SPONSORED BY

Twistlock™

GEEK GUIDE

# Deploying Kubernetes with Security and Compliance in Mind

LINUX™ JOURNAL

# Table of Contents



**PETROS KOUTOUPIS** is currently a senior software developer at **IBM** for its **C**loud **O**bject **S**torage division (formerly **C**leversafe). **H**e is also the creator and maintainer of the **R**apid**D**isk **P**roject (http://www.rapiddisk.org). **P**etros has worked in the data storage industry for more than a decade and has helped pioneer the many technologies unleashed in the wild today.

**GEEK GUIDES:**
Mission-critical information for the most technical people on the planet.

## About the Sponsor

### Twistlock

Twistlock protects today's applications from tomorrow's threats with advanced intelligence and machine learning capabilities. Automated policy creation and enforcement along with native integration to leading CI/CD tools provide security that enables innovation by not slowing development. Robust compliance checks and extensibility allow full control over your environment from developer workstations through to production. As the first end-to-end container security solution, Twistlock is purpose-built to deliver modern security.

# Deploying Kubernetes with Security and Compliance in Mind

PETROS KOUTOUPIS

## Introduction

Often, when exciting new technologies gain momentum, many are quickly abandoned or forgotten due to a lack of orchestration or centralized management platform. I speak of a method by which one can manage multiple instances of that new technology and across multiple nodes or racks in a data center. This is always the desired result. Kubernetes is one such management framework, but before diving

into this software platform, you need to understand the technology it is designed to manage: containers.

## Docker

It all began with Linux Containers (LXC). LXC is about as close to bare metal that one can get when running applications in a sandboxed environment. It imposes very little to no overhead when hosting self-contained instances by decoupling software applications or services (often referred to as microservices) from the operating system, giving users a clean and minimal Linux environment while running everything else in one or more isolated "containers". First introduced in 2008, LXC adopted much of its functionality from the Solaris Containers (or Solaris Zones) and FreeBSD

