ConfAdvisor: A Performance-centric Configuration Tuning Framework for Containers on Kubernetes

Tatsuhiro Chiba, Rina Nakazawa, Hiroshi Horii
IBM Research
19-21 Nihonbashi Hakozaki, Chuo-ku, Tokyo JAPAN
{chiba, rina, horii}@jp.ibm.com

Sahil Suneja, Seetharami Seelam
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY USA
{suneja, seelam}@us.ibm.com

***Abstract*—Configuration tuning of software is often a good option to improve application performance without any application code modifications. Although we can casually change configurations, it is not easy to apply optimal configurations, as optimal configurations require deep knowledge of the underlying system. This is problematic because applications with suboptimal configuration result in poor performance. As container and container management systems have emerged as an application platform on the cloud, configuration tuning becomes even more challenging because containers add more complexity to the application performance. We need to consider not only fundamental misconfiguration but also container image verification, deployment configuration, application characteristics awareness based on metrics and logs. Although previous knowledge regarding how we should tune configurations for a system software is sometimes available, knowledge about performance tuning practices is neither normalized nor reusable to expand on any advice for misconfiguration to the containers. Even in the cloud-native environment, there is no centralized service to deliver knowledge continuously to application containers nor a framework to develop a misconfiguration fix rule for a container throughout its lifetime. In this paper, we propose a performance-centric configuration tuning framework for containers on Kubernetes, named ConfAdvisor, that enables containers to achieve a higher performance by validating various misconfigurations adaptively. ConfAdivsor gives config tuning advice to application containers, images, and Kubernetes specs and also provides a development framework to build configuration validation rules. We present the design of ConfAdvisor and provide several case studies to tune application containers in the real world.**

***Index Terms*—cloud computing, containers, microservices, kubernetes, configuration tuning, performance tuning**

I. Introduction

Optimizing system software performance when the software does not achieve the expected performance has long been a challenging problem. Most modern software programs provide a bundle of configurations ("configs" for short) that enable us to change their internal processing without application code modification, so most application developers, performance engineers, and system administrators consider config tuning a viable option to improve performance. The advantage of config tuning is that we can improve the application performance by ourselves, but unfortunately there is also the potential for a huge performance drawback if an incorrect config (i.e., a misconfig) occurs.

Numerous studies [1][2] have revealed that misconfigs are very often the underlying cause of system outage and performance degradation issues. For example, a previous study [2] reported that the number of issues categorized as misconfig in a Hadoop cluster was higher than that categorized as software bugs, and fixing misconfigs is very time consuming. Another study [1] reported that misconfigs exist everywhere in major open source software applications and have been responsible for severe performance problems.

Why does misconfig happen so many times in system software? One reason is the increase of complexity, which now numbers a hundred or more tunable parameters. As described in an earlier study [3], several hundred configurable parameters in recent system software applications are quite general, and to make matters worse, these parameters increase with every update. Since setting proper configs is essentially difficult for almost all general users, a performance engineering task requires expert knowledge about the system and workload characteristics based on painstaking investigation such as logging and monitoring. As a consequence, poor performance resulting from misconfigs will create huge annoyance every single time. Many studies have tackled the misconfig issue with various approaches such as config anomalies detection [4] and config validation tools [5]. Several studies [6][7] have investigated how to improve performance via config tuning. Still, it remains an open problem in containers and microservices on the cloud.

Microservices architecture has recently gained popularity as a software system to deploy applications and services on the cloud [8]. Combining microservices with complementary technologies such as DevOps and CI/CD can bring many benefits, including scalability, portability, and reduced time-to-market. For example, Facebook has deployed new code for bug fixes, testing, and production twice a day by harnessing the power of microservices [9]. Along with microservices, containers and container orchestration systems, which work behind the scenes as a microservices platform, have emerged. System software programs are also containerized and can be more easily distributed to different environments as a container image than ever before.

This convenience of portability and extendibility has dramatically changed the software development cycle, but the possibility of misconfigs sneaking into system software is higher than ever due to several difficulties unique to containers and microservices. First, a base image might contain a serious

misconfig that would then possibly be carried over to a successor image without being fixed. Second, many versions of software (e.g. Redis v4 or v5, MongoDB 3.6 or 4.0) will be maintained in the same orchestration system, and since tuning tips are different in each version, we need to keep upgrading the proper configs for each container. Third, we need to keep in mind not only software versions but also deployment environments such as resource quota and architecture.

To address these challenges, in this paper, we propose a sustainable performance-centric configuration tuning framework for containers: ConfAdvisor. ConfAdvisor offers a pluggable and programmable analysis framework to develop performance improvement config advice for container images, containerized applications, and Kubernetes deployment specs. By utilizing various telemetry systems on Kubernetes, the ConfAdvisor framework gives unified key-value data including configs and metrics to analysis plugins. ConfAdvisor also provides a built-in what-if rule engine for processing defined rules in analysis plugins. The rules and advice in a plugin are written in a declarative model, and config advice is generated when a what-if rule is applied. The ConfAdvisor engine embraces not only the declarative model but also an imperative model to construct config advice and rules. Advice and rules are also extended by adding user defined functions (UDFs) to evaluate complex analytics, which makes it easy to incorporate imperative models into analysis plugins. By introducing these two styles, ConfAdvisor conveys simplicity and flexibility to the performance engineers and domain experts who write the rules and advice. Moreover, ConfAdvisor delivers config advice as a service about user containers and images managed on Kubernetes. With this advisor service, an application owner who is dealing with misconfigured containers or images on Kubernetes can easily acquire continuous personalized performance improvement advice about an application's whole life that is based on its config and metrics.

We have designed ConfAdvisor for implementation on Kubernetes. In addition to the ConfAdvisor framework, we implement config crawling and integrate it with an existing Kubernetes-native telemetry and data curation system. We developed typical config tuning rules for several software applications (Liberty, Nginx, Node.js, MongoDB, Redis, and Cassandra) on ConfAdvisor and then conducted case studies to determine how well ConfAdvisor works. The results showed that ConfAdvisor achieved up to 2.5x, 1.1x, and 1.4x performance improvement in Cassandra, Liberty, and MongoDB, respectively, over the default config. The main contributions of this paper are as follows.

