The Expert's Guide to Running Containers in Production

portworx

INTRODUCTION

The benefits of container-based microservices 3
- Agility 3
- Resilience 5

The challenge of stateful containers 5
- Persistence 5
- High availability 5
- Security 6
- Scheduler-based automation 6
- Any database, any infrastructure 6

WORKING WITH STATEFUL CONTAINERS USING THE LEADING CONTAINER MANAGERS

Docker 7
- Data volumes 7
- Data volume plugins 8
- Docker Swarm 9

Kubernetes 9
- Volumes 9
- Persistent volumes 9
- Persistent volume claims 9
- Storage class 10
- Stateful sets 10
- Kubernetes volume plugins 11

Mesos 12
- Using Local Volumes 12
- External volumes 12

DATA ARCHITECTURES FOR CONTAINERIZED APPLICATIONS

Connector-based systems 15
- Low density of stateful containers per host 15
- Slow block device mounts 15

Key-value based systems 16

Container data services platform 16

COMMON USE CASES FOR STATEFUL CONTAINERS

Reliably deploy and operate containerized databases, queues, and key-value stores 17

Speeding up your CI/CD pipeline 18

Containerize your data processing workloads 19

Containerize your CMS, simplify management 20

SOURCES 20

APPENDIX

Storage drivers 21

INTRODUCTION:

The benefits and challenges of containerized apps

Thanks to Docker, containers have exploded onto the enterprise software scene like nothing else in the last decade.

Containers provide a process-runtime and application-packaging format, delivering concrete benefits in terms of speed-to-launch, density, and portability. But it is really the combination of containers and microservices that yields the biggest benefits. These outsized benefits explain why enterprises, often conservative when it comes to technologies that require large-scale platform refactoring, are leading the charge to adopt containers, with well-known companies such as GE and Capital One talking publicly about their move to container-based architectures.

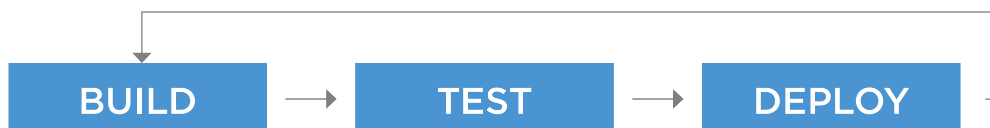
The benefits of container-based microservices

Container-based microservices provide two main benefits to enterprises: improved agility and application resilience. Let's look at each:

Agility

The big don't eat the small anymore; the fast eat the slow. Every major industry is in the process of being transformed by software-powered innovation. If a team or company can speed up the build-test-deploy cycle more than its competitors, it can capture a larger share of the market by responding better to changing conditions.

Speeding up this cycle makes you more competitive



Container-based microservices improve agility by breaking an application into multiple smaller parts that can be independently built, tested, and deployed without affecting any other part of the application.

Container-based microservices stand in contrast to what are often called "monolithic" applications. Often, in a monolithic application, improvement to one part of the code requires changes to another. This "tight coupling" of features into a single codebase leads to infrequent and high-risk software releases. When new versions of software are released only quarterly or annually, enterprises expose themselves to disruption by more nimble competitors.

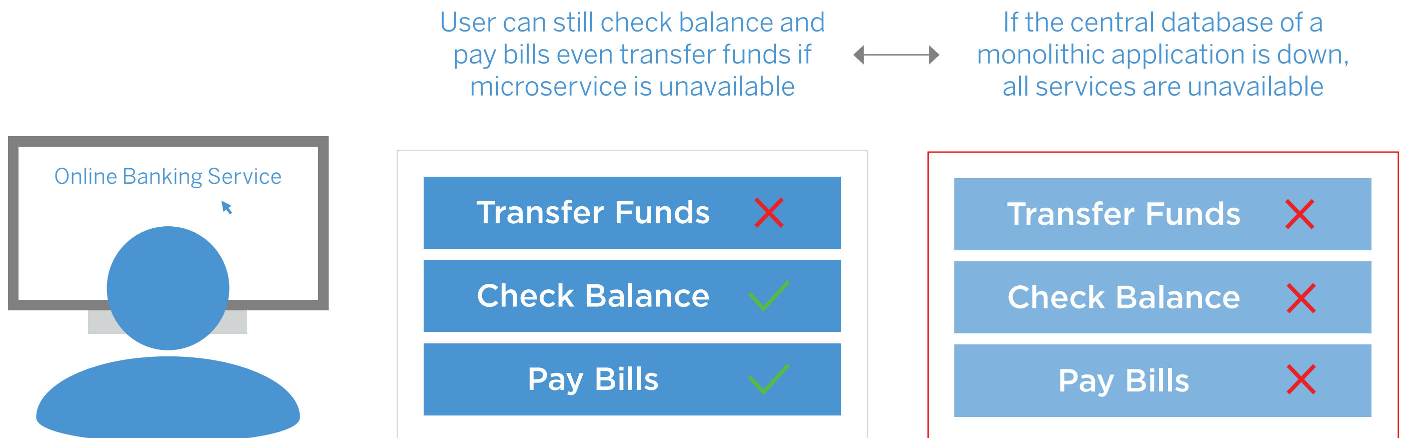
The other way that container-based microservices improve agility is by putting a premium on automation. If something can be automated, it can be done faster and more frequently without increasing the risk of human error. It also leaves the humans to concentrate on automating additional tasks that are currently manual and error-prone.

Resilience

Container-based microservices are faster to build, test, and deploy, but are they of higher quality? The evidence from enterprises would suggest yes.

