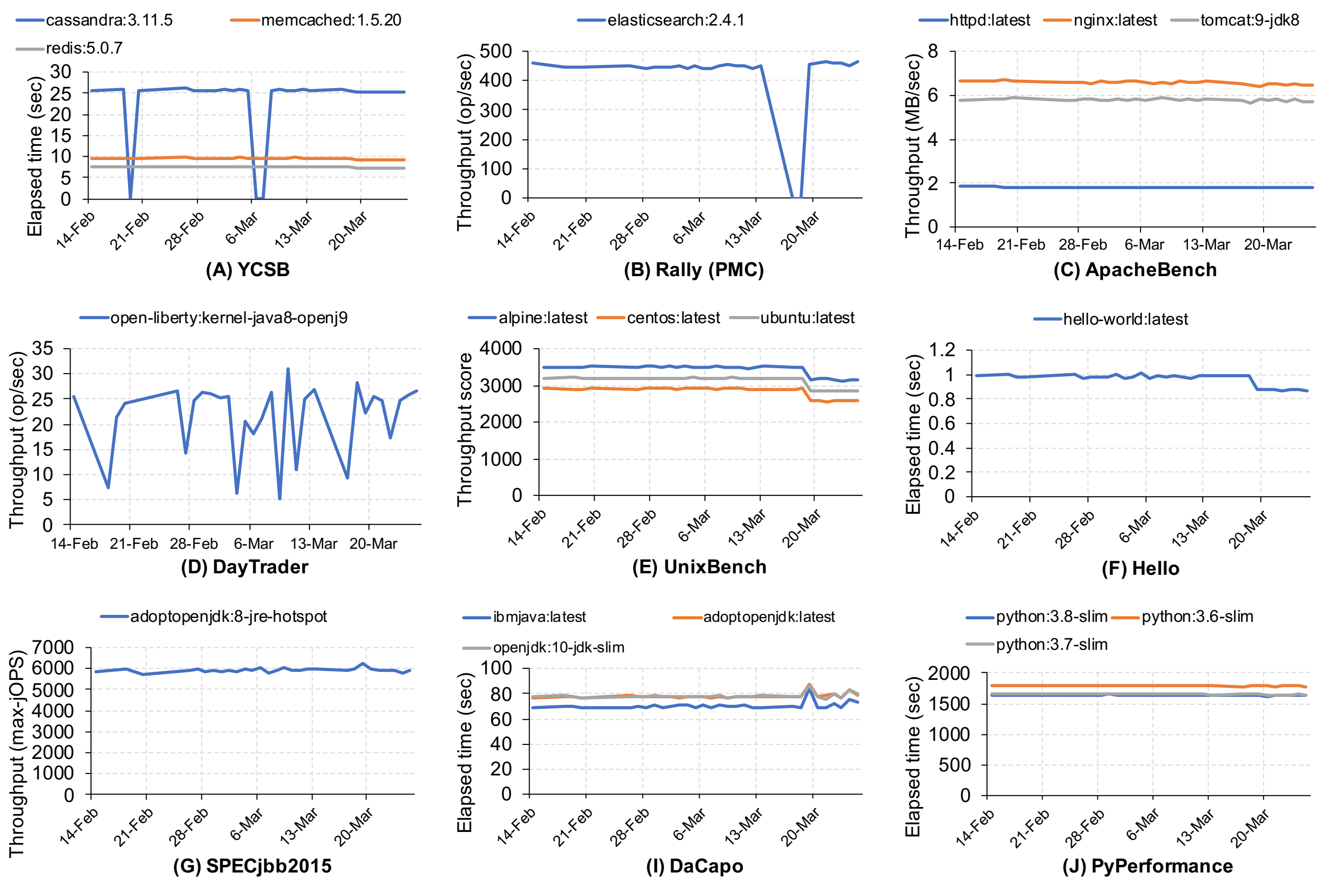


Figure 9. **Performance of benchmarks with different container images** Note that each total time and throughput are the total values of workloads defined in each benchmark. For example, the PyPerformance result is the total time of 47 Python workloads.

What was the reason for different performance characteristics between container images with the same benchmark?

