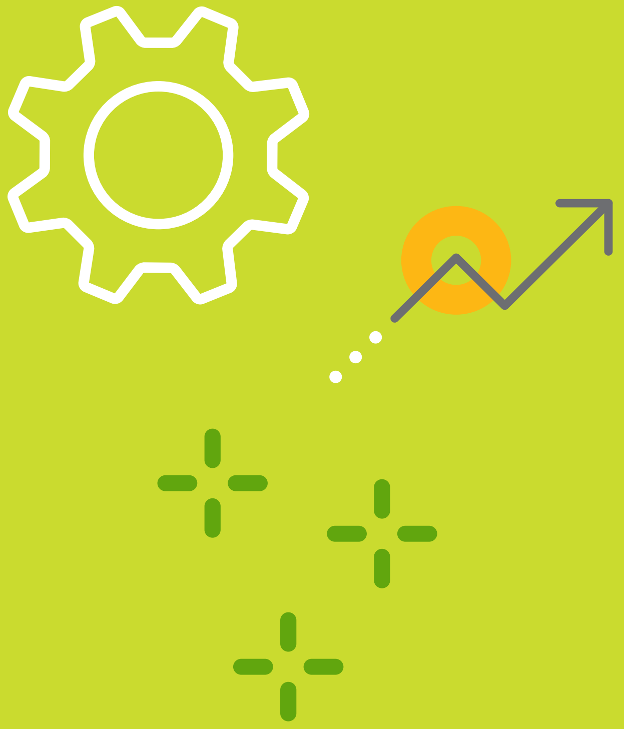
A BEGINNER'S GUIDE TO KUBERNETES MONITORING

splunk>



A BEGINNER'S GUIDE TO KUBERNETES MONITORING

Since the introduction of the concept in 2013, containers have become the buzz of the IT world. It's easy to see why: Application container technology is revolutionizing app development, bringing previously unimagined flexibility and efficiency to the development process.

Businesses are embracing containers in droves. According to **Gartner**, more than half of global enterprises will be running containerized applications in production by 2020, up from less than 20 percent today. And **IDC predicts** by 2021, more than 95 percent of new microservices will be deployed in containers. That mass adoption makes it clear that organizations need to adopt a container-based development approach to stay competitive.

To that end, let's look at what's involved with containerization and how your organization can leverage it to gain an edge.

WHAT IS A CONTAINER?

The easiest way to understand the concept of a container is to consider its namesake. A physical container is a receptacle used to hold and transport goods from one location to another.

A software container performs a similar function. It allows you to package up an application's code, configuration files, libraries, system tools, and everything else needed to execute that app into a self-contained unit so you can move and run it anywhere.

Further, containers enable a "microservices" approach that breaks applications down into single-function modules that are accessed only when they're needed. This allows a developer to modify and redeploy a particular service rather than the whole application whenever changes are required.

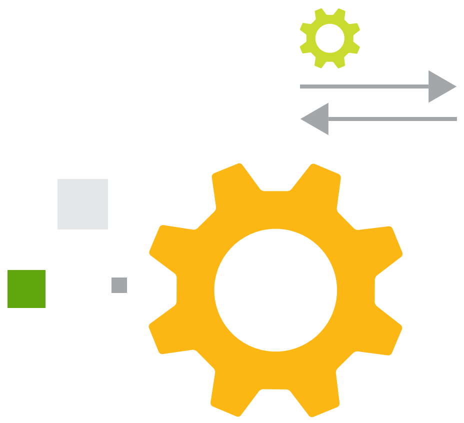
WHY ARE CONTAINERS SUCH A BIG DEAL?

Containers remedy an all-too-common problem in IT Operations: getting software to run reliably and uniformly no matter where it is deployed. As an app is moved from one computing environment to another — from staging to production, for example — it can run into problems if the operating system, network topology, security policies or other aspects of the environment are different. Containers isolate the app from its environment, abstracting away these environmental differences.

Prior to containers, virtual machines (VMs) were the primary method for running many isolated applications on a single server. Like containers, VMs abstract away a machine's underlying infrastructure so that hardware and software changes won't affect app performance. But there are significant differences to how each does this.

A VM abstracts hardware to turn a physical server into several virtual ones. It does so by running on top of a hypervisor, which itself runs on a physical computer called the "host machine." The hypervisor is essentially a coordination system that emulates the host machine's resources —CPU, RAM, etc.— making them available to the VM or "guest machine." The apps and everything required to run them, including libraries and system binaries, are contained in the guest machine. Each guest machine also includes a complete operating system of its own. So a server running four VMs, for example, would have four operating systems in addition to the hypervisor coordinating them all. That's a lot of demand on a one machine's resources, and things can bog down in a hurry, ultimately limiting how many VMs a single server can operate.