The reason is that because microservices are "loosely coupled," a failure in one part of the system is less likely to affect another. For example, if an online banking service built using microservices is having a problem with its 'transfer funds' function, a user can still check account balances or pay bills online because each of the individual features is its own microservice, complete with its own database. While the user might experience a degraded experience, the service is still useful as a composition of the functioning parts.

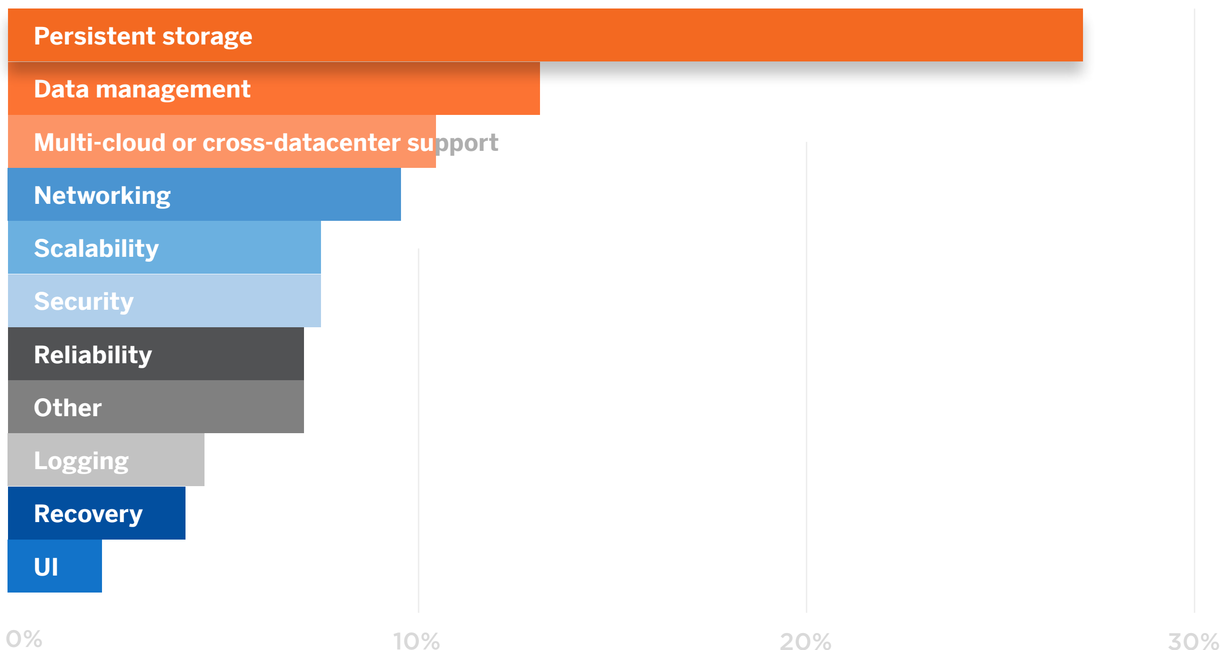
Contrast this microservices-based resilience with a monolithic banking application, where the inability to access a single Oracle database causes the user to be unable to check account balances, transfer funds, or pay a bill.



The challenge of stateful containers

The benefits of container-based microservices are clear. However, innovative enterprises that have gone down the microservices path have run into problems converting their databases, queues, and key-value stores to run inside containers. In fact, a recent industry survey by Portworx indicates that persistent storage was the number one barrier to production deployments of containers, as seen by the question and responses below:

In order to deploy containers, which challenge has been the most difficult to overcome?



Source: Portworx Container Adoption Survey 2017

Persistence

The first problem with stateful containers is that native Docker doesn't provide a persistence layer that enables a containerized database to survive host failure. You can mount a volume onto the host, which means if your container crashes, your data survives because it exists outside of the container. However, because the volume is host-bound, if you lose your host, you also lose your data. Additionally, if your container is rescheduled to a new host, its data does not move with it.

High availability

While HA has always been a requirement for certain databases, it is increasingly becoming a requirement for all. It used to be common for Ops teams to issue "Maintenance Notices" to their internal and external customers indicating that a service would be unavailable for a time during a migration or update. While

disruptive when software was released quarterly or annually, these maintenance windows are completely unacceptable in a world where applications are being updated on a daily or weekly basis. As a result, databases must be available even during updates, deployments, and a variety of host, network, and disk failures, as well as during high-traffic events.

Traditional solutions for HA, however, are less applicable for modern container-based microservices. Mitigating failure with custom hardware is expensive (not to mention anti-DevOps), while application-layer replication can be slow and can tank your performance—especially as the cluster rebuilds itself.

The problems of persistence cited above compound the issues related to ensuring HA for stateful containers.

Common failure modes:

- Server
- Disk
- Network
- Bug
- Traffic spike
- Load balancer
- The list goes on...

Security

Security is important for all parts of an application, but it is of particular importance for stateful containers. That's due to the sensitivity around data, especially for regulated industries. Enterprises moving to stateful containers need to explore two important areas: encryption and access controls.

For encryption, data in motion must be encrypted using SSL or other protocols, and data at rest must be encrypted with a key that only the customer retains access to.

Just as important, access controls must be in place to ensure that only containers with sufficient privileges can access certain data volumes.

Scheduler-based automation

The next challenge for stateful containers is scheduler-based automation of data services. While you could theoretically create a system to persist and secure your data using custom hardware solutions or highly controlled manual processes, these practices would offset the agility gains provided by microservices.

To be agile, DevOps teams can't wait days or weeks for storage to be provisioned so they can deploy their application. They need stateful containers to be as easy to deploy and manage as the stateless parts of their app. That means provisioning and managing data via a container manager, also known as a scheduler or orchestration framework. The problem is that most popular schedulers vary in their support for stateful containers.

