O'REILLY®

Kubernetes Best Practices

Blueprints for Building Successful Applications on Kubernetes

Free Chapters

compliments of

Cockroach Labs

Brendan Burns, Eddie Villalba, Dave Strebel & Lachlan Evenson

Cockroach Labs

K8S + CockroachDB = Effortless App Deployment

Run your application on the cloud-native database uniquely suited to Kubernetes.

Scale elastically with distributed SQL

Say goodbye to sharding and time-consuming manual scaling.

Survive anything with bullet-proof resilience

Rest easy knowing your application data is always on and always available.

Build fast with PostgreSQL compatibility

CockroachDB works with your current applications and fits how you work today.

Get started for free today

cockroachlabs.com/k8s

Trusted by innovators

COMCAST SPACEX BOSE LUSH rubrik wework

Kubernetes Best Practices

Blueprints for Building Successful Applications on Kubernetes

This Excerpt contains Chapter 7 and Chapter 16 of *Kubernetes Best Practices*. The final book is available on the O'Reilly Learning Platform and through other retailers.

Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes Best Practices
by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Elizabeth Kelly
Copyeditor: Charles Roumeliotis
Proofreader: Sonia Saruba
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the First Release

2019-11-12: First Release
2020-07-02: Second Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=0636920273219* for release details.

978-1-492-09083-0

[LSI]

Table of Contents

Foreword

It's an undeniable fact that Kubernetes (K8s) has taken the world of app development and delivery by storm in the last five years. The container orchestration project—bolstered by the rise of microservices—has fundamentally changed how we think about structuring applications, managing deployments, and using infrastructure. The level of freedom, flexibility, and control Kubernetes has brought to us is unparalleled in past innovations. Unless you work at Google, Spotify, or one of the other innovative early adopters of Kubernetes, you (like many of us) are still trying to understand and fully unlock the benefits of moving to this system.

Modern applications are overwhelmingly data-intensive. We have all come to expect highly dynamic, personalized experiences to be available instantly, with up-to-the-millisecond data. There is zero tolerance for latency, downtime, or (the cardinal sin) incorrect data. These applications are increasingly global in nature, as well—think Netflix, Uber, or Square—and have complex needs in trying to balance both consistency and locality. How do you square the desire for a single codebase and dataset with the demand for ultra-fast user experiences and data sovereignty rules?

Thankfully, these are two areas that Kubernetes has helped make much, much easier. Kubernetes has dramatically simplified our ability to deliver distributed global or multiregion applications. And, when used in partnership with a distributed database or as a platform to directly manage stateful services like databases, Kubernetes creates the ideal platform for the next generation of breakthrough applications.

It is for this reason that Cockroach Labs is proud to sponsor and make available two chapters from *Kubernetes Best Practices*: "Worldwide Application Distribution and Staging" and "Managing State and Stateful Applications."

Cockroach Labs is the maker of CockroachDB, the leading cloud native, distributed SQL database for modern applications—and *the only database built for Kubernetes*. Our customers are at the forefront of innovation in distributed, data-intensive applications, and we're excited to help bring that perspective and insight to every developer who could benefit.

We hope you enjoy this book excerpt, and that you consider CockroachDB for your next development project.

CHAPTER 7

Worldwide Application Distribution and Staging

So far throughout this book, we have seen a number of different practices for building, developing, and deploying applications, but there is a whole different set of concerns when it comes to deploying and managing an application with a worldwide footprint.

There are many different reasons why an application might need to scale to a global deployment. The first and most obvious one is simply scale. It might be that your application is so successful or mission critical that it simply needs to be deployed around the world in order to provide the capacity needed for its users. Examples of such applications include a worldwide API gateway for a public cloud provider, a large-scale IoT product with a worldwide footprint, a highly successful social network, and more.

Although there are relatively few of us who will build out systems that require worldwide scale, many more applications require a worldwide footprint for latency. Even with containers and Kubernetes there is no getting around the speed of light, and thus to minimize latency to our applications, it is sometimes necessary to distribute our applications around the world to minimize the distance to our users.

Finally, an even more common reason for global distribution is locality. Either for reasons of bandwidth (e.g., a remote sensing platform) or data privacy (geographic restrictions), it is sometimes necessary to deploy an application in specific locations for the application to be possible or successful.

In all of these cases, your application is no longer simply present in a small handful of production clusters. Instead it is distributed across tens to hundreds of different geographic locations, and the management of these locations, as well as the demands of rolling out a globally reliable service, is a significant challenge. This chapter covers approaches and practices for doing this successfully.

Distributing Your Image

Before you can even consider running your application around the world, you need to have that image available in clusters located around the globe. The first thing to consider is whether your image registry has automatic geo-replication. Many image registries provided by cloud providers will automatically distribute your image around the world and resolve a request for that image to the storage location nearest to the cluster from which you are pulling the image. Many clouds enable you to decide where you want to replicate the image; for example, you might know of locations where you are not going to be present. An example of such a registry is the Microsoft Azure container registry, but others provide similar services. If you use a cloud-provided registry that supports geo-replication, distributing your image around the world is simple. You push the image into the registry, select the regions for geo-distribution, and the registry takes care of the rest.