- We propose a ConfAdvisor framework to write config advices and rules in a declarative model based on the config and metrics of image and container.
- We build a ConfAdvisor service on this framework, which delivers an optimal config for images, containers, and Kubernetes specs to user and system.
- We demonstrate that the ConfAdvisor service helps to improve the performance of several containers with optimal config over default config.

II. Background

A. Linux Containers and Images

Linux-based container technology has been generating a lot of interest among users and these days is widely utilized as an application deployment platform in various environments from laptops to the cloud. A container is made up of a process-level virtualization that isolates applications by encapsulating application binary, libraries, and configs into a small package. The core concept of Linux container technologies is based on kernel-level features such as control groups (cgroups) and namespace, which give information about resource limitation and resource isolation, respectively. Thus, using containers results in a system that is more lightweight, has a lower footprint, and is much more efficient than hypervisor-based virtualized systems [10].

A container image is a standalone executable package that includes everything needed to run a container. An image consists of multiple immutable layers based on a copy-on-write (CoW) union mount filesystem such as OverlayFS [11]. A container image makes applications portable and copiable, and thus it is easy to distribute containers to a different environment. Moreover, they enable a new image to be built on top of the base image. This image extendibility produces a big container ecosystem: anyone can publish a custom image on an image registry service, and then anyone can download that image and of course reuse it for further extension. In DockerHub, which is a major container image registry, more than one million public images are available [12]. Prebuilt system software containers such as MongoDB, Redis, Cassandra, etc., which we focus upon in this paper, are also available in DockerHub.

B. Container Orchestration System

The increased usage of containers has prompted the emergence of container orchestration systems such as Kubernetes [13] and Apache Mesos [14]. A container orchestration system provides scalability and maintainability for containers running on a cloud, essentially acting as a cloud scale operating system for containers.

At present, Kubernetes has become a de facto container orchestration system and is continuously being updated by the open source community. Originally developed by Google [15], Kubernetes has adopted many units of knowledge from the predecessor scheduling system, namely, Borg [16]. With its utilization of features to container scheduling, auto scaling, and self-healing after failure, Kubernetes contributes not only to the reliability of container applications but also to the ease of building cloud-native applications on the cloud. As a consequence, many cloud-native systems (monitoring, logging, service discovery, etc.) have been released as a part of the Kubernetes ecosystem.

Among the many concepts and features of Kubernetes, the pod is a key abstraction in the Kubernetes object model. A pod is the minimal deployable unit that enables collocating containers with the same network namespace and shared storages.

```
<server description="new server">
  <featureManager>
    <feature>microProfile-2.0</feature>
  </featureManager>
  <!-- optimal thread size depends on env or workload -->
  <executor coreThreads="1" maxThreads="10"/>
</server>
```

```
worker_processes  100;              # depends on env
events {
  worker_connections  1024;         # depends on env
}
http {
  sendfile     off;                 # better set on
  tcp_nopush   off;                 # better set on
  keepalive_timeout  60;            # depends on role
}
```

Fig. 1. Performance-sensitive config examples: server.xml in Liberty (top) and nginx.conf in Nginx (bottom).

As shown in an earlier work [17], multi-container application patterns are beneficial for maintaining an application container nearby, so we also adopt the ambassador pattern for monitoring application metrics in this paper. Besides, Kubernetes provides a volume to retain data beyond the container life cycle. A volume is mounted to a directory and then containers in a pod can access new files in the volume transparently even though they are bound to a specified directory. This feature also enables a new config to be attached over the existing contents in an image at the time of deployment.

C. Configurations

Application Configs: Most applications externalize their configs to a config file and change the internal parameters later. The config format, which is different for each software (e.g., yaml, json, xml, conf, or the original schema), is loaded when the application is initialized or automatically reloaded when a change is detected.

Examples of config files in Open Liberty (Liberty for short) and Nginx are given in Fig. 1. Liberty is a Java application server that loads configs from *server.xml*, and Nginx is a Web server that loads configs from *nginx.conf*. In Liberty, we might have an opportunity to set the bound of an auto thread tuning range between *coreThreads* and *maxThreads*. Since too few *maxThreads* may limit the application performance, we should increase the value before deployment. The optimal value depends on the execution environment (e.g., how many hardware threads are available), so it needs to be tuned on the basis of the environment. In the Nginx case, *keepalive* is known as an important parameter to improve response time, but we should carefully choose the value or turn off the feature because it may be harmful when Nginx handles many lightweight connections.

Deployment Configs: A deployment spec should be carefully set to improve application performance in Kubernetes. The spec includes several important statements, such as the number of replicas, network service type, volume mount type, resource quota, and so on. Since Kubernetes provides a self-healing from failure mechanism, users may not notice that their container is terminating again and again due to an out-of-resource error (e.g., out-of-memory error). This unintentional restart may be harmful because the container will require an additional ramp-up, which takes time. Moreover, we should apply auto scaling options adaptively to a deployment spec when an application container runs out of resources.

Other Configs: This paper focuses on the above two types of configs, but ConfAdvisor is not limited to them. For example, the host-level config (e.g., */etc/sysctl.conf*) has candidate tunable parameters that affect container performance. This and certain other configs are beyond the scope of this work.

D. Metrics

Metrics are always important when it comes to evaluating resource usage and performance data from running containers. Below, we discuss which types of container metrics are available in Kubernetes.

Basic Metrics: Basic metrics such as those related to CPU, memory, network, and disk usage represent the raw measurement of resource usage, and knowing these can be helpful to determine whether a container utilizes the appropriate amount of resources in its workload. Since monitoring these metrics is always taken into consideration, a per-container metric collecting tool (e.g., cAdvisor) is utilized in the Kubernetes ecosystem.

Application Metrics: Application metrics represent the internal statistics of applications, such as the number of active or failure connections, summary of request or response size, performed cache entry hit or miss ratio, precise memory usage, total number of performed garbage collections (GC), and so on. Many systems provide an endpoint to extract these metrics, so they are more helpful than basic metrics for understanding application characteristics. Currently, Prometheus [18] stores and gather these metrics in the Kubernetes ecosystem.