Any database, any infrastructure

Finally, not only must enterprise teams solves persistence, HA, security, and data automation for one database running in one environment, but they must also do it for many databases in many environments.

Gone are the days of running everything on an Oracle database managed by a dedicated team of DBAs. Now, DevOps teams must consistently manage a wide mix of SQL and NoSQL databases as well as other stateful services such as Jenkins, Gitlab, or WordPress. Additionally, they need to be able to do it in many different environments: AWS, Azure, Google, VMware, and bare metal.

WORKING WITH STATEFUL CONTAINERS USING THE LEADING CONTAINER MANAGERS

The problems associated with persistence—HA, security, data automation, and support for heterogeneous databases and infrastructure—are solvable. The rest of this guide will explore how to address these issues while using the most popular software platforms for running containers: Docker, Kubernetes, and Mesosphere DC/OS.

Docker

Docker is the king of containers, and many of its key concepts carry through to the other schedulers. So starting with Docker makes the most sense.

Data volumes

The most fundamental stateful container concept in Docker is the data volume, which presents a formatted file system as a directory that can be mapped onto any path inside a container. Reads and writes to these data volumes operate on the host itself. The underlying file system will write to a block device, either local storage or some type of shared storage such as Amazon EBS or Google Persistence Disk.

In this command, a host `directory` `/src/webapp` is mounted into the container at `/webapp`.

```
$ docker run -d -P --name web -v /src/webapp:/webapp
training/webapp python app.py
```

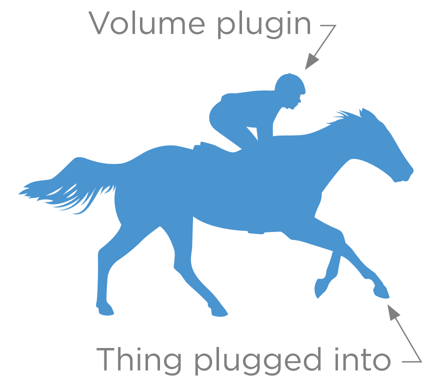
The most important feature of a data volume is that it persists beyond the life of the container, thus creating the most basic persistence for that container—i.e., if it restarts, the data would still be there. However, host data volumes, meaning volumes on the same host as a container, are not resilient to whole host or disk failure. As a result, Docker introduced the concept of volume plugins as a way to manage external host volumes and, in many cases, connect them to external shared storage.

Data volume plugins

A data volume provides the container with an agnostic directory it can read and write to—without having knowledge of the infrastructure required to provision the underlying storage. It is the job of a data volume plugin to do the provisioning and attach the underlying storage to the correct node, before the container starts.

It is this decoupling that is particularly useful if you are running large numbers of stateful containers across different environments.

Essentially, a Docker volume plugin acts as a control plane mechanism for the data volume. Sometimes, this is referred to as storage orchestration.



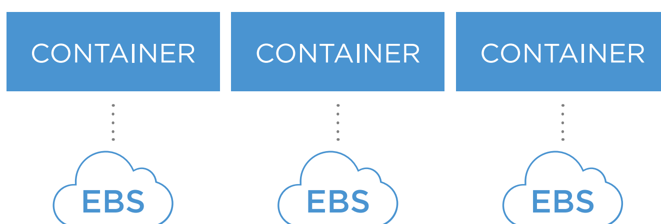
The horse and rider metaphor was originally used by Lew Moorman to describe an API and the system it connects to, but it works just as well for volume plugins and storage.

Each volume plugin can take a different approach to what type of storage it offers and, more importantly, how it orchestrates that storage in a busy container scheduler environment.

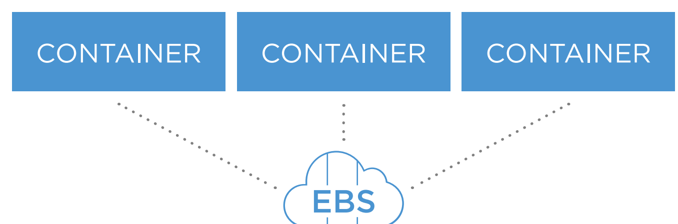
For example: If running containers on a common cloud platform such as AWS, we might choose to attach Elastic Block Storage (EBS) devices to our containers using a volume plugin. One plugin might choose to attach a new EBS device for each container, while another might pool each EBS device into a storage fabric and offer a virtual slice to each container.

In the example above, your application performance is going to differ dramatically in the two instances because mounting EBS volumes to EC2 hosts is a notoriously slow process, while formatting a virtual volume on top of a block device is nearly instantaneous.

The important point is that you need to understand what capabilities are provided by your underlying storage system, not just what the capabilities are provided by the volume plugin.



One block device per container has slow performance due to costly mount/unmount operations.



Slicing one block device into many virtual volumes provides instant volume provisioning.

Another way to think about this is that before you pick a volume plugin, you need to think about the operational characteristics that you want your Docker storage to support, then pick a volume plugin that supports it.

Docker Swarm

Docker provides a container scheduler called Swarm that can be used to automatically schedule containers in a cluster. When combined with a Docker volume plugin, Docker Swarm can be used to provide persistence, HA, and scheduler-based automation for stateful containers—subject to the caveats above about the limitations of some Docker volume plugins.

Kubernetes

One of the most popular container schedulers is Kubernetes. While many of the storage concepts are the same as for Swarm, Kubernetes goes beyond the capabilities of Docker to provide more sophisticated volume management primitives.

Volumes

As with Docker generally, on-disk files in a container are ephemeral. If your database container crashes, you don't want to lose your data so you don't want to store your data in a container.