If you are not using a cloud registry, or your provider does not support automatic geo-distribution of images, you will need to solve that problem yourself. One option is to use a registry located in a specific location. There are several concerns about such an approach. Image pull latency often dictates the speed with which you can launch a container in a cluster. This in turn can determine how quickly you can respond to a machine failure, given that generally in the case of a machine failure, you will need to pull the container image down to a new machine.

Another concern about a single registry is that it can be a single point of failure. If the registry is located in a single region or a single datacenter, it's possible that the registry could go offline due to a large-scale incident in that datacenter. If your registry goes offline, your CI/CD pipeline will stop working, and you'll be unable to deploy new code. This obviously has a significant impact on both developer productivity and application operations. Additionally, a single registry can be much more expensive because you will be using significant bandwidth each time you launch a new container, and even though container images are generally fairly small, the bandwidth can add up. Despite these negatives, a single registry solution can be the appropriate answer for small-scale applications running in only a few global regions. It certainly is simpler to set up than full-scale image replication.

If you cannot use cloud-provided geo-replication and you need to replicate your image, you are on your own to craft a solution for image replication. To implement such a service, you have two options. The first is to use geographic names for each image registry (e.g., `us.my-registry.io`, `eu.my-registry.io`, etc.). The advantage of this approach is that it is simple to set up and manage. Each registry is entirely independent, and you can simply push to all registries at the end of your CI/CD pipeline.

The downside is that each cluster will require a slightly different configuration to pull the image from the nearest geographic location. However, given that you likely will have geographic differences in your application configurations anyway, this downside is relatively easy to manage and likely already present in your environment.

Parameterizing Your Deployment

When you have replicated your image everywhere, you need to parameterize your deployments for different global locations. Whenever you are deploying to a variety of different regions, there are bound to be differences in the configuration of your application in the different regions. For example, if you don't have a geo-replicated registry, you might need to tweak the image name for different regions, but even if you have a geo-replicated image, it's likely that different geographic locations will present different load on your application, and thus the size (e.g., the number of replicas) as well as other configuration can be different between regions. Managing this complexity in a manner that doesn't incur undue toil is key to successfully managing a worldwide application.

The first thing to consider is how to organize your different configurations on disk. A common way to achieve this is by using a different directory for each global region. Given these directories, it might be tempting to simply copy the same configurations into each directory, but doing this is guaranteed to lead to drift and changes between configurations in which some regions are modified and other regions are forgotten. Instead, using a template-based approach is the best idea so that most of the configuration is retained in a single template that is shared by all regions, and then parameters are applied to that template to produce the region-specific templates. Helm is a commonly used tool for this sort of templating.

Load-Balancing Traffic Around the World

Now that your application is running around the world, the next step is to determine how to direct traffic to the application. In general, you want to take advantage of geographic proximity to ensure low-latency access to your service. But you also want to failover across geographic regions in case of an outage or any other source of service failure. Correctly setting up the balancing of traffic to your various regional deployments is key to the establishment of both a performant and reliable system.

Let's begin with the assumption that you have a single hostname that you want to serve as your service. For example, *myapp.myco.com*. One initial decision that you need to make is whether you want to use the Domain Name System (DNS) protocol to implement load balancing across your regional endpoints. If you use DNS for load balancing, the IP address that is returned when a user makes a DNS query to *myapp.myco.com* is based on both the location of the user accessing your service as well as the current availability of your service.

Reliably Rolling Out Software Around the World

After you have templatized your application so that you have proper configurations for each region, the next important problem is how to deploy these configurations around the world. It might be tempting to simultaneously deploy your application worldwide so that you can efficiently and quickly iterate your application, but this, although Agile, is an approach that can easily leave you with a global outage. Instead, for most production applications, a more carefully staged approach to rolling out your software around the world is more appropriate. When combined with things like global load balancing, these approaches can maintain high availability even in the face of major application failures.

Overall, when approaching the problem of a global rollout, the goal is to roll out software as quickly as possible, while simultaneously detecting issues quickly—ideally before they affect any other users. Let's assume that by the time you are performing a global rollout, your application has already passed basic functional and load testing. Before a particular image (or images) is certified for a global rollout, it should have gone through enough testing that you believe the application is operating correctly. It iss important to note that this *does not* mean that your application *is* operating correctly. Though testing catches many problems, in the real world, application problems are often first noticed when they are rolled out to production traffic. This is because the true nature of production traffic is often difficult to simulate with perfect fidelity. For example, you might test with only English language inputs, whereas in the real world, you see input from a variety of languages. Or your set of test inputs is not comprehensive for the real-world data your application ingests. Of course, any time that you do see a failure in production that wasn't caught by testing, it is a strong indicator that you need to extend and expand your testing. Nonetheless, it is still true that many problems are caught during a production rollout.

With this in mind, each region that you roll out to is an opportunity to discover a new problem. And, because the region is a production region, it is also a potential outage to which you will need to react. These factors combine to set the stage for how you should approach regional rollouts.

Pre-Rollout Validation

Before you even consider rolling out a particular version of your software around the world, it's critically important to validate that software in some sort of synthetic testing environment. If you have your CD pipeline set up correctly, all code prior to a particular release build will have undergone some form of unit testing, and possibly limited integration testing. However, even with this testing in place, it's important to consider two other sorts of tests for a release before it begins its journey through the release pipeline. The first is complete integration testing. This means that you assemble the entirety of your stack into a full-scale deployment of your application but without any real-world traffic. This complete stack generally will include either a copy of your production data or simulated data on the same size and scale as your true production data. If in the real world, the data in your application is 500 GB, it's critical that in preproduction testing your dataset is roughly the same size (and possibly even literally the same dataset).