Utilizing Metrics: After collecting metrics, some of them are utilized in visualization dashboard tools, which are helpful for understanding when an anomaly happens and for intuitively knowing which anomaly events are likely to happen frequently. However, when a system detects an anomaly that has been caused by an injected misconfig, most tools do not care about why it happens or which config should be fixed. Grafana, which is a widely used metrics dashboard tool, shows the metrics in various ways, but it does not provide any action plans about what to do next.

III. Challenges and Motivations

In this section, we take a look at several existing approaches and tools and then clarify what we can learn from them and what we need to consider in cloud-native applications.

A. Existing Approaches: What should we learn from?

Config management tool: System config management tools such as Ansible, Chef, and Puppet have been developed to validate software installation and configs in the infrastructures. These tools are helpful to keep the system stable and idempotent automatically.

However, these tools are basically designed to manage not containers but servers; at least the SSH capability is required to

TABLE I
THE NUMBER OF CONFIGS IN OFFICIAL DOCKER IMAGES

Software & Config	No. of tuning knobs default	No. of tuning knobs available	image name
Nginx (nginx.conf)	20	732	nginx:1.15
Apache2 (httpd.conf)	72	1011	httpd:2.4.37
Redis (redis.conf)	0	103	redis:5.0
MongoDB (mongod.conf)	8	127	mongo:4.1.4
Open Liberty (jvm.options)	0	146	open-liberty:javaee8
Cassandra (jvm.options)	32	763	cassandra:3.3.1

deliver configs to the system. This is a huge disadvantage for containers, as containers may not always permit SSH access [19]. Moreover, it is difficult to reflect container metrics or logs adaptively in their management model since they are not intended to improve performance.

Performance sensitive config tuning system: A wide variety of config optimization systems have been developed for application and cloud resource allocation to maximize performance with various approaches, including those based on heuristics [20][21], on machine learning [22], on building a performance model with white-box [23], and on black-box Bayesian optimization [24]. For example, Starfish [20] provides an optimized Hadoop config based on profiling and its heuristics. OtterTune [22] is a config auto-tuning system for DBMS that identifies the importance of tuning knobs on the basis of workload characteristics and then finds the best config by means of a Gaussian process regression model.

These specialized tuning systems are useful for finding out the best config, but they may not be well suited to container workloads. One big concern is that containers are casually torn down and then cannot always dedicate resources. Noisy neighbors may reside in the same host, or the container orchestration system may kill and reschedule its container to free up host resources. That means the tuning system may not have enough time to determine the best parameters, and it also may need to rebuild the models. Moreover, we might need to keep maintaining each tuning system independently because they are specialized and do not rely on a generalized framework.

Misconfiguration validation language: Many frameworks have recently been proposed to fix and validate misconfigs in advance in order to avoid system outage. Some of the works in this vein [5][6][25][26] have also provided domain specific language (DSL) to describe the specifications or rules for config validation. Their declarative specifications can simplify our config validation tasks compared to imperative ones, and they can correspond to a wide range of config errors due to their completeness. However, a specialized DSL and grammar make it difficult to write new specifications due to the huge learning curve involved. Moreover, they are not designed for performance improvement tasks.

B. Motivations: What should we consider in cloud-native applications?

Since applications are containerized and their execution environments are rapidly moving from a dedicated bare metal system to a container cloud, performance engineering for containers is essential to achieve faster performance, even on the cloud. However, typical performance engineering tasks and procedures are neither standardized or open. As such, we also need to change the performance engineering methodology, tools, and our mentality so as to be more in alignment with the cloud-native way. In this section, we summarize the perspectives we need to consider more for containers.

Validating default config in image: A huge number of software images are available, and some of them are published as official images. However, the quality of the configs inside these images is unknown. Table I lists how many configs are preset as default and how many tuning knobs exist in each image. The table includes six representative official containerized images that we picked up from DockerHub. In the case of the Nginx and Apache2 containers, preset config accounted for just 2–7 % of the total number of directives. The Redis container has no default config, so it delegates all config setting to users. The MongoDB container has a config inside the image, but only a few parameters are preset. For Liberty and Cassandra, we compared not application configs but a runtime config, which is a Java Virtual Machine (JVM) option. Since these use different JVMs (Liberty uses OpenJ9 and Cassandra uses OpenJDK), the number of available options is also different. For the Liberty case, there are no preset JVM options. In contrast, Cassandra already includes some performance-aware JVM options, but it is not clear whether all configs are effective.

Of course, not all configs are related to performance, and some default configs are loaded without having explicit directives. Even so, there is a huge gap between default config and optimal config, even in containers.

Throughout entire container life cycle: From the viewpoint of container life cycle, we have three chances to tune configs in the application container. First, we can validate configs in the images. Most images do not include essential configs (as shown in the above example), even if they are official. Thus, image inspections are required before container deployment.

Second, we can notice misconfig in a container and deployment spec when it is launched. This type of misconfig is caused by the deployment environment (such as resource quotas pertaining to CPU and memory) or by underlying architecture differences (such as x86_64 or ppc64le). They become apparent after the containers have been scheduled or at the time of deployment. This type of inconsistent config must be fixed in order to avoid unexpected container stopping and performance degradation.

Third, we can improve configs after the application container has been running for a while. This is known as workload-aware config tuning. Since it is difficult to predict what kind of workloads are running, the optimal config settings change along with the application characteristics [27]. Therefore, it is important to build optimal config advice adaptively on the basis of metrics or logs in order to drive running container performance up to the next level.

Figure 2 shows an abstraction of these three config tuning

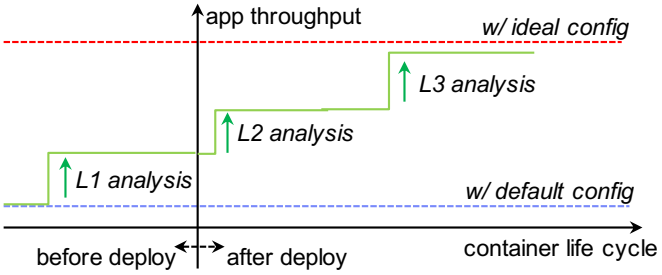

Fig. 2. Conceptual diagram of tuning timings throughout container lifetime.