Volumes in Kubernetes, as in Docker, provide some level of resilience against data loss. In Kubernetes, a volume persists after a container crashes. However, it doesn't persist after your pod crashes; the life of the volume is the same as the life of the pod. (In Kubernetes, a pod is the basic building block, consisting of one or more containers, resources such as volumes, and configuration options). If you need more persistence than is provided by a volume, you can use the aptly named persistent volume, as described below.

Persistent volumes

A `PersistentVolume` or (PV) is a volume that will last longer than the lifecycle of a single Kubernetes pod and represents a piece of networked storage in the cluster. For example, if you are running your Kubernetes pods on AWS, you could have a persistent volume backed by an EBS device. This PV is a resource in the cluster just like a physical server is a cluster resource.

Persistent volume claims

To use a PV in Kubernetes, you need create a `Persistent Volume Claim`. Remember, a PV is just a resource in the cluster. A persistent volume claim is a request for those resources. In the diagram below, we see a PV claim for a 3 gig read-write volume.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Storage class

Another useful feature of PVs in Kubernetes is storage class. Talking about "storage" is like talking about "fruit." What kind? Apples, bananas, mangos? In Kubernetes, when you create a PV, you can specify a storage class, for instance "fast" or "slow," and those descriptors can be defined with parameters. In this example, we have a "fast" storage class that means we run on a Google Persistent Disk using SSD. These storage classes could define iOps, backup policies, and more.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
 name: fast
provisioner: kubernetes.io/gce-pd
parameters:
 type: pd-ssd
```

Stateful sets

The last storage concept in Kubernetes we will examine is stateful sets. Stateful sets are a new concept in Kubernetes that take PV and PVCs a step further. In essence, they take management of stateful services from single node to multi-node. They provide:

- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, graceful deletion and termination

In other words, stateful sets give you the semantics to manage a group of stateful pods, such as a 3-node MySQL cluster with one master and two slaves, as a whole. You could use PV and PV Claims to manage these three containers individually, but chances are you want to manage your cluster as a cluster. For

instance, if you reschedule your MySQL cluster, you want your master to start up first and be recognizable as the master. Stateful sets provide this capability in the form of strict ordinal indexes that control that start-up ordering of pods within the stateful set, among other features.

In this example of spinning up a stateful set, we have three pods in our cluster, and the first one (mysql-0) is spun up and running before the second one, in that strict order.

```
kubectl get pods -w -l app=db
NAME       READY     STATUS
mysql-0     0/1       Pending
mysql-0     0/1       Creating
mysql-0     1/1       Running

mysql-1     0/1       Pending
mysql-1     0/1       Creating
mysql-1     1/1       Running