Generally speaking, this is the most difficult part of setting up a complete integration test environment. Often, production data is really present only in production, and generating a synthetic dataset of the same size and scale is quite difficult. Because of this complexity, setting up a realistic integration testing dataset is a great example of a task that it pays to do early on in the development of an application. If you set up a synthetic copy of your dataset early, when the dataset itself is quite small, your integration test data grows gradually at the same pace as your production data. This is generally significantly more manageable than if you attempt to duplicate your production data when you are already at scale.

Sadly, many people don't realize that they need a copy of their data until they are already at a large scale and the task is difficult. In such cases it might be possible to deploy a read/write-deflecting layer in front of your production data store. Obviously, you don't want your integration tests writing to production data, but it is often possible to set up a proxy in front of your production data store that reads from production but stores writes in a side table that is also consulted on subsequent reads.

Regardless of how you manage to set up your integration testing environment, the goal is the same: to validate that your application behaves as expected when given a series of test inputs and interactions. There are a variety of ways to define and execute these tests—from the most manual, a worksheet of tests and human effort (not recommended because it is fairly error prone), through tests that simulate browsers and user interactions, like clicks and so forth. In the middle are tests that probe RESTful APIs but don't necessarily test the web UI built on top of those APIs. Regardless of how you define your integration tests, the goal should be the same: an automated test suite that validates the correct behavior of your application in response to a complete set of real-world inputs. For simple applications it may be possible to perform this validation in premerge testing, but for most large-scale real-world applications, a complete integration environment is required.

Integration testing will validate the correct operation of your application, but you should also load-test the application. It is one thing to demonstrate that the application behaves correctly, it is quite another to demonstrate that it stands up to real-world load. In any reasonably high-scale system, a significant regression in performance—for example, a 20% increase in request latency—has a significant impact on the UX of the application and, in addition to frustrating users, can cause an application to completely fail. Thus, it is critical to ensure that such performance regressions do not happen in production.

Like integration testing, identifying the correct way to load-test an application can be a complex proposition; after all, it requires that you generate a load similar to production traffic but in a synthetic and reproduceable way. One of the easiest ways to do this is to simply replay the logs of traffic from a real-world production system. Doing this can be a great way to perform a load-test whose characteristics match what your application will experience when deployed. However, using replay isn't always foolproof. For example, if your logs are old, and your application or dataset has changed, it's possible that the performance on old, replayed logs will be different that the performance on fresh traffic. Additionally, if you have real-world dependencies that you haven't mocked, it's possible that the old traffic will be invalid when sent over to the dependencies (e.g., the data might no longer exist).

Because of these challenges, many systems, even critical systems, are developed for a long time without a load test. Like modeling your production data, this is a clear example of something that is easier to maintain if you start earlier. If you build a load-test when your application has only a handful of dependencies, and improve and iterate the load-test as you adapt your application, you will have a far easier time than if you attempt to retrofit load-testing onto an existing large-scale application.

Assuming that you have crafted a load test, the next question is the metrics to watch when load-testing your application. The obvious ones are requests per second and request latency because those are clearly the user-facing metrics.

When measuring latency, it's important to realize that this is actually a distribution, and you need to measure both the mean latency as well as the outlier percentiles (like the 90th and 99th percentile) since they represent the "worst" UX of your application. Problems with very long latencies can be hidden if you just look at the averages, but if 10% of your users are having a bad time, it can have a significant impact on the success of your product.

In addition, it's worth looking at the resource usage (CPU, memory, network, disk) of the application under load test. Though these metrics do not directly contribute to the UX, large changes in resource usage for your application should be identified and understood in preproduction testing. If your application is suddenly consuming twice as much memory, it's something you will want to investigate, even if you pass your load test, because eventually such significant resource growth will affect the quality and availability of your application. Depending on the circumstances, you might continue bringing a release to production, but at the same time, you need to understand why the resource footprint of your application is changing.

Canary Region

When your application appears to be operating correctly, the first step should be a *canary region*. A canary region is a deployment that receives real-world traffic from people and teams who want to validate your release. These can be internal teams that depend on your service, or they might be external customers who are using your service. Canaries exist to give a team some early warning about changes that you are about to roll out that might break them. No matter how good your integration and load testing, it's always possible that a bug will slip through that isn't covered by your tests, but is critical to some user or customer. In such cases, it is much better to catch these issues in a space where everyone using or deploying against the service understands that there is a higher probability of failure. This is what the canary region is.

Canaries must be treated as a production region in terms of monitoring, scale, features, and so on. However, because it is the first stop on the release process, it is also the location most likely to see a broken release. This is OK; in fact it is precisely the point. Your customers will knowingly use a canary for lower-risk use cases (e.g., development or internal users) so that they can get an early indication of any breaking changes that you might be rolling out as part of a release.

Because the goal of a canary is to get early feedback on a release, it is a good idea to leave the release in the canary region for a few days. This enables a broad collection of customers to access it before you move on to additional regions. The need for this length of time is that sometimes a bug is probabilistic (e.g., 1% of requests) or it manifests only in an edge case that takes some time to present itself. It might not even be severe enough to trigger automated alerts, but there might be a problem in business logic that is visible only via customer interactions.