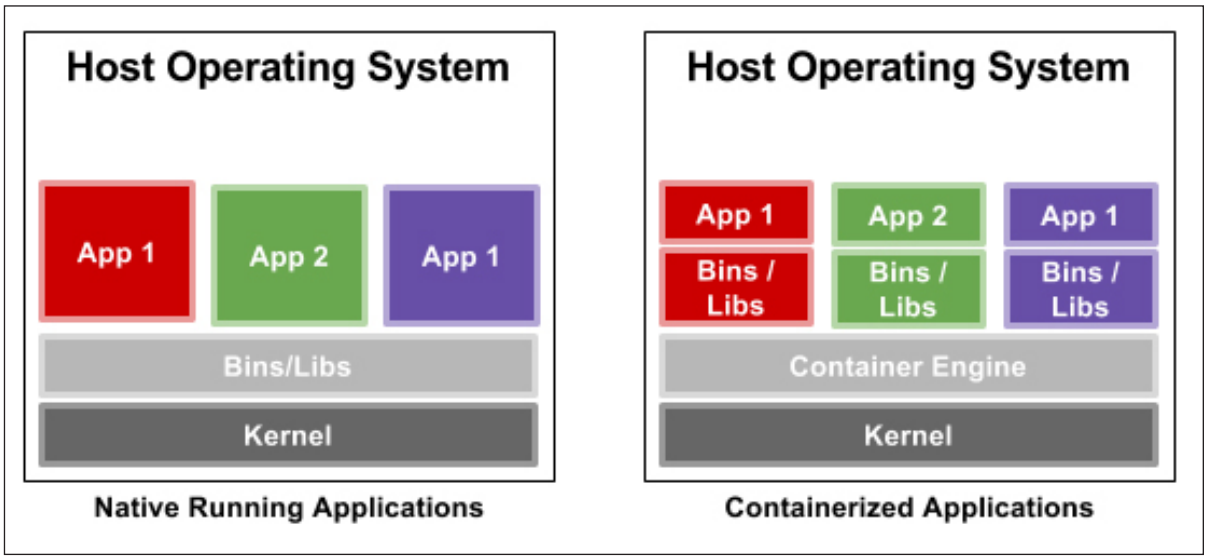


**FIGURE 1. A Comparison of Applications Running in a Traditional Environment to Containers**

jails that preceded it. Instead of creating a full-fledged virtual machine, LXC enables a virtual environment with its own process and network space. LXC makes use of namespaces to enforce process isolation. The kernel's very own cgroups (Control Groups) steps in to limit, account for and isolate the CPU, memory, disk I/O, network (and so on) usage of one or more processes. Think of this userspace framework as a very advanced form of chroot.

LXC brought with it some distinct advantages. For one, its method of isolation prevents processes running within a given container from monitoring or affecting processes running in another container. Second, these containerized services do not influence or disturb the host machine. This design added both security and stability to the Linux framework. As great as it was, LXC did come with a few limitations—limitations addressed by Docker.

Since its initial launch, Docker has taken the Linux computing world by storm. Docker is an Apache-licensed open-source containerization technology designed to automate the repetitive task of creating and deploying microservices inside containers. Docker treats containers as if they were extremely lightweight and modular virtual machines. Initially, Docker was built on top of LXC, but it has since moved away from that dependency, resulting in a better developer and user experience. Much like LXC, Docker continues to make use of the kernel cgroups subsystem. The technology is more than just running containers, it also eases the process of creating containers, building images, sharing those built images and versioning them.

Docker primarily focuses on the following:

- Portability: Docker provides an image-based deployment model. This type of portability allows for an easier way to share an application or set of services (with all of their dependencies) across multiple environments.

- Version control: a single Docker image is made up of a series of combined layers. A new layer is created whenever the image is altered. For instance, a new layer is created every time a user specifies a command, such as run or copy. Docker will reuse these layers for new container builds. Layering to Docker is its very own method of version control.

- Rollback: again, every Docker image has layers. If you do not wish to use the currently running layer, you can roll back to a previous version. This type of agility makes it easier for software developers to integrate and deploy their software technology continuously.

- Rapid deployment: provisioning new hardware often can take days. And, the amount of effort and overhead to get it installed and configured is quite burdensome. With Docker, you can avoid all of that by reducing the time it takes to get an image up and running to a matter of seconds. When you are done with a container, you can destroy it just as easily.

Fundamentally, both Docker and LXC are very similar.

They both are userspace and lightweight virtualization platforms that utilize cgroups and namespaces to manage resource isolation. However, there are a number of distinct differences between the two.

**Process Management** Docker restricts containers to run as a single process. If your application consists of X number of concurrent processes, Docker will want you to run X number of containers, each with its own distinct process. This is not the case with LXC, which runs a container with a conventional init process and, in turn, can host multiple processes inside that same container. For example, if you want to host a LAMP (Linux + Apache + MySQL + PHP) server, each process for each application will need to span across multiple Docker containers.

**State Management** Docker is designed to be stateless, meaning it does not support persistent storage. There are ways around this but, again, only necessarily when the process requires it. When a Docker image is created, it will consist of read-only layers. This will not change. During runtime, if the process of the container makes any changes to its internal state, a diff between the internal state and the current state of the image will be maintained until either a commit is made to the Docker image (creating a new layer) or until the container is deleted, resulting in causing that diff to disappear.

**Portability** This word tends to be overused when discussing Docker—that's because it is the single most important advantage Docker has over LXC. Docker does a much better job of abstracting away the networking, storage and operating system details from the application.

This results in a truly configuration-independent application, guaranteeing that the environment for the application always will remain the same, regardless of the machine on which it is enabled.