mysql-2     0/1       Pending
mysql-2     0/1       Creating
mysql-2     1/1       Running
```

There is no equivalent of a stateful set primitive on Swarm or Mesos.

Kubernetes volume plugins

Like Docker, Kubernetes supports the idea of volume plugins. In Kubernetes, plugins are referred to as "Types of Volumes" but the idea is the same: You can use a third-party system to add additional support for persistence, HA, and security to your data volumes above and beyond what is supported natively within the scheduler. Also like Docker, you need to understand the capabilities of your Kubernetes volume type before assuming that, for instance, you can use PV claims to provide HA for your database using the `awsElasticBlockStore volume` type. This volume type uses Amazon EBS, which takes at least 45 seconds to unmount and remount to a host when Kubernetes reschedules a container between EC2 instances.If 45 seconds or more of downtime is unacceptable for your database, you will need to look into other volume types. Using our example from earlier, a volume plugin that pooled the underlying EBS devices and synchronously replicated the data between hosts at block level would not suffer from this problem because it would need to attach the EBS device only once.

Mesosphere DC/OS

Like Docker and Kubernetes, Mesosphere DC/OS provides a mechanism to create a persistent volume from disk resources. When launching a task (the basic building block in Mesos similar to a pod in Kubernetes or a container in Docker), you can create a volume that exists outside the task's sandbox and will persist on the node even after the task dies or completes. When the task exits, its resources such as persistent volumes can be offered back to the framework, so that the same task can be launched again.

DC/OS offers two approaches to persistent storage: local and external persistent volumes. At a high-level, local persistent volumes involve using the storage available on a host that tasks run on, while external persistent volumes involve network-attached storage like Amazon EBS or an on-premises SAN. Both of these approaches have important limitations documented on the Mesosphere DC/OS website and outlined below. For full details see DC/OS docs on local and external volumes:

Caveats when using local persistent volumes

Tasks are pinned to a single host

Once a task has been run on a server on a local volume, it is "pinned" to that server and can't be rescheduled to run onto a different server, even if the initial host becomes unavailable.

Resource requirements are fixed at start time

Just as a task run on a host with a local volume is forced to always run on that host, the initial resource requirements of a task using a local volume are fixed. That means that if a database is running out of space, the administrator can't resize the volume without stopping the application, taking down the task, spinning up a new host with a larger disk, transferring the data to the new host, and bringing the task back up..

External management for replication and backups

Because a given task is pinned to a host and if that hosts dies, all data related to that task is lost, teams must think about replication and backups to protect any mission-critical infrastructure.

Caveats when using external volumes

One of the ways to get external volume support today is through connector plugins to legacy storage products. But this connector product has several pitfalls as documented below and on DC/OS website.

Only one task per volume

External volumes accessed via a connector plugin can only be used by one task at a time. This prevents applications from accessing their data from multiple hosts. The connector products prevent customers from realizing the true agility, elasticity and scale that could be unleashed by containerizing the application.

Cross-AZ limitations

Since connector plugins do not provide storage, they may not be able to ensure cross-AZ availability and security of container data volumes as cloud-based block devices and some on-prem SANs are not available across AZs or data centers.

Inconsistent launch times

Similarly, since connectors are not full fledged storage services stacks themselves, they are highly dependent on the underlying capabilities of the storage system. For EBS or Google Persistent Disk, simply starting a task with an volume mounted can take nearly a minute or more. Similar launch times can affect on-prem SANs.

Low density of stateful containers per host

There are some other limitations of connector-based external volumes that are important to understand. First, while thousands of containers can run per host, there are limitations to how many block devices can mount to a host due to the networking limitations in the Linux kernel itself. For example, when using a connector plugin, a maximum of 40 EBS volumes per EC2 instance can be mounted before risking boot failure. So while simple, the model of one EBS volume per container volume can severely limit the density of containers that can be achieved per host. If reducing infrastructure costs or increasing density is a desired attribute for an application, a connector-based approach might not be a good fit.

Slow block device mounts

Additionally, the mount operation of a physical block device to a container host can be a slow operation, taking up to 2 minutes, and frequently failing altogether, requiring a host reboot. As a result of the time it takes to unmount and remount a block device, failover operations, such as those supported by connector plugins, do not function well as a HA mechanism.

Volumes at run time can't be dynamically provisioned

Connector plugins do not support in-line volume specification which allows DC/OS platform users to launch containers that use storage without having to log a ticket and wait for someone to manually create a volume for their app.

Can't scale up stateful apps

One of the biggest benefits of a container platform is the ability to easily add more