We evaluated our collection of benchmark images in Figure 4 from February 14 to March 25, 2020. During the experimental period, storage of the node was reconfigured storage on March 13. Our experiments ran on a cloud instance on AWS EC2 with four virtual x86_64 CPUs, 16 GB RAM, and a 100 GB attached volume. The host OS of all nodes was Ubuntu 18.04LTS running Linux 4.15 and Docker engine 19.01. We ran all experiments in single-node mode to eliminate any effects of the network infrastructure.

A. Overall comparison of container images

For typical DB workloads, we evaluate Cassandra 3.11.5, Redis 5.0.7, Memcached 1.5.20, and Elasticsearch 2.4.1. We use YCSB for Cassandra, Redis, and Memcached and Rally PMC for Elasticsearch. Our experiments with YCSB measure the total elapsed times for load (100% insert), workload A (50%/50% read/write), workload B (95%/5% read/write), and workload C (100% read). Rally PMC performs textual analysis on open access technical articles. We used default benchmark configurations, for example, YCSB operates 1000 records as the workloads. The experimental results of the DB images are in Figure 9 (A) and (B). The disk reconfiguration caused a performance degradation with Rally resulting in both timeout and throughput reported as zero. However, the performance recovered to the same level as before the disk reconfiguration. In contrast, YCSB did not show performance degradation due to the reconfiguration. However, Cassandra experienced timeout failures several times. Redis and Memcached showed steady performance.

We next conduct performance benchmarks of Web-related images. Web benchmarks require client images for the stress testing of Web images. Our ApacheBench runs four client threads that retrieve a plain HTML file 10,000 times from unmodified server images of `httpd`, `nginx`, and Tomcat. We use the binary for ApacheBench within the `httpd` image. For DayTrader, we built a JMeter image and ran four client threads to access a Web application with Derby-backed DayTrader 8 built with Open Liberty. We selected the image of Open Liberty with the OpenJ9 version 8 runtime. The results of Web workloads are shown in Figure 9 (C) and (D). ApacheBench showed steady throughput for

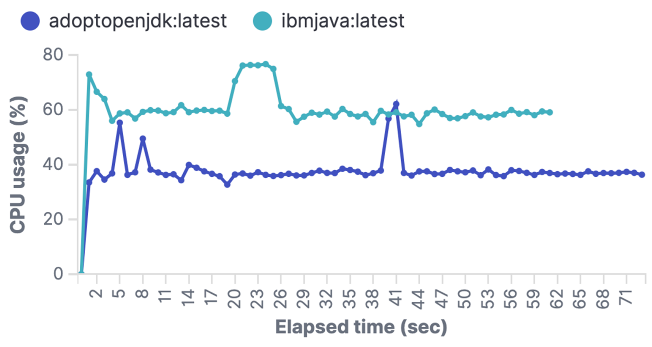
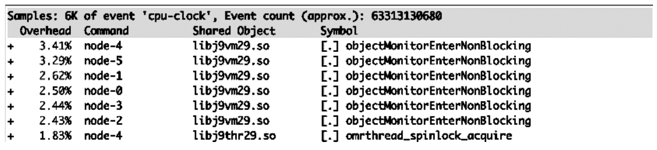
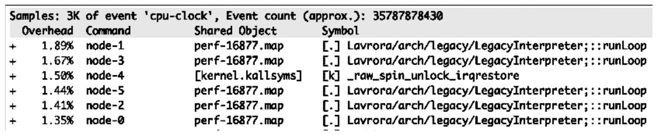


Figure 10. **CPU usage for DaCapo.avrora** ibmjava:latest finished faster than adoptopenjdk:latest



```
Samples: 6K of event 'cpu-clock', Event count (approx.): 63313130680
  Overhead  Command         Shared Object      Symbol
+    3.41%  node-4          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    3.29%  node-5          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    2.62%  node-1          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    2.50%  node-0          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    2.44%  node-3          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    2.43%  node-2          libj9vm29.so       [.] objectMonitorEnterNonBlocking
+    1.83%  node-4          libj9thr29.so      [.] omrthread_spinlock_acquire
```