Containers, on the other hand, abstract at the operating system level. A single host operating system runs on the host (this can be a physical server, VM, or cloud host), and the containers — using a containerization engine like the Docker Engine — share that OS's kernel with other containers, each with its own isolated user space. There's much less overhead here than with a virtual machine, and as a result, containers are far more lightweight and resource efficient than VMs, allowing for much greater utilization of server resources.



5 BENEFITS OF DEPLOYING CONTAINERS

A container-based infrastructure offers a host of benefits. Here are the five biggest.

1. **Speed of delivery**—Applications installed on a virtual machine typically take several minutes to launch. Containers don't have to wait for an operating system boot, so they start up in a fraction of a second. They also run faster since they use fewer host OS resources, and they only take a few seconds to create, clone or destroy. All of this has a dramatic impact on the development process, allowing organizations to more quickly get software to market, fix bugs and add new features.

2. **DevOps first**—Containers' speed, small footprint, and resource efficiency make them ideal for a DevOps environment. A container-based infrastructure enables developers to work as quickly and efficiently as possible on their preferred platform without having to waste time on non-business-critical tasks.

3. **Accessibility—**As mentioned earlier, containers pack up only the app and its dependencies. That makes it easy to move and reliably run containers on Windows, Linux or Mac hardware. Containers can run on bare metal or on virtual servers, and within public or private clouds. This also helps avoid vendor lock-in should you need to move your apps from one public cloud environment to another.

4. **Increased scalability**—Containers tend to be small because they don't require a separate OS the way that VMs do. One container is typically sized on the order of tens of megabytes, whereas a single VM can be tens of gigabytes — roughly 1,000 times the size of a container. That efficiency allows you to run many more containers on a single host operating system, increasing scalability.

5. **Consistency**—Because containers retain all dependencies and configurations internally, they ensure developers are able to work in a consistent environment regardless of where the containers are deployed. That means developers won't have to waste time troubleshooting environmental differences and can focus on addressing new app functionality. It also means you can take the same container from development to production when it's time to go live.

KUBERNETES AND CONTAINERS 101

To get started with container orchestration, you need specialized software to deploy, manage and scale containerized applications. One of the earliest and most popular choices today is Kubernetes, an open-source automation platform developed by Google and now managed by the Cloud Native Computing Foundation.

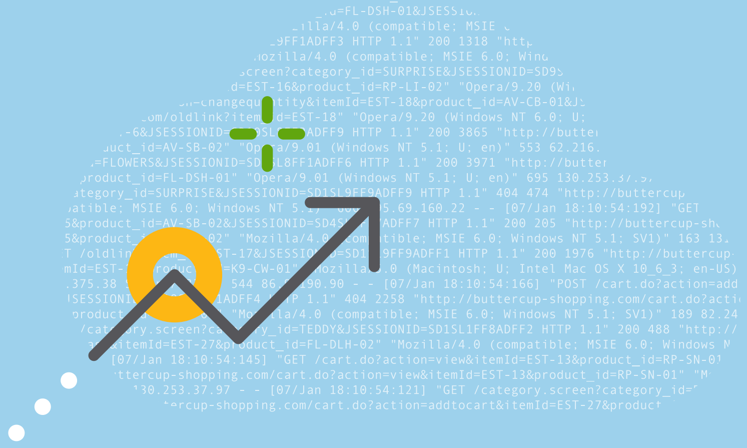
Kubernetes can dramatically enhance the development process by simplifying container management, automating updates, and minimizing downtime so developers can focus on improving and adding new features to applications. To better understand how, let's look at Kubernetes' basic components and how they work together.

Kubernetes uses multiple layers of abstraction defined within its own unique language. There are many parts to Kubernetes. This list isn't exhaustive, but it provides a simplified look at how hardware and software is represented in the system.

Nodes: In Kubernetes lingo, any single "worker machine" is a node. It can be a physical server or virtual machine on a cloud provider such as AWS or Microsoft Azure. Nodes were originally called "minions," which gives you an idea of their purpose. They receive and perform tasks assigned from the Master Node and contain all the services required to manage and assign resources to containers.