Docker is designed to benefit both developers and system administrators. It has made itself an integral part of many DevOps (developers + operations) toolchains. Developers can focus on writing code without having to worry about the system ultimately hosting it. With Docker, there is no need to install and configure complex databases or worry about switching between incompatible language toolchain versions. Docker gives the operations staff flexibility, often reducing the number of physical systems needed to host some of the smaller and more basic applications. Docker streamlines software delivery. New features and bug/security fixes reach the customer quickly without any hassle, surprises or downtime.

## Orchestration

On its own, Docker is extremely simple to use, and running a few images simultaneously is also just as easy. Now, scale that out to hundreds, if not thousands, of images. How do you manage that? Eventually, you need to step back and rely on one of the few orchestration frameworks specifically designed to handle this problem. This is where Kubernetes truly shines.

**Kubernetes** Kubernetes, or k8s (k + eight characters), originally was developed by Google. It is an open-source platform aiming to automate container operations: "deployment, scaling, and operations of application

Kubernetes makes it possible to respond to consumer demands quickly by deploying your applications within a timely manner, scaling those same applications with ease, and seamlessly rolling out new features, all while limiting hardware resource consumption.

containers across clusters of hosts". Google was an early adopter and contributor to the Linux Container technology. In fact, it is Linux Containers that power Google's very own Cloud services. Anyway, Kubernetes eliminates all of the manual processes involved in the deployment and scaling of containerized applications. It is capable of clustering together groups of servers hosting Linux Containers while also allowing the administrator to manage those clusters easily and efficiently.

Kubernetes makes it possible to respond to consumer demands quickly by deploying your applications within a timely manner, scaling those same applications with ease, and seamlessly rolling out new features, all while limiting hardware resource consumption. Kubernetes can be configured to manage and monitor on-premises and public/private/hybrid deployments. It is extremely modular and easily can be hooked into by other applications or frameworks. It also provides additional self-healing services, including auto-placement, auto-restart,

auto-replication and auto-restart of containers. The very best part of Kubernetes is that it supports Docker. Sure, other orchestration frameworks support Docker as well (for example, Swarm), but none are as flexible, extensible and widely adopted as Kubernetes.

**Architecture** Kubernetes runs on top of an operating system (for example, Ubuntu Server, Red Hat Enterprise Linux, SUSE Linux Enterprise Server and others) and takes a master-slave approach to its functionality. The *master* signifies the machine (physical or virtual) that controls the Kubernetes nodes. This is where all tasks originate. It *is* the main controlling unit of the cluster and will take