Figure 11. **Hot functions in ibmjava:latest**



```
Samples: 3K of event 'cpu-clock', Event count (approx.): 35787878430
  Overhead  Command         Shared Object      Symbol
+    1.89%  node-1          perf-16877.map     [.] Lavrora/arch/legacy/LegacyInterpreter;::runLoop
+    1.67%  node-3          perf-16877.map     [.] Lavrora/arch/legacy/LegacyInterpreter;::runLoop
+    1.50%  node-4          [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
+    1.44%  node-5          perf-16877.map     [.] Lavrora/arch/legacy/LegacyInterpreter;::runLoop
+    1.41%  node-2          perf-16877.map     [.] Lavrora/arch/legacy/LegacyInterpreter;::runLoop
+    1.35%  node-0          perf-16877.map     [.] Lavrora/arch/legacy/LegacyInterpreter;::runLoop
```

Figure 12. **Hot functions in adoptopenjdk:latest**

every container image, while DayTrader showed the largest performance variation among our benchmarks. We observed that `nginx` showed more than 3x throughput improvement over `httpd` with the default configuration.

Our OS- and Docker-related workloads are UnixBench and hello-world. For UnixBench, we measured the total throughput score of 12 workloads with Alpine, Ubuntu, and CentOS. The score is calculated as the improved ratio from the baseline result. We also measured the elapsed time to display the message "hello world" with the hello-world image, which can approximate the start up and shutdown overhead of Docker containers. The results of UnixBench and hello-world are presented in Figure 9 (E) and (F). The storage reconfiguration largely affected the throughput of both benchmarks. In particular, filesystem-related workloads in UnixBench such as fstime, fsbuffer, and fsdisk showed large performance variation due to the reconfiguration.

Our final experiments tested different JDK images and multiple versions of Python with SPECjbb2015, DaCapo, and PyPerformance. The results in Figure 9 (G), (H), and (I) show that the Python image update improved the performance of PyPerformance. For DaCapo, IBM JDK had the best performance. The latest image of AdoptOpenJDK uses HotSpot-based runtimes as well as OpenJDK, while IBM JDK uses a different runtime.

B. *Comparison of container images*

1) IBM JDK vs. AdoptOpenJDK: We analyzed the performance characteristics of the latest container images for IBM JDK and AdoptOpenJDK with DaCapo. The selection of JVM often affects the performance of Java applications including common data analytics workloads like SQL-on-Hadoop [21].

Figure 10 shows the CPU usage for DaCapo avrora with IBM JDK and AdoptOpenJDK. We used default configurations for each JDK in this experiment, but the IBM JDK finished the benchmark iteration faster than AdoptOpenJDK due to high CPU utilization.

Hot functions identified with `perf` metrics in our framework reflected the concurrent properties of avrora (Figure 11 and 12). The IBM JDK used more CPU time to execute the fast path of entering object monitors, which is the primary synchronization primitive of Java applications. However, AdoptOpenJDK frequently executes other parts of avrora code except for kernel spinlocks instead of user-level synchronization. These two results mean that the low CPU utilization of AdotpOpenJDK came from relatively frequent CPU yields in object monitors. The implementation of the object monitor is full of heuristics, which, in our experience, makes performance prediction difficult without testing.

This example analysis indicates that the performance portability of container images is challenging especially for concurrent applications. Many container applications cannot avoid this because they are often built on type-safe language runtimes using garbage collection threads. We recommend that developers test the concurrent performance of their Java containers in pre-production environments.

2) Alpine vs. CentOS: Considering the results of the UnixBench shell1 benchmark in Figure 1, we analyzed the performance implications of the two base OS images, Alpine and CentOS. Despite the simplicity of the benchmark program, Alpine completed 50% more operations than CentOS. Alpine also had lower CPU usage than CentOS (Figure 13), and thus, we can obtain even better throughput by increasing the number of threads for this benchmark.

Figures 14 and 15 show the CPU usage differences between CentOS and Alpine. Alpine uses musl, a libc-compatible system library, but the hot functions for these experiments are slightly different from those of CentOS. CentOS spent more CPU time on string comparison and memory page management, which did not even show up as hot functions for Alpine. A small memory footprint can reduce the overhead of memory page management, but the difference in hot functions shows that musl shows different behavior than libc in CentOS for this workload.