Identifying Region Types

When you begin thinking about rolling out your software across the world, it's important to think about the different characteristics of your different regions. After you begin rolling out software to production regions, you need to run it through integration testing as well as initial canary testing. This means that any issues you find will be issues that did not manifest in either of these settings. Think about your different regions. Do some get more traffic than others? Are some accessed in a different way? An example of a difference might be that in the developing world, traffic is more likely to come from mobile web browsers. Thus, a region that is geographically close to more developing countries might have significantly more mobile traffic than your test or canary regions.

Another example might be input language. Regions in non-English speaking areas of the world might send more Unicode characters that could manifest bugs in string or character handling. If you are building an API-driven service, some APIs might be more popular in some regions versus others. All of these things are examples of differences that might be present in your application and might be different than your canary traffic. Each of these differences is a possible source of a production incident. Build a table of different characteristics that you think are important. Identifying these characteristics will help you plan your global rollout.

Constructing a Global Rollout

Having identified the characteristics of your regions, you want to identify a plan for rolling out to all regions. Obviously, you want to minimize the impact of a production outage, so a great first region to start with is a region that looks mostly like your canary and has light user traffic. Such a region is very unlikely to have problems, but if they do occur, the impact is also smaller because the region receives less traffic.

With a successful rollout to the first production region, you need to decide how long to wait before moving on to the next region. The reason for waiting is not to artificially delay your release; rather, it's to wait long enough for a fire to send up smoke. This time-to-smoke period is a measure of generally how long it takes between a rollout completing and your monitoring seeing some sign of a problem. Clearly if a rollout contains a problem, the minute the rollout completes, the problem is present in your infrastructure. But even though it is present, it can take some time to manifest. For example, a memory leak might take an hour or more before the impact of the leaked memory is clearly discernible in monitoring or is affecting users. The time-to-smoke is the probability distribution that indicates how long you should wait in order to have a strong probability that your release is operating correctly. Generally speaking, a decent rule of thumb is doubling the average time it takes for a problem to manifest.

If, over the past six months, each outage took an average of an hour to show up, waiting two hours between regional rollouts gives you a decent probability that your release is successful. If you want to derive richer (and more meaningful) statistics based on the history of your application, you can estimate this time-to-smoke even more closely.

Having successfully rolled out to a canary-like, low-traffic region, it's time to roll out to a canary-like, high-traffic region. This is a region where the input data looks like that in your canary, but it receives a large volume of traffic. Because you successfully rolled out to a similar looking region with lower traffic, at this point the only thing you are testing is your application's ability to scale. If you safely perform this rollout, you can have strong confidence in the quality of your release.

After you have rolled out to a high-traffic region receiving canary-like traffic, you should follow the same pattern for other potential differences in traffic. For example, you might roll out to a low-traffic region in Asia or Europe next. At this point, it might be tempting to accelerate your rollout, but it is critically important to roll out only to a single region that represents any significant change in either input or load to your release. After you are confident that you have tested all of the potential variability in the production input to your application, you then can start parallizing the release to speed it up with strong confidence that it is operating correctly and your rollout can complete successfully.

When Something Goes Wrong

So far, we have seen the pieces that go into setting up a worldwide rollout for your software system, and we have seen the ways that you can structure this rollout to minimize the chances that something goes wrong. But what do you do when something actually does go wrong? All emergency responders know that in the heat and panic of a crisis, your brain is significantly stressed and it is much more difficult to remember even the simplest processes. Add to this pressure the knowledge that when an outage happens, everyone in the company from the CEO down is going to be feverishly waiting for the "all clear" signal, and you can see how easy it is to make a mistake under this pressure. Additionally, in such circumstances, a simple mistake, like forgetting a particular step in a recovery process, can make a bad situation an order of magnitude worse.

For all of these reasons, it is critical that you are capable of responding quickly, calmly, and correctly when a problem happens with a rollout. To ensure that everything necessary is done, and done in the correct order, it pays to have a clear checklist of tasks organized in the order in which they are to be executed as well as the expected output for each step. Write down every step, no matter how obvious it might seem. In the heat of the moment, even the most obvious and easy steps can be the ones that are forgotten and accidentally skipped.

The way that other first responders ensure a correct response in a high-stress situation is to practice that response without the stress of the emergency. The same practice applies to all the activities that you might take in response to a problem with your rollout. You begin by identifying all of the steps needed to respond to an issue and perform a rollback. Ideally, the first response is to "stop the bleeding," to move user traffic away from the impacted region(s) and into a region where the rollout hasn't happened and your system is operating correctly. This is the first thing you should practice. Can you successfully direct traffic away from a region? How long does it take?

The first time you attempt to move traffic using a DNS-based traffic load balancer, you will realize just how long and in how many ways our computers cache DNS entries. It can take nearly a day to fully drain traffic away from a region using a DNS-based traffic shaper. Regardless of how your first attempt to drain traffic goes, take notes. What worked well? What went poorly? Given this data, set a goal for how long a traffic drain should take in terms of time to drain a percentage of traffic, for example, being able to drain 99% of traffic in less than 10 minutes. Keep practicing until you can achieve that goal. You might need to make architectural changes to make this possible. You might need to add automation so that humans aren't cutting and pasting commands. Regardless of necessary changes, practice will ensure that you are more capable at responding to an incident and that you will learn where your system design needs to be improved.