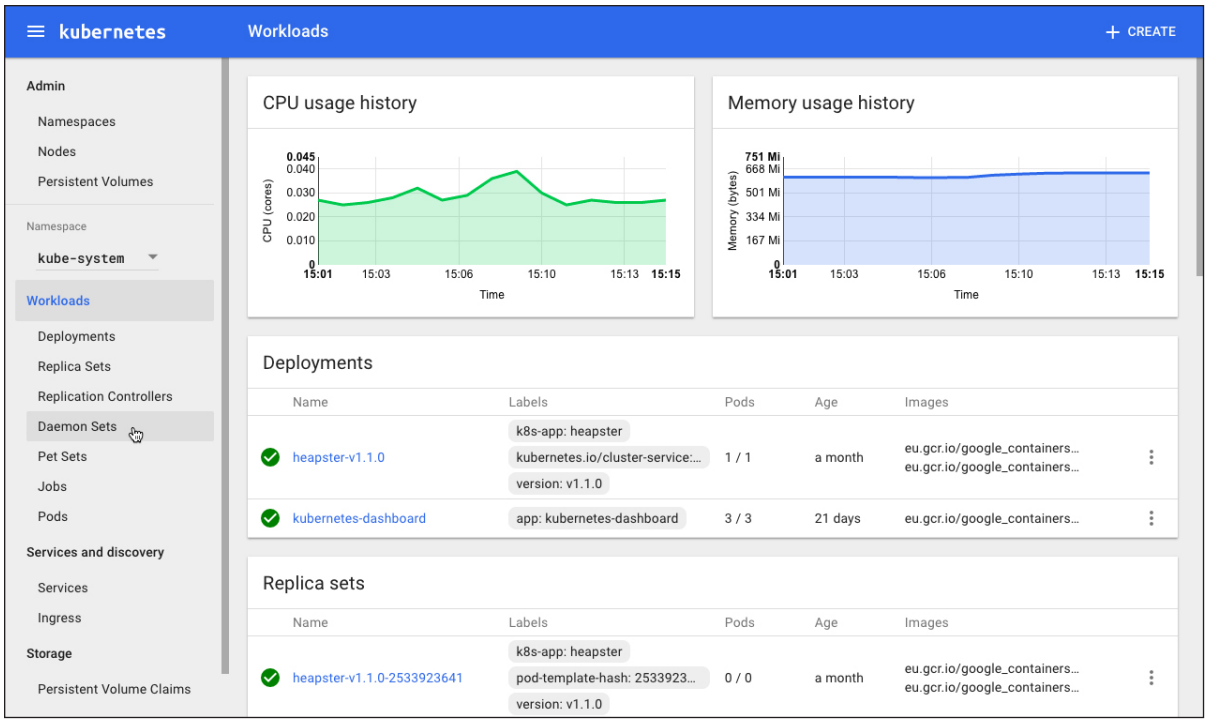


**FIGURE 2.** The Kubernetes Web UI Dashboard (from https://kubernetes.io)

the commands issued by an administrator or DevOps team and, in turn, relay it to the underlying nodes. The master node can be configured to run on a single machine or across multiple machines in a high-availability cluster. This is to ensure fault-tolerance of the cluster and reduce the likelihood of downtime. The *nodes* are the machines that perform the tasks assigned by the master. The node is sometimes referred to as the Worker or Minion.

Kubernetes is broken down into a set of components, some of which manage individual nodes while the rest are part of the control plane.

*Control plane management:*

- etcd — a lightweight and distributed cluster manager. It is persistent, and it reliably stores the configuration data of the cluster, providing a consistent and accurate representation of the cluster at any given point of time.

- API server — serves the Kubernetes API using JSON over HTTP. It provides both an internal and external interface to Kubernetes. The server processes and validates RESTful requests and enables communication between and across several tools and libraries.

- Scheduler — selects on which node an unscheduled pod should run. This logic is based on resource availability. The scheduler also tracks resource utilization of each node, ensuring that the assigned workload never exceeds what is available on the physical or virtual machine.

- Control Manager — is the process hosting the DaemonSet and Replication controllers. The controllers communicate with the API server to create, update or delete managed resources.

*Node management:*

- kubelet — is responsible for the running state of each node, making sure that all containers on the node are healthy. It handles the starting/stopping of application containers (see how this differs with Docker in the section below) within a Pod as directed by the manager in the control plane.

- kube-proxy — is a network proxy and load balancer. It is responsible for routing traffic to the appropriate container.

- cAdvisor — is an agent that monitors and collects system resource utilization and performance metrics (for example, CPU, memory, file and network) of each container on each node.

**Controllers** A controller drives the state of the cluster by managing a set of pods. There is the *Replication Controller* that handles pod replication and scaling by running a specified number of copies of a given pod across the entire cluster of nodes. It also can handle the creation of replacement pods in the event of a failing node. The *DaemonSet Controller* is in charge of running exactly one pod per node. The *Job Controller* runs pods to completion (as part of a batch job).

So, how does Docker fit into all of this? Docker still functions as it was meant to function.

**Services** In Kubernetes terms, a *service* consists of a set of pods working together (a one-tier or multi-tier application). As Kubernetes provides service discovery and request routing (by assigning the appropriate static networking parameters), it ensures that all service requests get to the right pod, regardless of where it moves across the cluster. Some of this movement may be a result of pod or node failure. In the end, Kubernetes' self-healing capabilities will get those ailing services back to a pristine state automatically.