instances to a running application so we can smoothly keep up with incoming user requests. This is known as app scaling and the concept is supported from with the DC/OC UI. An operator can simply increase the number of instances from 1 to 2, for example. If the app was using a volume exposed via a connector, however, that request would be blocked by the platform since the use of a connector volume limits the number of tasks an app can have to 1. An operator can "scale down" to 0 to effectively pause the app and then scale back up to 1 but can not go past that when using a connector.

Being able to scale apps is important, for example, for running Consul on Marathon. Other examples include things like DiskCache and EHCache which would also like to be able to scale volumes with compute when running on Marathon.

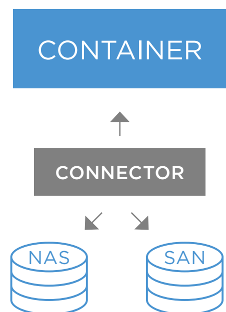
To run stateful services successfully on Mesosphere DC/OS, it is important to choose a data services platform such as Portworx that overcomes these issues.

DATA ARCHITECTURES FOR CONTAINERIZED APPLICATIONS

With an understanding of the storage primitives available to containerized applications, let's turn now to how you can architect your application to take advantage of these primitives.

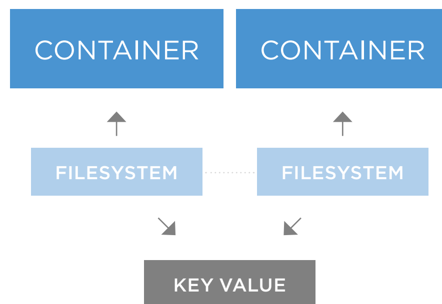
Three types of stateful architectures have emerged for distributed applications, and while they all leverage a plugin model (Docker volume plugin for Docker Swarm and Mesos, or native Kubernetes plugins for Kubernetes), they are quite different in terms of their performance, due to important architectural differences. Ultimately, your choice of the storage technology matters a lot and depends on your applications and usage, e.g., if you are in production or development. Let's look at each architecture in turn.

Connector Based



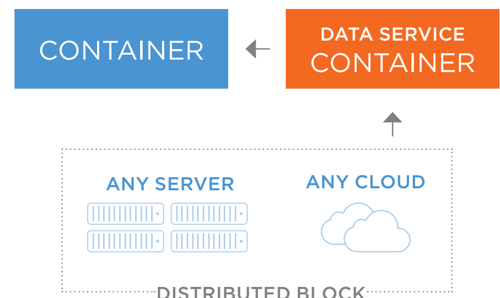
Supports mapping to OS volumes.

Key Value Based



Supports file syncing.

Container Data Services



Supports distributed block, file, object.

Connector-based systems

The most common type of volume management that has emerged is the connector-based system. Examples include Flocker from ClusterHQ, EMC RexRay, and a growing number of native Docker storage drivers for storage systems such as Amazon EBS. These volume plugins take physical or software-defined volumes and map them 1:1 into a container.

If you remember the rider and horse example above, the rider is the connector, for example RexRay, and the horse is the storage system that it plugs into, perhaps EMC Isilon or Amazon EBS. They are called connector-based systems because they connect storage to containers; these connectors don't provide the storage itself.

Advantages of connector-based systems:

- Allow you to use your current storage system for container storage
- Are generally free to use
- Relatively easy setup

Connector-based systems also have some disadvantages, specifically related to the fact that they provide a persistence layer for containers by plugging into an existing storage solution. As a result, they pass through the storage characteristics of the underlying system to the containerized application.

For example, the EBS Docker plugin makes it trivially easy to mount an EBS volume to a Docker container using Kubernetes, Mesos, or Swarm. However, some issues can emerge as a result. We will use AWS EBS as an example of these general, storage-system-related problems. You should look at your own storage system to see if these specific problems apply.

Low density of stateful containers per host

First, while you can run thousands of containers per host, you can mount a maximum of 40 EBS volumes per EC2 instance before risking boot failure. So while simple, the model of one EBS volume per container volume can severely limit the density of containers you can achieve per host. If reducing infrastructure costs or increasing density is a desired attribute of your application, a connector-based approach might not be a good fit.

Slow block device mounts

Secondly, the mount operation of a physical block device to a container host can be a slow operation, taking 45 seconds or longer, and frequently failing altogether, requiring a host reboot. As a result of the time it takes to unmount and remount a block device, failover operations, such as those supported by Kubernetes using persistent volume claims, do not function well as an HA mechanism.

Plugins such as RexRay, Flocker, EBS, and ScaleIO all suffer from these storage-system-related limitations. It is not that the plugin itself is faulty. The plugins are just passing along the capabilities of the underlying storage.

Key-value based systems

Another type of container storage has emerged based on key-value storage. Examples include Infinit, acquired by Docker, and the recently deprecated Torus by CoreOS.

These new storage systems, many so new they are still in alpha, are good for file streaming and non-critical workloads bound by web access latencies, but due to the architectural choice of building on top of a key-value store, are not suitable for transactional, low-latency, or data-consistency-dependant applications.

There are two main types storage systems that use a key-value-based backend storage. The first store the actual volume data in the key-value backend. An example includes Infinit from Docker. This is similar to creating a filesystem or a storage system based on an object store in the back. The problem is that object stores like S3 and key-value stores like etcd are meant for write-once, read-many workloads. This means that with regular primary filesystem workloads, the key-value backend will very quickly deteriorate and end up with major garbage collection issues. They are also not designed to be highly performant for transactional and low latency workloads, which means the applications like databases cannot run on them.

Other types of storage systems attempt to encode volume metadata (either a file's location or a block's logical location) into the key-value store. An example of this includes the recently deprecated Torus from CoreOS. This implies that the key-value store is in the data path for every single IO operation in order to lookup the data's physical location. This creates a single point of failure and a bottleneck. Again, transactional workloads like databases cannot rely on a system like this.

Container data services platform