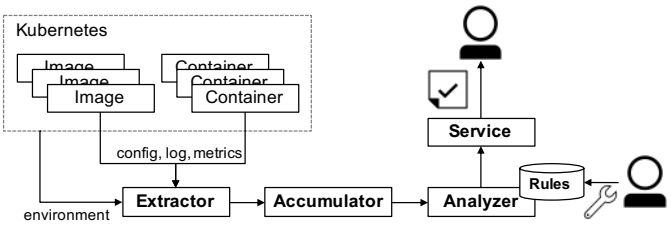

Fig. 3. Design overview for config advice flow.

timings to achieve ideal performance if the config tuning affects application throughput. These timings require different levels of information, so we respectively define them as L1, L2, and L3. As a result, a config tuning system should cover the entire lifespan of a container in order to apply sustainable config tuning in a cloud-native environment.

Advisor Service - From what happens to why it happens: Application diagnostic tools such as Prometheus and Grafana are helpful to understand metrics in a system through a rich dashboard or query-able interface. Monitoring dashborad provides a good summary or performance anomaly and then reveals what happened in the system intuitively, but the user still needs to investigate why something happened or what config should be changed through multi-dimensional analysis. For those users who do not have sufficient knowledge about the target application, there is a risk that they will wander around the config tuning forest aimlessly.

Moreover, although a vast amount of knowledge and tuning information is available on public sites like Stack Overflow or official document, the predecessor's knowledge has been opened but not returned to the tuning. To achieve continuous config tuning, we should build a service to deliver optimal config to the containers by standardizing the knowledge.

IV. Design

A. Principles and Overview of Architecture

Here, we summarize the required properties in the config tuning system on the basis of the discussion in the previous section. Our objective is to build a sustainable config tuning framework for container performance on the cloud, so the following five properties need to be taken into consideration.

- **Scalable:** low monitoring overhead and fast advice
- **Descriptive:** writing rules easily and minimizing learning curve as much as possible
- **Extendable:** the capability to add new tuning analytics
- **Unchanged:** no modification in containers and images
- **Affinitive:** utilizing existing cloud-native ecosystem

In general, there are three basic building blocks for a config tuning system: extracting, accumulating, and analyzing configs. They are common pieces even in containers and microservices on the cloud. Figure 3 provides an overview of how these three components collectively work and how the verified config tuning advice is delivered to users. In the following sections, we discuss design options for involving the above properties in each part.

B. Extracting Configs, Metrics, and Logs

Scalability is a mandatory feature in a config and metric extractor on the cloud. The extractor has to find out what config exists in containers and images and keep tracking the config evolution while the container is running because the config will probably be updated at some point either manually or by the system. As new images are continuously registered in a registry service and new containers are launched in any Kubernetes-managed node, the extractor should detect new images and containers immediately. Moreover, the extractor should not require any user-side modifications in containers and images, as code changes (such as instrumenting an agent into a container) impose a burden on users.

In terms of metrics and logs, both scalability and no user-side modification are important. Since not all containers expose an endpoint to extract metrics, the extractor should make sure which container opens the endpoint and then periodically crawl all available metrics from them in a scalable way.

C. Accumulating Configs, Metrics, and Logs

After extracting the data, the accumulator has to index it in order to identify where it came from, and of course it needs to keep it for a while. In terms of container image indexing, registry services typically require unique attributes such as namespace, image name, and tag to register an image, so the accumulator can append these as an index. To run containers on Kubernetes, all containers must include namespace, deployment type, pod name, and container name. Moreover, each container has a unique container ID assigned by container runtime (e.g., Docker). By attaching these names to the index, configs and metrics should be searchable in the accumulator.

D. Config Tuning Approach

Config tuning approaches are roughly divided into two types: black-box and white-box. Black-box approaches based on a well-structured search algorithm [28] or Bayesian optimization (BO) [29] are powerful in terms of determining the best config without special knowledge, and they may achieve better performance than predefined heuristics rules. One drawback of black-box approaches is that the search task

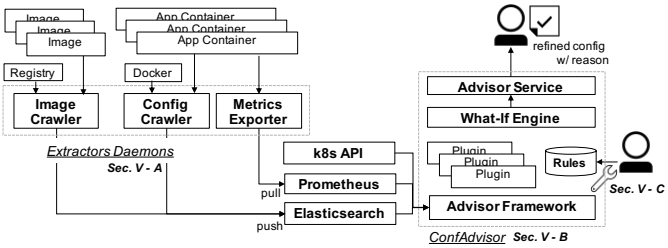

Fig. 4. Overall architecture of end-to-end config advice in ConfAdvisor.

```
"jvm.options": [
{
  "key": "mx",
  "what-if": "current.mx > current.limits_memory",
  "advice": "current.limits_memory * 0.75",
  "order": 0,
  "message": "should keep mx less than memory limits"
},
{
  "key": "ms",
  "what-if": "current.ms != advice.mx",
  "advice": "current.limits_memory * 0.75",
  "order": 1,
  "message": "should keep ms same as mx"
}]
"server.xml": [
{
  "key": "maxThreads",
  "what-if": "current.maxThreads < (current.cpu * 5)",
  "advice" : "current.cpu * 5",
  "order": 0,
  "message": "should start with 5 * vcpu"
}],
"k8s": [
{
  "key": "scale_pod",
  "what-if": "rate_cpu_usage(vars.interval) > 0.8"
  "advice": "current.replica += 1",
  "order": 0,
  "message": "check average cpu usage in last interval"
}]
```

Fig. 5. Example rules used in Liberty plugin (JSON format).

is time-consuming. White-box approaches based on expert validation rules [5][6] are practical and applicable in many types of config tuning with well-known heuristics. While not completely opposite, both types have a complementary style of config tuning. We will take the black-box approach into consideration in the future, but as it is excessive for validating configs of images and containers, especially in the ramp-up phase, we limit ourselves to the white-box approach here.

E. Imperative or Declarative

When describing our heuristics for a config tuning rule, there are two potential paradigms for writing the rule: imperative or declarative. The imperative style provides flexibility to a rule in terms of allowing it to express complex analytics algorithms directly, while the declarative style lends a rule a simplicity that makes it easy to understand. The performance engineers who write the rules have different levels of skill, so the tuning framework should enable a paradigm that achieves both simplicity and flexibility.