**Pods** When a Kubernetes master deploys a group of one or more containers to a single node, it does so by creating a *pod*. Pods abstract the networking and storage from the container, and all of the containers within a pod will share the same IP address, hostname and more, allowing it to be moved around in the cluster without complication.

The kubelet will monitor each and every pod. If it is not in a good state, it will redeploy that pod to the same node. Apart from this, a heartbeat messaging mechanism will relay the node status to the master every few seconds. As soon as the master detects a node failure, the Replication Controller will launch the now affected pods onto another healthy node.

So, how does Docker fit into all of this? Docker still functions as it was meant to function. When a

Kubernetes master schedules a pod to a node, the kubelet running on that node will direct Docker in launching the desired containers. The kubelet will continue by monitoring those containers while also collecting information for the master. Docker still will be in full control of the containers running on the node and also will be responsible for starting and stopping them. The only difference here is that you now have an automated system sending these requests to Docker instead of the systems administrator running the same tasks manually.

So, now that you have a good idea of how Kubernetes works, what comes next? The answer: *security and compliance*.

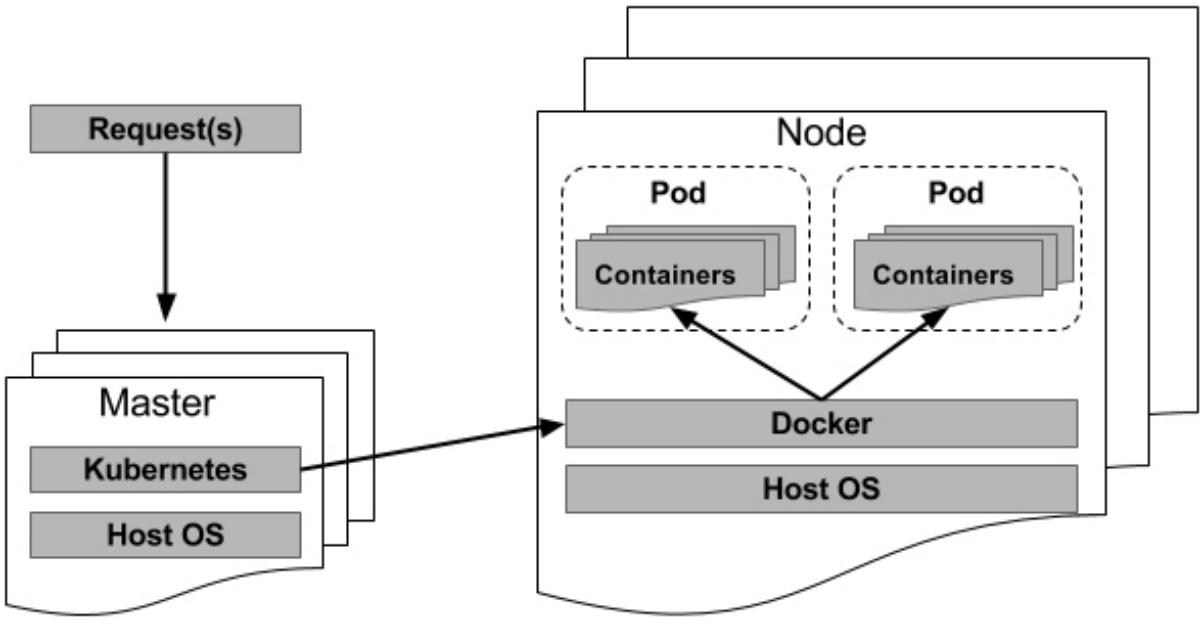


**FIGURE 3.** A General Model of Pod Creation/Management

## Finding the Missing Pieces to the Puzzle

Twistlock develops and distributes a product of the same name focusing on nothing but Docker image security and compliance. The company is committed to providing enterprise security with DevOps agility. Twistlock also can be operated from and managed by orchestration platforms including Kubernetes.