The same sort of practice applies to every action that you might take on your system. Practice a full-scale data recovery. Practice a global rollback of your system to a previous version. Set goals for the length of time it should take. Note any places where you made mistakes, and add validation and automation to eliminate the possibility of mistakes. Achieving your incident reaction goals in practice gives you confidence that you will be able to respond correctly in a real incident. But just like every emergency responder continues to train and learn, you too need to set up a regular cadence of practice to ensure that everyone on a team stays well versed in the proper responses and (perhaps more important) that your responses stay up to date as your system changes.

Worldwide Rollout Best Practices

- Distribute each image around the world. A successful rollout depends on the release bits (binaries, images, etc.) being nearby to where they will be used. This also ensures reliability of the rollout in the presence of networking slowdowns or irregularities. Geographic distribution should be a part of your automated release pipeline for guaranteed consistency.
- Shift as much of your testing as possible to the left by having as much extensive integration and replay testing of your application as possible. You want to start a rollout only with a release that you strongly believe to be correct.
- Begin a release in a canary region, which is a preproduction environment in which other teams or large customers can validate *their* use of your service before you begin a larger-scale rollout.
- Identify different characteristics of the regions where you are rolling out. Each difference can be one that causes a failure and a full or partial outage. Try to roll out to low-risk regions first.
- Document and practice your response to any problem or process (e.g., a rollback) that you might encounter. Trying to remember what to do in the heat of the moment is a recipe for forgetting something and making a bad problem worse.

Summary

It might seem unlikely today, but most of us will end up running a worldwide scale system sometime during our careers. This chapter described how you can gradually build and iterate your system to be a truly global design. It also discussed how you can set up your rollout to ensure minimal downtime of the system while it is being updated. Finally, we covered setting up and practicing the processes and procedures necessary to react when (note that we didn't say "if") something goes wrong.

CHAPTER 16

Managing State and Stateful Applications

In the early days of container orchestration, the targeted workloads were usually stateless applications that used external systems to store state if necessary. The thought was that containers are very temporal, and orchestration of the backing storage needed to keep state in a consistent manner was difficult at best. Over time the need for container-based workloads that kept state became a reality and, in select cases, might be more performant. Kubernetes adapted over many iterations to not only allow for storage volumes mounted into the pod, but those volumes being managed by Kubernetes directly was an important component in orchestration of storage with the workloads that require it.

If the ability to mount an external volume to the container was enough, many more examples of stateful applications running at scale in Kubernetes would exist. The reality is that volume mounting is the easy component in the grand scheme of stateful applications. The majority of applications that require state to be maintained after node failure are complicated data-state engines such as relational database systems, distributed key/value stores, and complicated document management systems. This class of applications requires more coordination between how members of the clustered application communicate with one another, how the members are identified, and the order in which members either appear or disappear into the system.

This chapter focuses on best practices for managing state, from simple patterns such as saving a file to a network share, to complex data management systems like MongoDB, mySQL, or Kafka. There is a small section on a new pattern for complex systems called Operators that brings not only Kubernetes primitives, but allows for business or application logic to be added as custom controllers that can help make operating complex data management systems easier.

Volumes and Volume Mounts

Not every workload that requires a way to maintain state needs to be a complex database or high throughput data queue service. Often, applications that are being moved to containerized workloads expect certain directories to exist and read and write pertinent information to those directories. The ability to inject data into a volume that can be read by containers in a pod is covered in another chapter; however, data mounted from ConfigMaps or secrets is usually read-only, and this section focuses on giving containers volumes that can be written to and will survive a container failure or, even better, a pod failure.

Every major container runtime, such as Docker, rkt, CRI-O, and even Singularity, allows for mounting volumes into a container that is mapped to an external storage system. At its simplest, external storage can be a memory location, a path on the container's host, or an external filesystem such as NFS, Glusterfs, CIFS, or Ceph. Why would this be needed, you might wonder? A useful example is that of a legacy application that was written to log application-specific information to a local filesystem. There are many possible solutions including, but not limited to, updating the application code to log out to a `stdout` or `stderr` of a sidecar container that can stream log data to an outside source via a shared pod volume or using a host-based logging tool that can read a volume for both host logs and container application logs. The last scenario can be attained by using a volume mount in the container using a Kubernetes `hostPath` mount, as shown in the following:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
    spec:
      containers:
      - name: nginx-webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
          - name: hostvol
            mountPath: /usr/share/nginx/html
      volumes:
        - name: hostvol
```