V. Implementation

On the basis of the design considerations discussed above, we implement a config advisor framework and service on Kubernetes that integrates data extractors and a data curator system. An overview of the end-to-end config advice architecture in ConfAdvisor is provided in Fig. 4. In the following section, we describe the details of each component.

A. Crawling Configs and Metrics

Image and Config Crawler: We utilize an open-source crawling tool called an agentless system crawler (crawler for short) to inspect configs and other information including packages, files, and environments [30]. The crawler resides in each host as a daemon process, and it automatically and periodically captures a specified set of entities from active containers without enforcing guest cooperation.

To inspect configs inside an image, we implement an image crawling feature on the original crawler. The image crawler keeps watching the image registry to see when new images are registered. Once a new image is registered, the image crawler launches a new container in a sand box space. By mounting a sleep binary in a container and overwriting an entry point, the image crawler can capture configs in any image without unexpected container halt.

In terms of the config crawler, we also extend the original crawler, which can detect a newly launched container immediately, by monitoring container creation events through a Unix domain socket in a host Docker runtime, since the original crawler checks the configs of containers only periodically.

Finally, the retrieved configs are normalized and then stored in Elasticsearch with an index and timestamp. We leverage an open-source config parser called the Augeas tool [31] to convert configs into a tree format and then normalize the tree format to a simple flatten dictionary. With this normalization step, an advice rule and analytics routine can access each config value uniformly in a key-value manner.

Metrics Exporter: For collecting application metrics, we use Prometheus and a Prometheus exporter. Prometheus is an open-source monitoring system and time-series database that is tightly integrated with Kubernetes. Various exporters that can expose metrics as a Prometheus format are maintained officially. An exporter basically works as an adapter container style [17] beside a target container, so we introduce a corresponding exporter into an application pod. Although we also introduce a raw log file extractor (e.g., logstash), we do not utilize the raw log in the framework at this time because the metrics summarize application statistics and the available data would be duplicated. The exposed application metrics are automatically pulled by Prometheus and then consumed in the ConfAdvisor framework.

B. ConfAdvisor Service and Framework

ConfAdvisor consists of three building blocks: an advisor service, an advisor framework, and an advisor rule engine. All components are implemented in Python. We introduce these components one by one.

Advisor Service: The advisor service is a frontend to handle an advice request between users and a backend analytics engine. Users specify a target container and have the option of

setting the config advice type, application type, etc. in a query. Once it receives an advice request from a user, the advisor service loads a corresponding plugin to build personalized config advice for the image or container. After processing all config verification rules and analytics in the plugin, the advisor service returns verified config advice to the user. We used gRPC to implement the APIs so that the advisor service could be run as part of the microservices.

Advisor Framework: The advisor framework exposes normalized configs and metrics to the analytics plugins. On the basis of the query, the advisor framework collects all related topics from data sinks such as Elasticsearch, Prometheus, and Kubernetes. Then it provides a unified key-value dataset or materialized time-series dataset to be utilized in the plugins. The advisor framework also introduces APIs or utility functions that enable us to easily write rules or analytics code. According to the rule specs in Fig. 5, for example, the scale_pod rule uses a predefined function named rate_cpu_usage() that computes the average CPU usage in a specified interval. The advisor framework introduces statistical functions similar to the Prometheus functions.

Analytics Plugin: ConfAdvisor utilizes a plugin system to introduce flexibility and extendibility to config analytics. When a request is made, a corresponding plugin (e.g., mongodb_plugin or redis_plugin) is selected automatically in ConfAdvisor and then it loads declarative config validation rules (showing them later) for each application. A plugin also functions as a placeholder to enhance UDFs written in Python, which enables us to write complex analytics imperatively.

Config Advice Rule Spec: Here, we define the config advice rule spec for writing declarative rules. Figure 5 shows example rules that are used in the Liberty plugin. This rule contains four rules in total: two in jvm.options, one in server.xml, and one in the Kubernetes environment. Developers can append any rules here on the basis of the following format. The important predicates exist in the *what-if* and *advice* elements. Once a predicate in *what-if* returns true, the rule engine evaluates a predicate in *advice*, which is a candidate validated config value for a specified config (e.g., mx, ms, or maxThreads). We can apply any Python code in these predicates, and can also call predefined functions such as rate_cpu_usage() in a scale_pod rule example or UDFs loaded in a plugin. All of the current configs and metrics for the target container are accessible through the *current* object, so we can minimize the evaluation code to make it as simple as possible. Using raw Python code for the predicate reduces the rule writing difficulty compared to using a specialized external DSL, so this constraint in the spec makes it easy to understand for both users and developers.

What-if Rule Engine: The rule engine is a core processor designed to consume the above rules on the fly. We implement it on jinja2, which is a powerful and widely used template engine in Python. Upon receiving a query, the rule engine tries to apply all related rules one by one for each config in a target container and then generates config advice through advice predicate evaluation with other entities on the basis of defined rule specs such as message, reason, reference, and so on. The rule engine also computes the dependencies between each rule so that any advice that has been generated is purified by all related rules. All components in ConfAdvisor are stateless in order to achieve high scalability.

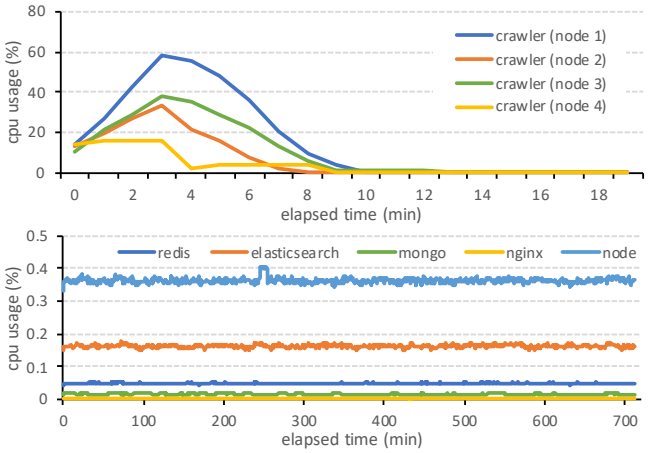

Fig. 6. 3-mins moving average of CPU usage: config crawler (top) and metrics exporter (bottom).