The company offers the first end-to-end security solution built for containerized environments. It protects against software exploits, malware and active threats through its advanced intelligence and machine-learning capabilities. It automatically will profile expected container behavior, and create and enforce security models at runtime. It also will automatically build security models of expected behavior and enforces these via whitelisting. Because it is automated, security can be introduced much earlier in the lifecycle to identify and block threats from developer workstations through to production. And here's the best part: when deployed as part of a Kubernetes cluster, that same security model will scale across all nodes in the cluster.

Twistlock sources more than 30 vulnerability and threat intelligence feeds, combining it with its proprietary research. This ensures that Twistlock's customers are kept updated, in real time, on all known application CVEs (Common Vulnerabilities and Exposures), exploits and threats.

## The Many Benefits of Using Twistlock

Twistlock is a tool to both harden your images in

development and protect them against runtime threats. Twistlock is built as a Docker image and runs as a privileged container image on top of the Docker Engine. The idea is to run a single instance of this Twistlock image on every physical or virtual machine hosting Docker containers.

Each instance of Twistlock can be managed from the Twistlock Console. Through this very same console, you can create/remove security policies, establish image compliance and also monitor the security state of each running container. If that container surpasses the defined threshold of vulnerabilities or does not comply to the parameters you have set, Twistlock either will prevent that container from running or disconnect it completely from the network.