```
          hostPath:
            path: /home/webcontent
```

Volume Best Practices

- Try to limit the use of volumes to pods requiring multiple containers that need to share data, for example adapter or ambassador type patterns. Use the `emptyDir` for those types of sharing patterns.
- Use `hostDir` when access to the data is required by node-based agents or services.
- Try to identify any services that write their critical application logs and events to local disk, and if possible change those to `stdout` or `stderr` and let a true Kubernetes-aware log aggregation system stream the logs instead of leveraging the volume map.

Kubernetes Storage

The examples so far show basic volume mapping into a container in a pod, which is just a basic container engine capability. The real key is allowing Kubernetes to manage the storage backing the volume mounts. This allows for more dynamic scenarios where pods can live and die as needed, and the storage backing the pod will transition accordingly to wherever the pod may live. Kubernetes manages storage for pods using two distinct APIs, the PersistentVolume and PersistentVolumeClaim.

PersistentVolume

It is best to think of a PersistentVolume as a disk that will back any volumes that are mounted to a pod. A PersistentVolume will have a claim policy that will define the scope of life of the volume independent of the life cycle of the pod that uses the volume. Kubernetes can use either dynamic or statically defined volumes. To allow for dynamically created volumes, there must be a StorageClass defined in Kubernetes. PersistentVolumes can be created in the cluster of varying types and classes, and only when a PersistentVolumeClaim matches the PersistentVolume will it actually be assigned to a pod. The volume itself is backed by a volume plug-in. There are numerous plug-ins supported directly in Kubernetes, and each has different configuration parameters to adjust:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: pv001
labels:
  tier: "silver"
spec:
capacity:
  storage: 5Gi
accessModes:
- ReadWriteMany
persistentVolumeReclaimPolicy: Recycle
storageClassName: nfs
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /tmp
  server: 172.17.0.2
```

PersistentVolumeClaims

PersistentVolumeClaims are a way to give Kubernetes a resource requirement definition for storage that a pod will use. Pods will reference the claim, and then if a `persistentVolume` that matches the claim request exists, it will allocate that volume to that specific pod. At minimum, a storage request size and access mode must be defined, but a specific StorageClass can also be defined. Selectors can also be used to match certain PersistentVolumes that meet a certain criteria will be allocated:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClass: nfs
    accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      tier: "silver"
```

The preceding claim will match the PersistentVolume created earlier because the storage class name, the selector match, the size, and the access mode are all equal.

Kubernetes will match up the PersistentVolume with the claim and bind them together. Now to use the volume, the `pod.spec` should just reference the claim by name, as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
    spec:
      containers:
      - name: nginx-webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
          - name: hostvol
            mountPath: /usr/share/nginx/html
      volumes:
        - name: hostvol
          persistentVolumeClaim:
            claimName: my-pvc
```

Storage Classes

Instead of manually defining the PersistentVolumes ahead of time, administrators might elect to create StorageClass objects, which define the volume plug-in to use and any specific mount options and parameters that all PersistentVolumes of that class will use. This then allows the claim to be defined with the specific StorageClass to use, and Kubernetes will dynamically create the PersistentVolume based on the StorageClass parameters and options:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: nfs
provisioner: cluster.local/nfs-client-provisioner
parameters:
  archiveOnDelete: True
```

Kubernetes also allows operators to create a default storage class using the DefaultStorageClass admission plug-in. If this has been enabled on the API server, then a default StorageClass can be defined and any PersistentVolumeClaims that do not explicitly define a StorageClass. Some cloud providers will include a default storage class to map to the cheapest storage allowed by their instances.

Container Storage Interface and FlexVolume

Often referred to as "Out-of-Tree" volume plug-ins, the Container Storage Interface (CSI) and FlexVolume enable storage vendors to create custom storage plug-ins without the need to wait for direct code additions to the Kubernetes code base like most volume plug-ins today.

The CSI and FlexVolume plug-ins are deployed on Kubernetes clusters as extensions by operators and can be updated by the storage vendors when needed to expose new functionality.

The CSI states its objective on GitHub as:

> To define an industry standard Container Storage Interface that will enable storage vendors (SP) to develop a plug-in once and have it work across a number of container orchestration (CO) systems.

The FlexVolume interface has been the traditional method used to add additional features for a storage provider. It does require specific drivers to be installed on all of the nodes of the cluster that will use it. This basically becomes an executable that is installed on the hosts of the cluster. This last component is the main detractor to using FlexVolumes, especially in managed service providers, because access to the nodes is frowned upon and the masters practically impossible. The CSI plug-in solves this by basically exposing the same functionality and being as easy to use as deploying a pod into the cluster.

Kubernetes Storage Best Practices

Cloud native application design principles try to enforce stateless application design as much as possible; however, the growing footprint of container-based services has created the need for data storage persistence. These best practices around storage in Kubernetes in general will help to design an effective approach to providing the required storage implementations to the application design:

- If possible, enable the DefaultStorageClass admission plug-in and define a default storage class. Many times, Helm charts for applications that require PersistentVolumes default to a `default` storage class for the chart, which allows the application to be installed without too much modification.
- When designing the architecture of the cluster, either on-premises or in a cloud provider, take into consideration zone and connectivity between the compute and data layers using the proper labels for both nodes and PersistentVolumes, and using affinity to keep the data and workload as close as possible. The last thing you want is a pod on a node in zone A trying to mount a volume that is attached to a node in zone B.
- Consider very carefully which workloads require state to be maintained on disk. Can that be handled by an outside service like a database system or, if running in a cloud provider, by a hosted service that is API consistent with currently used APIs, say a mongoDB or mySQL as a service?
- Determine how much effort would be involved in modifying the application code to be more stateless.

- While Kubernetes will track and mount the volumes as workloads are scheduled, it does not yet handle redundancy and backup of the data that is stored in those volumes. The CSI specification has added an API for vendors to plug in native snapshot technologies if the storage backend can support it.
- Verify the proper life cycle of the data that volumes will hold. By default the reclaim policy is set to for dynamically provisioned persistentVolumes which will delete the volume from the backing storage provider when the pod is deleted. Sensitive data or data that can be used for forensic analysis should be set to reclaim.

Stateful Applications

Contrary to popular belief, Kubernetes has supported stateful applications since its infancy, from mySQL, Kafka, and Cassandra to other technologies. Those pioneering days, however, were fraught with complexities and were usually only for small workloads with lots of work required to get things like scaling and durability to work.

To fully grasp the critical differences, you must understand how a typical ReplicaSet schedules and manages pods, and how each could be detrimental to traditional stateful applications:

- Pods in a ReplicaSet are scaled out and assigned random names when scheduled.
- Pods in a ReplicaSet are scaled down in an arbitrary manner.
- Pods in a ReplicaSet are never called directly through their name or IP address but through their association with a `Service`.
- Pods in a ReplicaSet can be restarted and moved to another node at any time.
- Pods in a ReplicaSet that have a PersistentVolume mapped are linked only by the claim, but any new pod with a new name can take over the claim if needed when rescheduled.

Those that have only cursory knowledge of cluster data management systems can immediately begin to see issues with these characteristics of ReplicaSet-based pods. Imagine a pod that has the current writable copy of the database just all of a sudden getting deleted! Pure pandemonium would ensue for sure.

Most neophytes to the Kubernetes world assume that StatefulSet applications are automatically database applications and therefore equate the two things. This could not be further from the truth in the sense that Kubernetes has no sense of what type of application it is deploying. It does not know that your database system requires leader election processes, that it can or cannot handle data replication between members of the set, or, for that matter, that it is a database system at all. This is where StatefulSets come in to play.

StatefulSets

What StatefulSets do is make it easier to run applications systems that expect more reliable node/pod behavior. If we look at the list of typical pod characteristics in a ReplicaSet, StatefulSets offer almost the complete opposite. The original spec back in Kubernetes version 1.3 called `PetSets` was introduced to answer some of the critical scheduling and management needs for stateful-type applications such as complex data management systems:

- Pods in a StatefulSet are scaled out and assigned sequential names. As the set scales up, the pods get ordinal names, and by default a new pod must be fully online (pass its liveness and/or readiness probes) before the next pod is added.
- Pods in a StatefulSet are scaled down in reverse sequence.
- Pods in a StatefulSet can be addressed individually by name behind a headless Service.
- Pods in a StatefulSet that require a volume mount must use a defined Persistent-Volume template. Volumes claimed by pods in a StatefulSet are not deleted when the StatefulSet is deleted.

A StatefulSet specification looks very similar to a Deployment except for the Service declaration and the PersistentVolume template. The headless Service should be created first, which defines the Service that the pods will be addressed with individually. The headless Service is the same as a regular Service but does not do the normal load balancing:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    name: mongo
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None #This creates the headless Service
  selector:
    role: mongo
```