C. Advisor Rules and Plugins

We are currently in the midst of developing various performance improvement config rules and analytics as a plugin in ConfAdvisor. We implemented many rules on top of ConfigAdvisor, and at this moment, it supports the following six software applications: Liberty, Nginx, Node.js, MongoDB, Redis, and Cassandra. Table II shows examples of config advice rules and advice picked up from the ConfAdvisor plugins. Since the rules are written in the declarative style, they can be easily understood when evaluated. ConfAdvisor also gives revised configs as a key-value format with reference, so the user or system can take over the knowledge about which configs need to be fixed and why they should be fixed.

VI. Evaluation

We ran two Kubernetes cluster systems on individual IaaS clouds via IBM Cloud Private CE, which is a package to set up Kubernetes and related ecosystems easily on our infrastructure. The first Kubernetes cluster (cluster A) has four nodes on a Xen-based x86_64 architecture, where each node is equipped with eight vCPU cores, 16 GB RAM, and a 400-GB disk. The second cluster (cluster B) has four nodes on a KVM-based ppc64le architecture, where each node is equipped with 32 vCPU cores, 16 GB RAM, and a 200-GB disk. All systems run on Ubuntu 16.04.5 (kernel: 4.15.0-33-generic) and are installed with Docker 18.06.1-ce, Kubernetes 1.11.1, Prometheus 2.3.1, and Elasticsearch 5.5.1.

A. Runtime Overhead

Config and metrics crawling: Figure 6 shows the 3-min moving average of CPU usage for two extractor processes on

TABLE II
EXAMPLE OF CONFIG ADVICE PSEUDO RULES AND ADVICE AVAILABLE IN CONFADVISOR

software	config	level	what-if rule description	advice description
Liberty	server.xml	1	maxPoolSize != coreThreads	start with 1:1 mapping to the coreThreads for database connections [32]
Redis	redis.conf	1	maxmemory == None	maxmemory = memory_limit * 0.8, maxmemory-policy = allkeys-lru [33]
Node.js	k8s spec	3	rate_cpu_usage(1h) > 0.9	Increasing replica since Node.js is bound on a single CPU core [34]
MongoDB	mongod.conf	2	cacheSizeGB > memory_limit	storage.wiredTiger.engineConfig.cacheSizeGB = memory_limit / 2
Cassandra	jvm.options	2	-XX:MaxHeapSize > 16GB	-XX:+UseG1GC [35]
Cassandra	jvm.options	1	openjdk_vesion < 1.8.0_192-b01	-XX:+UseCGroupMemoryLimitForHeap [36]
Cassandra	k8s spec	2	isParsisted(/var/lib/cassandra/data)	Using Persistent Volumes

TABLE III
CONFIG CRAWLER PERFORMANCE

	node 1	node 2	node 3	node 4
No. of containers	89	65	74	23
Elapsed time (sec)	326	130	282	75

TABLE IV
CONFADVISOR QUERY PROCESSING THROUGHPUT

	Rule throughput (query/sec)				
category	**avg**	**stdev**	**min**	**max**	**median**
L1/L2	25.6	1.93	20.1	29.7	25.7
L3	3.32	1.92	0.56	5.55	4.39

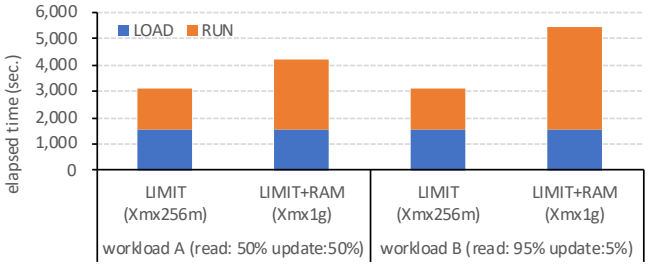

Fig. 7. Comparison of config tuning for Cassandra container on Kubernetes using YSCB benchmark.

cluster A: config crawler (top) and metrics exporter (bottom). Config crawlers are running on all four nodes. In our Kubernetes cluster A, 89, 65, 74, and 23 containers were respectively running on each of the four nodes; the results are also shown in Table III. Since the config crawler works sequentially, crawling performance depends on the number of running containers. Moreover, the config crawler captures not only config files but also other data (such as the installed package list) in the container, so crawling performance also depends on the content size of each container. As shown at the top of Fig. 6, config crawler spent 20–60% of the CPU in a short time (1–5 min). After calculating crawling throughput, we found that the config crawler achieved around 0.3 container/sec on average.

For container metrics, we used several Prometheus exporters (redis_expoter, node_exporter, etc.). Prometheus pulled metrics from each endpoint once per minute. As shown at the bottom of Fig. 6, metrics exporter overheads were quite small. Compared with exporters for system software, a node_exporter eats up more of the CPU, but it still requires less than 1% in 3 mins on average.

ConfAdvisor query performance: Next, we investigated the end-to-end ConfAdvisor performance. Table IV lists the rule query processing throughput and statistics in each analysis category. While processing L1 and L2 analysis, ConfAdvisor accesses the Elasticsearch and Kubernetes APIs. In addition to those APIs, for L3 analysis, ConfAdvisor needs to access the Prometheus API to run complex queries. Prometheus query response time fluctuated with the number of tasks. As a result, L3 analysis performed about 5x worse than the L1/L2 analysis.

B. Case Study: Config Advice with L1/L2 analysis

As indicated in Table I, the Cassandra container image already includes several performance tuning JVM options. Unfortunately, however, the included config does not consider that Cassandra runs as a container.

In the previous OpenJDK, JVM runtime did not recognize whether it runs on a container or not, so the runtime estimated available CPU or memory not from the cgroup limit but from the available host capacity [36]. The more recent OpenJDK considers the cgroup limitation, but we need to append several options explicitly if the underlying runtime is not the latest or is over a specified version (i.e., 1.8.0_192-b01 in Table II) because the options are backported from Java 10.