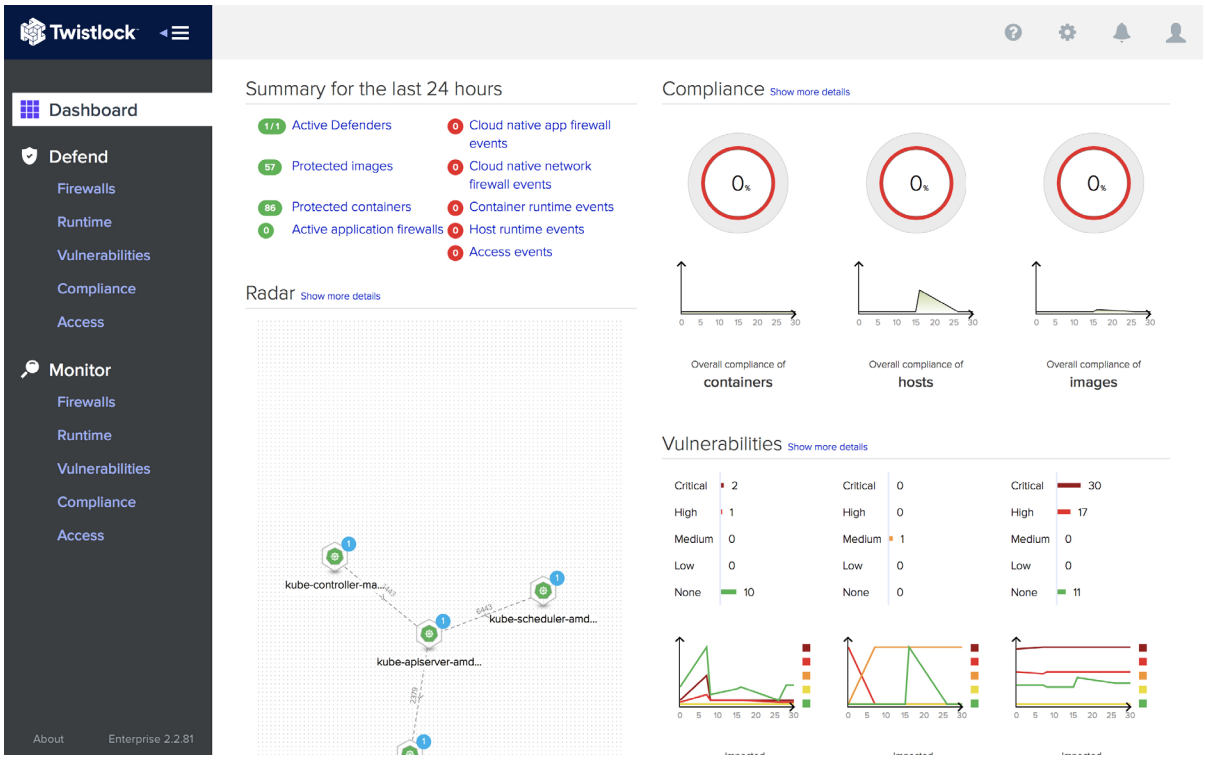


**FIGURE 4.** The Twistlock Management Console

**Runtime Protection** Twistlock runtime defense protects your containers against detected exploits, compromises, application flaws and configuration errors. It actively monitors container activities and detects policy violations. Twistlock will report all anomalous behaviors while also taking the appropriate actions to disconnect or isolate them, preventing disruption to any and all other containers across the Kubernetes cluster. Twistlock can identify when a container does something that it shouldn't be doing. For instance, if a container running nginx suddenly invokes netstat and netstat isn't a whitelisted process for that image, Twistlock will detect it.

**Vulnerability Management** Twistlock is constantly scanning container images in registries, workstations and servers for known vulnerabilities and misconfigurations. All detected vulnerabilities are reported and extend across Linux distributions (Debian, Ubuntu, Fedora), application frameworks (Node.js, Python, Java) and even your custom application packages. Twistlock breaks the Docker image apart and parses each individual layer, specifically searching for these threats. Twistlock can and will take remediation actions based on the severity of the vulnerability during runtime. Twistlock provides users with granular control when managing the types of vulnerabilities beyond their severity ratings. You can block individual CVEs explicitly while ignoring others.

**Continuous Integration** Twistlock was written to integrate directly into your Continuous Integration (CI) process (such as Jenkins). This way, it can find and report problems before they ever make it out into production.

Twistlock has developed 150+ built-in checks to validate the recommended practices from this benchmark.

In some cases, when a package with an open CVE is reported, Twistlock also will report the package version that has the fix. Developers are given clear insight into the vulnerabilities present in every build. These plugins allow you to define and enforce your vulnerability policies at build time. For instance, you can set a policy requiring that one build job must not have any vulnerability, or you can flag specific CVEs while ignoring the rest.

**Compliance** The Center for Internet Security (CIS) Docker and CIS Kubernetes Benchmarks provide guidance for establishing a secure configuration of a Docker container. In short, this benchmark provides the best security practices for deploying Docker. Twistlock has developed 150+ built-in checks to validate the recommended practices from this benchmark. In parallel to this, Twistlock includes an extensive list of configuration checks for the host machine, Docker dæmon, Docker files and directories. Organizations using Twistlock will be able to enforce Trusted Registries (containing images approved by Twistlock) and Trusted Images. When configured, Twistlock can enforce that the images from these trusted lists are the only ones deployed onto production servers.

**Access Control** Using Twistlock, you can define and enforce policies governing user access to both Docker and Kubernetes resources, limiting specific users to individual functions or APIs. Out of the box, Twistlock supports enterprise identity directories that include Active Directory, OpenLDAP and SAML providers. This way, you can specify access policies to container resources without the need to create new identities and groups. You can monitor detailed user access audit trails, action types, services requested and more from the console.

**Analytics** Twistlock's built-in analytics allow you to visualize all relevant data and enable you to enforce standard configurations, container best practices and recommend deployment templates. This way, your containers will remain compliant to industry or company policies.

## Summary

More often than not, production applications will span across multiple containers, and those containers may be deployed across multiple physical server machines. Kubernetes gives you the orchestration and management capabilities required to deploy and scale those containers to accommodate the always changing workload requirements. Kubernetes also provides the proper facilities to deploy Twistlock for a more stable and secure containerized operating environment.■