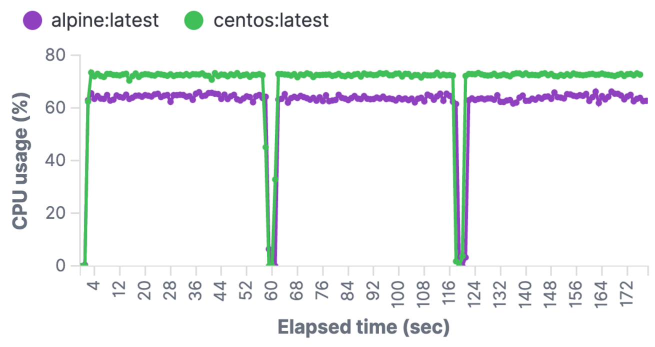
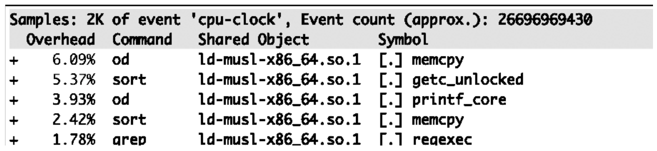


Figure 13. **CPU usage for UnixBench.shell1**



```
Samples: 2K of event 'cpu-clock', Event count (approx.): 26696969430
  Overhead  Command  Shared Object         Symbol
+    6.09%  od       ld-musl-x86_64.so.1  [.] memcpy
+    5.37%  sort     ld-musl-x86_64.so.1  [.] getc_unlocked
+    3.93%  od       ld-musl-x86_64.so.1  [.] printf_core
+    2.42%  sort     ld-musl-x86_64.so.1  [.] memcpy
+    1.78%  grep     ld-musl-x86_64.so.1  [.] regexec
```

Figure 14. **Hot functions in alpine:latest**

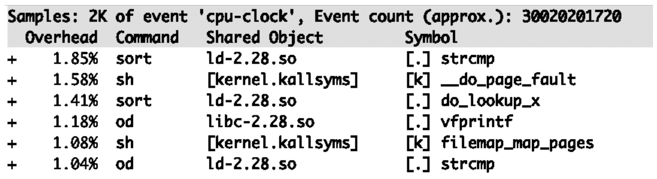
VI. Discussion

In this work, our scope is the relative performance of container images, and thus, please note that our analysis covers only limited aspects of the performance characteristics of general container workloads. We do not cover many interesting aspects to understand the absolute performance characteristics of containers. In this section, we describe other interesting performance aspects of container workloads and what users need to take into account to understand them.

For example, all of our experiments run on virtualized environments, which can cause unpredictable performance fluctuation due to noisy neighbors and virtualization overheads. The performance fluctuation should appear as unstable results across the entire experiments, but our daily experiments of many benchmarks mostly showed steady performance. A potential reason for this is that our experiments ran on a single node, and containers communicate with each other through local networks.

Runtime environments around containers including networks, virtualization, and the OS kernel can heavily affect the performance of container workloads. We ensured all the experiments run with the same software configuration and versions except for container images. To measure the performance effects of container runtimes, we recommend that users ensure using the same container image and running multiple days to determine performance variation due to external factors.

Container images inherently reuse application binaries, and thus, similar performance characteristics we found may occur on non-containerized workloads. In other words, users can leverage our framework for application development if they build Docker container images packaging their applications.



```
Samples: 2K of event 'cpu-clock', Event count (approx.): 30020201720
  Overhead  Command  Shared Object         Symbol
+    1.85%  sort     ld-2.28.so            [.] strcmp
+    1.58%  sh       [kernel.kallsyms]     [k] __do_page_fault
+    1.41%  sort     ld-2.28.so            [.] do_lookup_x
+    1.18%  od       libc-2.28.so          [.] vfprintf
+    1.08%  sh       [kernel.kallsyms]     [k] filemap_map_pages
+    1.04%  od       ld-2.28.so            [.] strcmp
```

Figure 15. **Hot functions in centos:latest**

The efficiency of metrics collection is not our focus, but optimizing metrics collection will enable a more precise understanding of the absolute performance characteristics of container workloads. We did not observe that metrics collection caused memory pressure, CPU overuses, and other exceeding resource consumption for collecting and streaming data to Fluentd.

VII. Related Work

CloudBench [3] and CloudPerf [4] are performance test frameworks for cloud-based deployments including distributed environments. CloudPerf also provides workload modeling and dynamic event injections, which our framework does not provide. asv [2] and ReBench [5] are tools for generic benchmarks. However, they do not focus on performance characteristics of container images, and hence they do not have any functionality for periodic image building nor specialized containers for metrics collection. asv also supports result visualization but does not provide the summary views we described in Section IV-E.