Fig. 7 shows the summarized performance of the YCSB benchmark (workloads A and B) for Cassandra on Kubernetes cluster A. We put a 1-GB memory limit on the Cassandra pod and loaded 1.5 M records (i.e., 1.5-GB dataset in total) with zipfian distribution, and the JVM runtime version was 1.8.0_181-b13. The default config could not run to the end due to a memory error, so we omitted the result. This occurred because the default config automatically set a 4-GB heap size that is computed from the host available 16-GB memory even in the 1-GB limit. LIMIT denotes adding -XX:+UnlockExperimentalVMOptions and -XX:+UseCGroupMemoryLimitForHeap options and LIMIT+RAM does -XX:MaxRAMFraction=1 option in addition. In the LIMIT case, JVM reserves a quarter size of the limit memory (i.e., 256 MB) for the max heap size, whereas it reserves up to the limit memory (i.e., 1 GB) in the LIMIT+RAM case.

As shown in the result, LIMIT was always faster than LIMIT+RAM in both workloads, and it achieved 1.7x and 2.5x improvement, respectively. This is because Cassandra also utilizes not only Java heap memory but also OS page

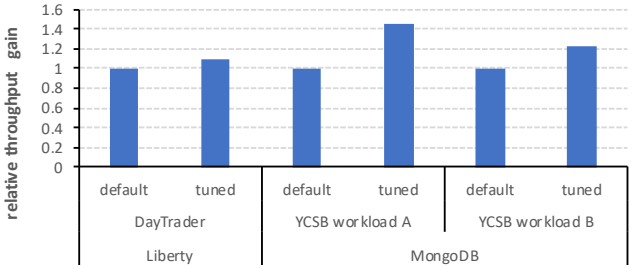

Fig. 8. Summary of performance improvement with memory config tuning.

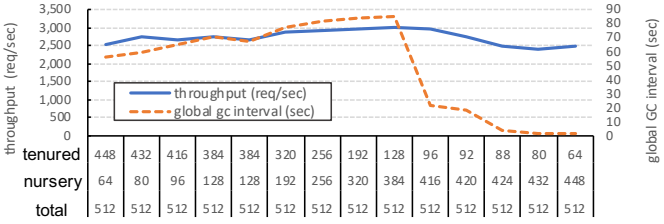

Fig. 9. Comparison of throughput and global GC interval while changing the ratio of nursery and tenured heap.

cache, so we should keep a half size of available memory for the Java heap to maximize the performance. To build this optimal config, we need knowledge about JVM, the container, Kubernetes, and Cassandra. In this case, ConfAdvisor not only prevents the unintentional memory error due to the default config but also provides the optimal config options (i.e., LIMIT) immediately to the container.

C. Case Study: Config Advice with L3 analysis

Here we introduce several case studies involving memory sizing based on workload, which is referred to as the L3 analysis in this paper. Then we discuss how much improvement we can achieve by setting the optimal config on a Kubernetes Cluster B.

Liberty: First, we introduce a memory sizing rule based on GC statistics for Liberty. We can utilize JVM internal metrics, including GC, which are exposed as the Prometheus format. To evaluate the Liberty performance, we used a DayTrader benchmark that emulates an online stock trading system for a Java EE application server and ran JMeter against the Liberty as a driver program of the DayTrader benchmark.

The left side of Fig. 8 shows a summarized result of relative throughput, comparing the default config with a tuned one. The default config reserved 512 MB of heap memory, which consisted of a 128-MB nursery heap and 384-MB tenured heap. The ratio (1:3) is regularly used unless we explicitly change it. By sliding the optimal ratio of nursery and tenured heap, we achieved a 10% throughput improvement over the default config.

Fig. 9 shows the details for throughput and global GC interval metrics while varying the ratio. When the nursery heap increased, throughput was improved over the default, whereas the global GC interval gradually became longer. Huge nursery space contributes to application throughput improvement when the heap is used effectively. However, if the heap is not tenured enough, many global GCs occur, which requires stop-the-world pause. In this result, global GC occurred four times more often with the 96-MB tenured heap than with the 128-MB one. In order to improve throughput within a resource quota constraint, the Liberty plugin can give advice to increase the nursery heap until a dramatic change occurs in the global GC interval metric, such as a shorter global GC interval event.

MongoDB: Next, we evaluate an in-memory cache sizing rule based on workload characteristics for MongoDB. By using mongodb_exporter, we can obtain internal statistics such as read/write operation counts and in-memory cache usage. We used a YCSB benchmark and evaluated throughput when utilizing an in-memory cache feature in MongoDB (i.e., WiredTiger) while changing the cache size. We deployed a MongoDB on Kubernetes with a 1-GB memory resource limit. For the YCSB setting, we ran workload A (50% reads and 50% updates) and workload B (95% reads and 5% updates) with one million uniform distribution records utilizing eight client threads.

The middle and right side of Fig. 8 show a summarized result of the improvement ratio in each workload comparing a default config with a tuned one. As shown, we were able to achieve a 40% improvement in workload A and a 20% improvement in workload B. Unfortunately, MongoDB does not consider the cgroups resource constraint when deciding the size of in-memory cache, so it tries to assign 7.5-GB memory, which is around half of the host equipped memory (i.e., 16 GB), for the internal cache as a default. This causes frequent memory thrashing because the memory usage is capped up to the specified limit. As a consequence, we cannot achieve expected performance in a containerized MongoDB with a resource limit. This example case shows that ConfAdvisor can give advice for resolving default misconfig depending on the environment.

Next, we investigate how much the cache size affects performance in order to determine the workload aware config advice when performing L3 analysis. The throughput of each workload while changing the cache size is shown in Fig. 10. In read-mostly workloads (i.e., workload B), the cache size did not affect performance improvement as long as the size was not over the resource limit. In contrast, it achieved the best throughput while setting 50% or 75% of available limit memory for the write-heavy workload (workload A). WiredTiger utilizes OS filesystem cache (page cache) as well as its internal memory cache. Too much internal memory cache leads to an inefficiency while running a write-heavy workload. Thus, ConfAdvisor can give advice to set half the size of available memory as a WiredTiger cache based on the read/write operation metrics if the config does not have that setting.