The StatefulSet definition will also look exactly like a Deployment with a few changes:

```yaml
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        role: mongo
        environment: test
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: mongo
          image: mongo:3.4
          command:
            - mongod
            - "--replSet"
            - rs0
            - "--bind_ip"
            - 0.0.0.0
            - "--smallfiles"
            - "--noprealloc"
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-persistent-storage
              mountPath: /data/db
        - name: mongo-sidecar
          image: cvallance/mongo-k8s-sidecar
          env:
            - name: MONGO_SIDECAR_POD_LABELS
              value: "role=mongo,environment=test"
  volumeClaimTemplates:
  - metadata:
      name: mongo-persistent-storage
      annotations:
        volume.beta.kubernetes.io/storage-class: "fast"
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 2Gi
```

Operators

StatefulSets has definitely been a major factor in introducing complex stateful data systems as feasible workloads in Kubernetes. The only real issue is, as stated earlier, Kubernetes does not really understand the workload that is running in the StatefulSet. All of the other complex operations, like backups, failover, leader registration, new replica registration, and upgrades, are all operations that need to happen quite regularly and will require some careful consideration when running as StatefulSets.

Early on in the growth of Kubernetes, CoreOS site reliability engineers (SREs) created a new class of cloud native software for Kubernetes called Operators. The original intent was to encapsulate the application domain-specific knowledge of running a specific application into a specific controller that extends Kubernetes. Imagine building up on the StatefulSet controller to be able to deploy, scale, upgrade, backup, and run general maintenance operations on Cassandra or Kafka. Some of the first Operators that were created were for etcd and Prometheus, which uses a time series database to keep metrics over time. The proper creation, backup, and restore configuration of Prometheus or etcd instances can be handled by an Operator and are basically new Kubernetes-managed objects just like a pod or Deployment.

Until recently, Operators have been one-off tools created by SREs or by software vendors for their specific application. In mid-2018, RedHat created the Operator Framework, which is a set of tools including an SDK life cycle manager and future modules that will enable features such as metering, marketplace, and registry type functions. Operators are not only for stateful applications, but because of their custom controller logic they are definitely more amenable to complex data services and stateful systems.

Operators are still an emerging technology in the Kubernetes space, but they are slowly taking a foothold with many data management system vendors, cloud providers, and SREs the world over who want to include some of the operational knowledge they have in running complex distributed systems in Kubernetes. Take a look at OperatorHub for an updated list of curated Operators.