As we've seen above, connector-based systems have the advantage of easily connecting storage to containers, but they can suffer from the drawbacks of their underlying storage systems, which are not optimized for container workloads. Key-value based systems, on the other hand, implement a storage system that works great for some workloads such as file streaming, but that is not optimized for more common database workloads.

Container data services platforms are storage systems that combine a native container integration of a connector-based system with a cloud-native

container-optimized storage system. Portworx is an example of a container data services platform.

Portworx provides cloud-native and container-granular data service solutions built on top of an enterprise-grade distributed block storage systems. Portworx supports workloads such as databases, queues, and other file-based applications, with cloud-native architectures in mind. The container storage solution is built with the founding principles of ease of use, DevOps-led programmability, and integration with any container scheduler. Built by a team of storage engineers with DevOps experience, Portworx has data correctness, availability, integrity, and performance at its core, without sacrificing usability.

COMMON USE CASES FOR STATEFUL CONTAINERS

With four of the top 10 Docker images on hub.docker.com being stateful (Redis, MySQL, MongoDB, ElasticSearch), it's clear that stateful containers are being used today in large numbers. The following section describes some of the most common use cases.

Reliably deploy and operate containerized databases, queues, and key-value stores

The most common use of stateful containers today is simply the reliable, automated deployment and operation of containerized databases, queues, and key-value stores. Any non-trivial application uses some type of database, queue, or key-value store to manage state, so this is an obvious place for teams running containerized apps to find themselves.

Advantages of running databases, queues, and key-value stores in containers:

- Higher density than can be achieved using VMs
- Near bare-metal performance compared to virtualization
- Process isolation

However, when it comes to running these data services in containers, problems can arise because containers and popular container schedulers such as Kubernetes, Mesos, and Swarm weren't designed to handle the problems associated with databases.

Problems with containerized databases today:

- When a container dies, it can lose data if persistence is not set up correctly
- Popular schedulers not designed for stateful services provide only limited functionality
- App-level replication requires domain-specific knowledge for each database

When picking the data architecture for your stateful application, consider these questions:

- Do you need the performance of a hyper-converged architecture, where your containers and your data run on the same host?
- Do you need your stateful container to be HA?
- How will you do backups?

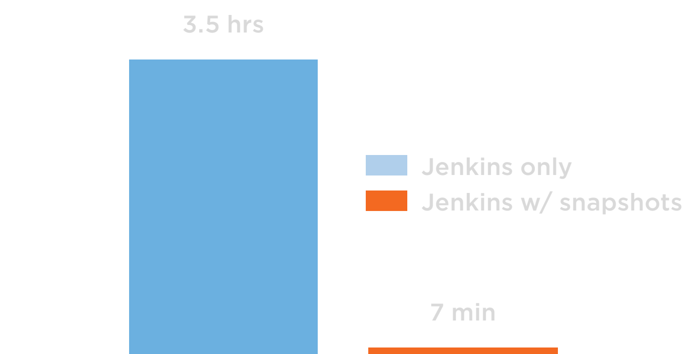
Speeding up your CI/CD pipeline

Often containerization is part of a larger microservices and DevOps transformation, and CI/CD is a key component of both. Effective CI/CD requires that you can quickly, and reliably, recreate your production environment for testing. Moreover, you must be able to reset your environment after each test run, or your results will be subject to failures caused by minor changes to configuration state between test runs.

Problems with CI/CD today:

- Pulling images to create new build environments is time-consuming
- Horizontal scaling is slow because an entire Jenkins /home or similar directory must be pulled
- Incremental builds speed up testing but are error-prone

Containerized CI systems are a good way to overcome these challenges, because container volumes can be snapshotted and used as a golden image for tests.



Horizontally scale your CI/CD cluster without the wait

The Jenkins `/home` directory or equivalent in TravisCI, Bamboo, and other popular CI/CD systems includes all the images and artifacts you need to run your tests. Once it's been built the first time from your container registry and version control system, you can snapshot it and use those snapshots as the basis for unlimited Jenkins slaves. With a container data services solution such as Portworx, snapshots are available to the cluster globally. So no matter where your Jenkins job is scheduled, your tests can begin right away, without any wait.

Speed up incremental builds

Because pulling all the images and artifacts needed for a test run is time consuming, it is tempting to use the same environment for multiple test runs. The problem is that database state or configuration can often change in the course of a test run, causing subsequent runs to fail—not due to real errors in your code, but because of minor differences in your configuration. Sorting out the cause of these errors is time consuming, so many revert to pulling Jenkins/home or equivalent fresh each time. However, once you've built

Jenkins /home, you can snapshot it and use the snapshot as a golden image for subsequent test runs. This will dramatically speed up your incremental builds.

Containerize your data processing workloads

In a world of Big Data, data processing applications play a major role in business intelligence. Workloads built on tools such as ElasticSearch, Riak, Cassandra, and Hadoop allow large amounts of data to be processed quickly. But to support these workloads, DevOps teams need to be able to support a variety of big data workloads on the same infrastructure.

What these workloads have in common is the need to scale compute independently from underlying storage. That means that data should be separate from, but accessible to, your compute cluster. Containers can help.

Problems with data processing today:

- Storage infrastructure doesn't map onto scale-out compute clusters
- Slow volume provisioning times make it hard to quickly scale compute
- Hard to support iOPS-intensive workloads and batch jobs on the same infrastructure

Containerized storage that maps onto your data processing workloads

Some container storage systems are designed to work alongside scale-out compute clusters like those powering big data workloads. By taking commodity servers and turning them into a hyper-converged scale-out storage cluster, you can scale your storage alongside your compute cluster.