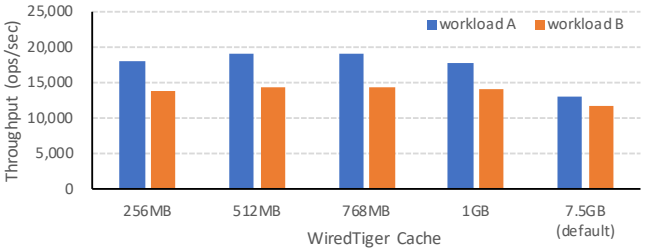

Fig. 10. Comparison of YCSB throughput on MongoDB container while changing wiredTiger cache size.

VII. Related Work

Performance anomaly analysis: X-ray [37] is a performance summarization system that utilizes binary instrumentation to track application execution. X-ray analyzes application flow and estimated performance costs coming from the metrics to the events in order to compute the likelihood of the root cause. Then it identifies which config made the performance worse. Roots [38] is application monitoring and diagnostic system for Web applications on a PaaS cloud. Roots runs a performance anomaly detector based on the analysis of the previous workload characteristics and then tries to identify a root caused component among candidate application pieces through a regression model and change point analysis.

These works determine the root cause of a performance anomaly on the basis of offline analysis with precise application trace data, while ConfAdvisor does all analysis at runtime. Online analysis is still effective for our objective at this time because the cost to analyze a target image, container, and spec is low. The trace analysis featured in the other studies will become more important in complex microservices, which consist of frontend Nginx and backend MongoDB, for example. We also intend to analyze trace data by taking inspiration from their works and distributed tracing techniques in the cloud.

Cloud scale config validation: Tang et al. [39] provide an end-to-end configuration management stack that runs on Facebook. Their system takes a configuration-as-code approach to compile a config from high-level source code. By integrating it with other systems such as code repository and the canary test framework, cloud scale config distribution systems for many of their services can be enabled. With ConfAdvisor, we also aim to offer a service that makes continuous config improvements in the DevOps cycle. However, their primary focus is config validation and compilation, whereas ConfAdvisor makes not only a config validation but also focuses on performance relying on metrics.

ConfValley [6] is a config verification system for cloud applications. It provides a verification DSL for writing validating specifications to apply expert heuristics. Similar to ConfValley, several works [5][26] have tried to detect incorrect configs before the occurrence of a fatal error with a special validation language. ConfValidator [26] has declarative syntax, but it does not have a tuning engine for config optimization. In contrast, while ConfAdvisor also provides the capability to write rules with the declarative format, it does not implement language itself but rather utilizes the existing template engine and Python grammar as much as possible.

Rule learning and parameter searching: Machine learning-based misconfig detection systems such as EnCore [40] and ConfigV [25] have been proposed. These systems learn config check rules or specifications from a training set of config files or rule templates. They then refer to the learned model to detect various types of errors (missing entry error, type error, etc.). BestConfig [28] finds the best config for various systems with a scalable sampling method and an optimized recursive bound search algorithm. Bayesian optimization-based approaches such as CherryPick [24] and BOAT [29] are also evolving in terms of config tuning because they can achieve near optimal performance with few samples. In future work, we plan to combine our heuristic rule-based approach with the essence of these works in the ConfAdvisor framework by rule upgrading and parameter searching. As a result, the coverage of ConfAdvisor will be expanded to long running container performance.

VIII. Conclusion

We proposed a sustainable performance-aware config tuning framework named ConfAdvisor for container images, running containers, and deployment specs on the cloud. Using declarative rules written by domain experts or general users as a basis, ConfAdvisor provides adaptive config advice for container workloads from beginning to end. We built this framework on Kubernetes and integrated it with telemetry systems on the Kubernetes ecosystem. Several case studies demonstrated that ConfAdvisor can achieve up to 2.5x improvement in Cassandra over default config, and also 1.1x and 1.4x improvement in Liberty and MongoDB, respectively.

References

[1] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043572

[2] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, July 2013.

[3] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 307–319. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786852

[4] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 619–634. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu

[5] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 416–430. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908083

[6] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 19:1–19:16. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741963

[7] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 154–168. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173206

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016.

[9] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, July 2013.

[10] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273025

[11] "Overlay Filesystem," https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[12] S. Singh, "Dockercon SF 18 Keynote," https://blog.docker.com/2018/06/day-1-keynote-highlights-dockercon-san-francisco-2018, 2018.

[13] "Kubernetes," https://kubernetes.io/.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444

[16] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[17] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[18] "Prometheus - monitoring system & time series database," https://prometheus.io/.

[19] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 313–319. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tak

[20] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, 2011, pp. 261–272.

[21] C. Steinbach and S. King, "Dr. elephant for monitoring and tuning apache spark jobs on hadoop," Spark Summit 2017. [Online]. Available: https://databricks.com/session/dr-elephant-for-monitoring-and-tuning-apache-spark-jobs-on-hadoop

[22] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1009–1024. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064029

[23] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 363–378. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[24] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[25] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 64:1–64:20, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133888

[26] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, ser. Middleware '17. New York, NY, USA: ACM, 2017, pp. 29–35. [Online]. Available: http://doi.acm.org/10.1145/3154448.3154453

[27] S. Suneja, R. Koller, C. Isci, E. de Lara, A. Hashemi, A. Bhattacharyya, and C. Amza, "Safe inspection of live virtual machines," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 97–111. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050766

[28] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 338–350. [Online]. Available: http://doi.acm.org/10.1145/3127479.3128605

[29] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "BOAT: Building auto-tuners with structured bayesian optimization," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 479–488. [Online]. Available: https://doi.org/10.1145/3038912.3052662

[30] "Agentless system crawler," https://github.com/cloudviz/agentless-system-crawler.

[31] "Augeas," http://augeas.net.

[32] "Tuning liberty," https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/twlp_tun.html.

[33] "Using redis as an LRU cache," https://redis.io/topics/lru-cache.

[34] "Node.js docs," https://nodejs.org/api/cluster.html#cluster_cluster.

[35] "Tuning java resources - choosing a java garbage collector," https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsTuneJVM.html.

[36] "Improve docker container detection and resource configuration usage," https://bugs.openjdk.java.net/browse/JDK-8146115.

[37] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 307–320. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/attariyan

[38] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 469–478. [Online]. Available: https://doi.org/10.1145/3038912.3052649

[39] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 328–343. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815401

[40] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 687–700. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541983