Camargo et al. proposed a performance test framework to break down individual service performances for a microservice application [22]. They focused on throughput and elapsed time for performance analysis and did not collect other metrics such as CPU usage and Linux perf.

Our framework can also be regarded as a benchmark collection for testing complex cloud applications. Cockroach Labs provide a benchmark collection with iPerf, sysbench, TPC-C, etc. in their performance report [23]. Our example workloads use a different set of benchmarks because our focus is to compare container images, not cloud environments. However, we can extend our framework to support their benchmark collection as well.

VIII. Conclusion

In this paper, we described the design and implementation of our continuous performance test framework for containers. The framework consists of a periodic image builder, benchmark driver, metrics collection, and visualization tool to aid with the performance characterization of container images. We also demonstrated our framework with case studies of performance characterization of nine popular benchmarks and sixteen container images. The demonstration led to the discovery of implications for container development and testing: performance variation due to deployed environments and characteristics of Alpine and JDK images.

Acknowledgement

We thank the anonymous reviewers for their helpful feedback. We also thank Marcelo Amaral, Scott Trent, and Seetharami Seelam for their feedback on this paper.

References

[1] Docker Inc., "Docker Hub," https://hub.docker.com, Accessed 2020-06.

[2] "airspeed velocity — airspeed velocity 0.4.1 documentation," https://asv.readthedocs.io/en/stable/, Accessed 2020-06.

[3] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. d. Silva, "CloudBench: Experiment automation for cloud environments," in *2013 IEEE International Conference on Cloud Engineering (IC2E '13)*, 2013, pp. 302–311.

[4] N. Michael, N. Ramannavar, Y. Shen, S. Patil, and J.-L. Sung, "CloudPerf: A performance test framework for distributed and dynamic multi-tenant environments," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*, 2017, pp. 189–200.

[5] S. Marr, "ReBench: Execute and document benchmarks reproducibly," August 2018, version 1.0.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '06)*, 2006, pp. 169–190.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 143–154.

[8] Standard Performance Evaluation Corporation, "SPECjbb®2015," https://www.spec.org/jbb2015, Accessed 2020-06.

[9] "The Python Performance Benchmark Suite — Python Performance Benchmark Suite 1.0.2 documentation," https://pyperformance.readthedocs.io, Accessed 2020-06.

[10] Elasticsearch B.V., "Rally 2.0.0 — Rally 2.0.0 documentation," https://esrally.readthedocs.io, Accessed 2020-06.

[11] The Apache Software Foundation, "ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4," https://httpd.apache.org/docs/2.4/programs/ab.html, Accessed 2020-06.

[12] "OpenLiberty/sample.daytrader8," https://github.com/OpenLiberty/sample.daytrader8, Accessed 2020-06.

[13] "kdlucas/byte-unixbench: Automatically exported from code.google.com/p/byte-unixbench," https://github.com/kdlucas/byte-unixbench, Accessed 2020-06.

[14] DataDog, "Cloud Monitoring as a Service — Datadog," https://www.datadoghq.com/, Accessed 2020-06.

[15] "Prometheus - Monitoring system & time series database," https://prometheus.io/, Accessed 2020-06.

[16] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt, "Bolt: Towards a scalable docker registry via hyperconvergence," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019, pp. 358–366.

[17] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 199–212.

[18] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019, pp. 331–344.

[19] M. Bostock, V. Ogievetsky, and J. Heer, "D^3: data-driven documents," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 12, pp. 2301–2309, November 2011.

[20] Elasticsearch B.V., "Kibana: Explore, visualize, discover data — elastic," https://www.elastic.co/products/kibana, Accessed 2020-06.

[21] T. Chiba, T. Yoshimura, M. Horie, and H. Horii, "Towards selecting best combination of SQL-on-Hadoop systems and JVMs," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD '18)*, 2018, pp. 245–252.

[22] A. de Camargo, I. Salvadori, R. d. S. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services (iiWAS '16)*, 2016, p. 422–429.

[23] P. Bardea, C. Dillon, N. VanBenschoten, and A. Woods, "2020 cloud report," Cockroach Labs, Tech. Rep., 2019.