StatefulSet and Operator Best Practices

Large distributed applications that require state and possibly complicated management and configuration operations benefit from Kubernetes StatefulSets and Operators. Operators are still evolving, but they have the backing of the community at large, so these best practices are based on current capabilities at the time of publication:

- The decision to use Statefulsets should be taken judiciously because usually stateful applications require much deeper management that the orchestrator cannot really manage well yet (read the "Operators" on page 21 section for the possible future answer to this deficiency in Kubernetes).
- The headless Service for the StatefulSet is not automatically created and must be created at deployment time to properly address the pods as individual nodes.
- When an application requires ordinal naming and dependable scaling, it does not always mean it requires the assignment of PersistentVolumes.
- If a node in the cluster becomes unresponsive, any pods that are part of a StatefulSet are not not automatically deleted; they instead will enter a `Terminating` or `Unkown` state after a grace period. The only way to clear this pod is to remove the node object from the cluster, the kubelet beginning to work again and deleting the pod directly, or an Operator force deleting the pod. The force delete should

be the last option and great care should be taken that the node that had the deleted pod does not come back online, because there will now be two pods with the same name in the cluster. You can use `kubectl delete pod nginx-0 --grace-period=0 --force` to force delete the pod.

- Even after force deleting a pod, it might stay in an `Unknown` state, so a patch to the API server will delete the entry and cause the StatefulSet controller to create a new instance of the deleted pod: `kubectl patch pod nginx-0 -p '{"metadata":{"finalizers":null}}'`.
- If you're running a complex data system with some type of leader election or data replication confirmation processes, use `preStop hook` to properly close any connections, force leader election, or verify data synchronization before the pod is deleted using a graceful shutdown process.
- When the application that requires stateful data is a complex data management system, it might be worth a look to determine whether an Operator exists to help manage the more complicated life cycle components of the application. If the application is built in-house, it might be worth investigating whether it would be useful to package the application as an Operator to add additional manageability to the application. Look at the CoreOS Operator SDK for an example.

Summary

Most organizations look to containerize their stateless applications and leave the stateful applications as is. As more and more cloud native applications run in cloud provider Kubernetes offerings, data gravity becomes an issue. Stateful applications require much more due diligence, but the reality of running them in clusters has been accelerated by the introduction of StatefulSets and Operators. Mapping volumes into containers allow Operators to abstract the storage subsystem specifics away from any application development. Managing stateful applications such as database systems in Kubernetes is still a complex distributed system and needs to be carefully orchestrated using the native Kubernetes primitives of pods, ReplicaSets, Deployments, and StatefulSets, but using Operators that have specific application knowledge built into them as Kubernetes-native APIs may help to elevate these systems into production-based clusters.

About the Authors

Brendan Burns is a distinguished engineer at Microsoft Azure and cofounder of the Kubernetes open source project. He's been building cloud applications for more than a decade.

Eddie Villalba is a software engineer with Microsoft's Commercial Software Engineering division, focusing on open source cloud and Kubernetes. He's helped many real-world users adopt Kubernetes for their applications.

Dave Strebel is a global cloud native architect at Microsoft Azure focusing on open source cloud and Kubernetes. He's deeply involved in the Kubernetes open source project, helping with the Kubernetes release team and leading SIG-Azure.

Lachlan Evenson is a principal program manager on the container compute team at Microsoft Azure. He's helped numerous people onboard to Kubernetes through both hands-on teaching and conference talks.

Colophon

The animal on the cover of *Kubernetes Best Practices* is an Old World mallard duck (*Anas platyrhynchos*), a kind of dabbling duck that feeds on the surface of water rather than diving for food. Species of Anas are typically separated by their ranges and behavioral cues; however, mallards frequently interbreed with other species, which has introduced some fully fertile hybrids.

Mallard ducklings are precocial and capable of swimming as soon as they hatch. Juveniles begin flying between three and four months of age. They reach full maturity at 14 months and have an average life expectancy of 3 years.

The mallard is a medium-sized duck that is just slightly heavier than most dabbling ducks. Adults average 23 inches long with a wingspan of 36 inches, and weigh 2.5 pounds. Ducklings have yellow and black plumage. At around six months of age, males and females can be distinguished visually as their coloring changes. Males have green head feathers, a white collar, purple-brown breast, gray-brown wings, and a yellowish-orange bill. Females are mottled brown, which is the color of most female dabbling ducks.

Mallards have a wide range of habitats across both northern and southern hemispheres. They are found in fresh- and salt-water wetlands, from lakes to rivers to seashores. Northern mallards are migratory, and winter father south. The mallard diet is highly variable, and includes plants, seeds, roots, gastropods, invertebrates, and crustaceans.

Brood parasites will target mallard nests. These are species of other birds who may lay their eggs in the mallard nest. If the eggs resemble those of the mallard, the mallard will accept them and raise the hatchlings with their own.

Mallards must contend with a wide variety of predators, most notably foxes and birds of prey such as falcons and eagles. They have also been preyed upon by catfish and pike. Crows, swans, and geese have all been known to attack the ducks over territorial disputes. Unihemispheric sleep (or sleeping with one eye open), which allows one hemisphere of the brain to sleep while the other is awake, was first noted in mallards. It is common among aquatic birds as a predation-avoidance behavior.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black and white engraving from *The Animal World*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.