Volumes are ready as soon as your container starts

One of the main benefits of containers is how quickly they start. However, if you have to wait 45 seconds or more to mount a volume to a container each time it starts, bursting to 1000 nodes for quick data processing is slowed down dramatically. While it might not be appropriate to use connector-based storage systems dependent on mount times of the underlying hardware, a storage system that enables on-demand data volumes for your containers could be a good choice.

Class of service lets you pick the right storage for your data processing job

Not all Big Data workloads are created equal. Your ElasticSearch-based business intelligence tooling might need iOPS-optimized storage while your Hadoop batch jobs might be fine running on slightly slower, but much cheaper, HDDs. With support for Storage Classes in orchestration systems such as Kubernetes, you can match the containerized workload to the storage infrastructure optimized for the task at hand.

Systems such as Portworx can automatically fingerprint all storage resources

in your cluster and provide this storage back to containerized applications based on the class-of-service (COS) requested by the container. By setting the COS to High for ElasticSearch and Low for Hadoop, your jobs will automatically run on the most efficient hardware—allowing you to tier your storage.

Containerize your CMS, simplify management

Wordpress and other popular content management systems including Drupal and Joomla are inherently stateful, storing important data in the application itself. This makes horizontally scaling these services extremely difficult.

However, container-aware storage systems that expose an NFS-based file layer let you horizontally scale out CMS sites like Wordpress while managing a single file layer. Specifically, you can run multiple Wordpress, Drupal, or Joomla containers and have access to the same underlying data volumes, sharing states between hosts, without worrying about users colliding on the same file.

Problems with containerizing CMSs today:

- CMS are inherently stateful
- Hard to share a single volume between hosts

Multiple CMS containers, one volume

Some container storage systems support both multi-reader and multi-writer modes. That means you can spin up multiple Wordpress or other CMS containers and back them all onto a single (but shared) Docker volume. This allows you to horizontally scale your CMS without adding multiple new volumes that you will have to back up, make HA, and otherwise manage.

Ops automation for your CMS

In addition to providing a single volume for all your CMS containers, many storage systems can automate operations with snapshots, backups, and encryption. They can also let you manage your CMS containers from directly within your scheduler of choice, be it Kubernetes, Mesos, or Swarm.

SOURCES

https://docs.mesosphere.com/
https://docs.docker.com/
https://kubernetes.io/docs/

APPENDIX

Storage drivers

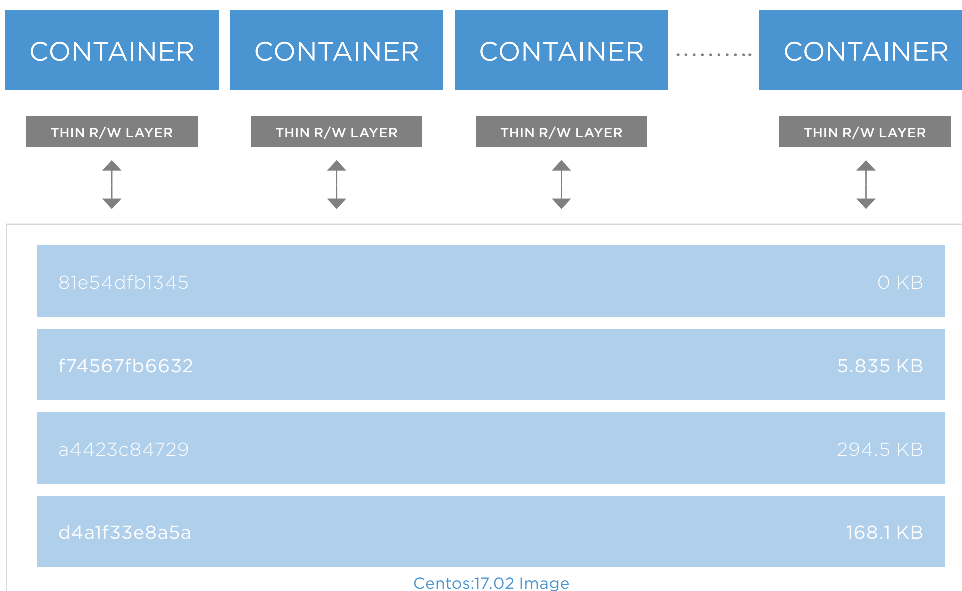
One area of Docker data management that causes a lot of confusion is storage drivers. Storage drivers are tools such as Device Mapper, AUFS, Overlay, and LCFS that many people have heard of and used without understanding what purpose they serve or how one is different from another.

While an important part of state management for containers, storage drivers are not the same thing as volume plugins that we saw earlier.

While volume plugins manage the data volume associated with a container, a storage driver manages the state of the container image itself. In other words, it does not manage the data in a MongoDB database. Rather, it manages the code that makes up MongoDB itself and the underlying operating system files and anything else that is not a specifically mounted data volume. 100% of all containers, even so-called "stateless" containers, use a storage driver for these ephemeral layers.

Every time a container is built, committed, pulled, or destroyed, a storage driver is used. They are absolutely fundamental to overall container performance, even if the container being run is 100% stateless such as Apache or NGINX.

The Docker storage driver manages a thin writeable layer that every container has. To see what this means, look at the diagram below.



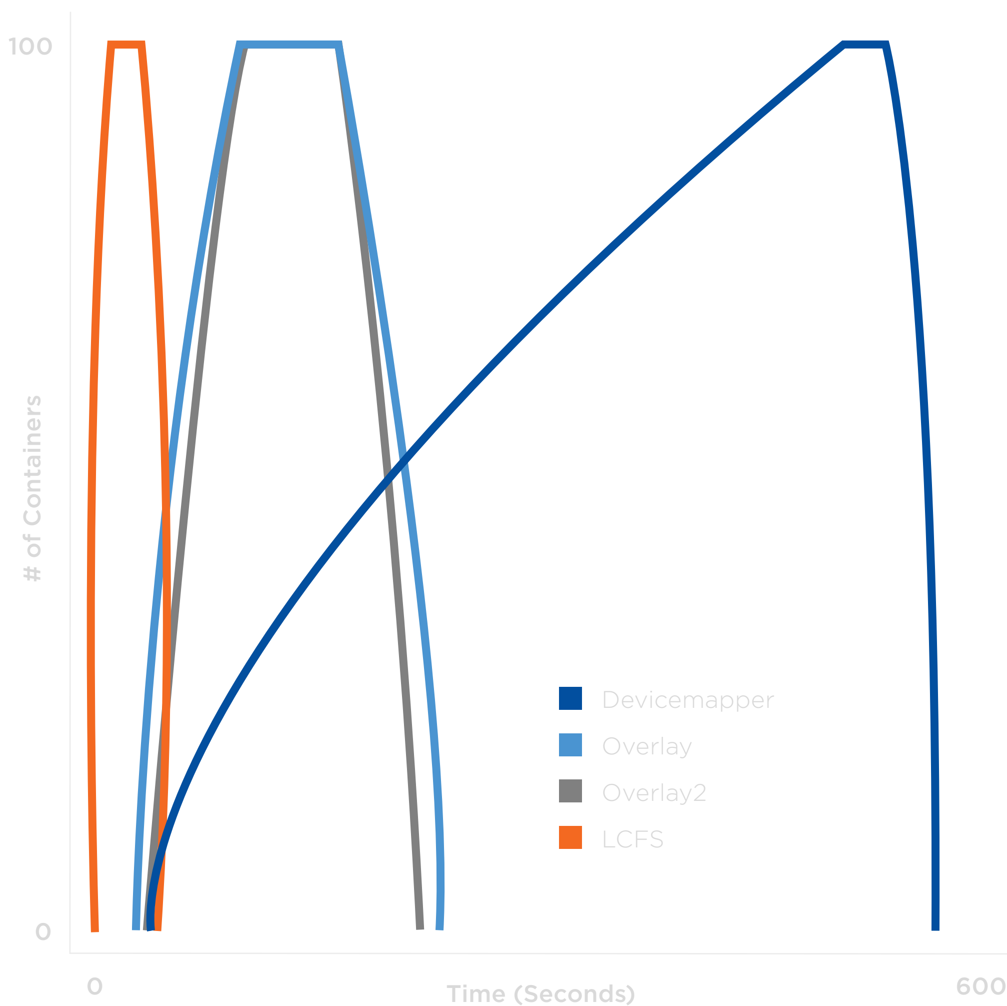
There are two parts to this picture. Starting at the bottom, in the box, we have 4 "layers" that make up the base image. We could run 1000 CentOs containers and they would all share this base and it never changes because it is read-only.

On top of this base, we have a thin read-write layer that is sometimes called the container layer. All writes made to a container are stored in this thin writable layer: any change that is made to a container gets made in this top-most layer.

Without a storage driver, you could never pull or run, commit or build a container. The reason is because each container needs this thin read-write layer to operate and the storage driver manages that layer.

The reason that this is important, is because how a storage driver manages the thin read-write layer affects performance of all container operations. Not just stateful container operations.

This chart shows the times in seconds to create and then destroy a certain number of containers. Some take more time than others.



For example, imagine a container image with 100 layers. If an application running in a container needs to add a single new value to the very bottom layer, how long will that take? The answer depends on how the storage driver implements that operation.

For example, in AUFS, the most popular driver on RHEL, the driver will search each image layer for the file starting at the top and going to the bottom. When the file that needs to be modified is found, the entire file will be copied up to the container's top writable layer. From there, it can be opened and modified.

The larger the file that needs to be copied, or the more layers there are in the container, the longer that operation will take. Other storage drivers take a different approach, making things faster or slower as a result.

Portworx, Inc.
4940 El Camino Real, Suite 200
Los Altos, CA 94022
Tel: 650-241-3222 | info@portworx.com | www.portworx.com
© Portworx BPG7-15-17