Master Node: This is the machine that orchestrates all the worker nodes and is your point of interaction with Kubernetes. All assigned tasks originate here.

Cluster: A cluster represents a master node and several worker nodes. Clusters consolidate all of these machines into a single, powerful unit. Containerized applications are deployed to a cluster, and the cluster distributes the workload to various nodes, shifting work around as nodes are added or removed.



Pods: A pod represents a collection of containers packaged together and deployed to a node. All containers within a pod share a local network and other resources. They can talk to each other as if they were on the same machine, but they remain isolated from one another. At the same time, pods isolate network and storage away from the underlying container.

A single worker node can contain multiple pods. If a node goes down, Kubernetes can deploy a replacement pod to a functioning node.

Despite a pod being able to hold many containers, it's recommended they wrap up only as many as needed: a main process and its helper containers, which are called "sidecars." Pods scale as a unit no matter what their individual needs are and overstuffed pods can be a drain on resources.

Deployments: Instead of directly deploying pods to a cluster, Kubernetes uses an additional abstraction layer called a "deployment." A deployment enables you to designate how many replicas of a pod you want running simultaneously. Once it deploys that number of pods to a cluster, it will continue to monitor them and automatically recreate and redeploy a pod if it fails.

Ingress: Kubernetes isolates pods from the outside world, so you need to open a communication channel to any service you want to expose. This is another abstraction layer called Ingress. There are a few ways to add ingress to a cluster, including adding a LoadBalancer, NodePort or Ingress controller.

WHAT DOES A TYPICAL KUBERNETES DEPLOYMENT LOOK LIKE?

Being familiar with the fundamental components of Kubernetes should give you some idea of how the system approaches container orchestration, but a deeper understanding requires visualizing each of these components in action. Here we will look at how to use Kubernetes to deploy an app to a cluster on Google Cloud using an example put together by Google Engineer Daniel Sanche.

Sanche's tutorial uses Gitea, an open-source git hosting service, as the deployed app, but as he notes this walkthrough could be employed using virtually any app.

Create a cluster

Two commands are critical when setting up a Kubernetes environment: *kubectl* and *gcloud*. *kubectl* is the primary tool for interacting with the Kubernetes API and is used to create and manage software resources such as pods and deployments. However, because Kubernetes is platform-agnostic, the *kubectl* command can't provision nodes directly from your chosen cloud provider, so a third-party tool is needed. If Google Cloud is your provider, for example, you could use Google Kubernetes Engine's gcloud command to provision your nodes.

Once you've set up your Kubernetes environment, these commands are used to create a default cluster of three nodes:

```
gcloud container clusters create my-cluster --zone us-
west1-a
gcloud container clusters get-credentials my-cluster \
--zone us-west1-a
```

Your cluster will now be visible within the GKE section of the Google Cloud Console. The VMs you've provisioned as your nodes will appear in the GCE section.

Deploy an app

Now you can start assigning resources to your live cluster. Though you can do this interactively with the *kubectl add command*, Sanche recommends doing it by writing all of your Kubernetes resources in YAML files. This allows you to record the entire state of your cluster in easily maintainable files with all the instructions required to host your service saved alongside the actual code, which makes for simpler management.

To add a pod to your cluster using a YAML file, create a file called gitea.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
name: gitea-pod
spec:
containers:
       - name: gitea-container
       image: gitea/gitea:1.4
```

This code declares that you're creating a pod — named "gitea-pod" — defined in v1 of the Kubernetes API. It contains one container called "gitea-container." The final line defines which container image you want to run. Here the image is the one tagged as 1.4 in the gitea/gitea repository. Kubernetes tells the built-in container runtime to locate this container image and add it to the pod.

Next, apply the YAML file to the cluster by executing this command: *kubectl apply -f gitea.yaml*.

Kubernetes will read the file and add the pod to the cluster. You can see the new pod by running the *kubectl* get pods command. This will return data about the status of the pod, whether it has restarted, and how long it has been running.

You can also view the container's standard output by running the command *kubectl logs -f gitea-pod*, which will return something like this:

```
Generating /data/ssh/ssh_host_ed25519_key...
Feb 13 21:22:00 syslogd started: BusyBox v1.27.2
Generating /data/ssh/ssh_host_rsa_key...
Generating /data/ssh/ssh_host_dsa_key...
Generating /data/ssh/ssh_host_ecdsa_key...
/etc/ssh/sshd_config line 32: Deprecated option
UsePrivilegeSeparation
Feb 13 21:22:01 sshd[12]: Server listening on :: port 22.
Feb 13 21:22:01 sshd[12]: Server listening on 0.0.0.0 port 22.
2018/02/13 21:22:01 [T] AppPath: /app/gitea/gitea
2018/02/13 21:22:01 [T] AppWorkPath: /app/gitea
2018/02/13 21:22:01 [T] Custom path: /data/gitea
2018/02/13 21:22:01 [T] Log path: /data/gitea/log
2018/02/13 21:22:01 [I] Gitea v1.4.0+rc1-1-gf61ef28 built
with: bindata, sqlite
2018/02/13 21:22:01 [I] Log Mode: Console(Info)
2018/02/13 21:22:01 [I] XORM Log Mode: Console(Info)
2018/02/13 21:22:01 [I] Cache Service Enabled
2018/02/13 21:22:01 [I] Session Service Enabled
2018/02/13 21:22:01 [I] SQLite3 Supported
2018/02/13 21:22:01 [I] Run Mode: Development
2018/02/13 21:22:01 Serving [::]:3000 with pid 14
2018/02/13 21:22:01 [I] Listen: http://0.0.0.0:3000
```

Deployment

As discussed earlier, it's not typical to deploy pods directly in Kubernetes but rather to use the Deployment abstraction layer instead. To do this, you'll need to take a step back and delete the pod previously created with the *kubectl delete -f gitea.yaml* command so you can recreate it through the Deployment layer.

Next, go back to the YAML file you originally created and alter as shown below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
name: gitea-deployment
spec:
replicas: 1
selector:
matchLabels:
app: gitea
template:
metadata:
labels:
app: gitea
spec:
containers:
name: gitea-container
image: gitea/gitea:1.4
```

The first nine lines of this code define the deployment itself and the rest define the template of the pod the deployment will manage. Line 6 (replicas) is the most critical information, as it tells Kubernetes how many pods you want to run.

Now you can apply the modified YAML file with the command *kubectl apply -f gitea.yaml*.

Type *kubectl get pods* again to see the running pod(s). To verify the deployment information, enter *kubectl get deployments*.

One of the advantages of Kubernetes deployments is that if a pod goes down or is deleted, it will automatically be redeployed. To see that in action, delete the pod you just deployed by typing *kubectl delete pod <podname>* and you should see a new one added to your cluster.

For more information on deploying an app, adding ingress so you can access it through a browser, and more, take a look at Sanche's complete tutorial.

HOW DO YOU MONITOR KUBERNETES EFFECTIVELY?

For all the benefits that containers bring to IT organizations, they can also make cloud-based application management more complex. Some of the challenges they present include:

- **Significant Blind Spots**—Containers are designed to be disposable. Because of this, they introduce several layers of abstraction between the application and the underlying hardware to ensure portability and scalability. This all contributes to a significant blind spot when it comes to conventional monitoring.
- **Increased Need to Record**—The easy portability of so many interdependent components creates an increased need to maintain telemetry data to ensure observability into the performance and reliability of the application, container and orchestration platform.
- **The Importance of Visualizations**—The scale and complexity introduced by containers and container orchestration requires the ability to both visualize the environment to gain immediate insight into your infrastructure health but also be able to zoom in and view the health and performance of containers, node and pods. The right monitoring solution should provide this workflow.
- **Don't Leave DevOps in the Dark**—Containers can be scaled and modified with lightning speed. This accelerated deployment pace makes it more challenging for DevOps teams to track how application performance is impacted across deployments.

A good container monitoring solution will enable you to stay on top of your dynamic container-based environment by unifying container data with other infrastructure data to provide better contextualization and root cause analysis. Let's look at how one could provide several layers of monitoring for Docker, the most popular container implementation:

Hosts: The physical and virtual machines in your clusters can be monitored for availability and performance. Key metrics to track include memory, CPU usage, swap space used and storage utilization. This should be a core capability of any container monitoring tool.

Containers: Visibility into your containers in aggregate and individually is critical. A monitoring tool can provide information on the number of currently running containers, the containers using the most memory and the most recently started container. It can also provide insight into each container CPU and memory utilization and the health of its network I/O.

Application endpoints: In a typical container-based environment, each application service will be running on one or more containers. Ideally, application monitoring should be performed at the level of the container, pod and whole system.

GETTING STARTED.

Containers are a powerful tool in your development arsenal, but it's critical to understand how and how well your container environments are working. For more information, visit us online to see how we can help you get started with containers, orchestration and monitoring.

splunk>