SYSADMIN

How Kubernetes Changes Operations

BRENDAN BURNS

Brendan Burns is a Senior Staff Software Engineer at Google, Inc. and a founder of the Kubernetes project, leading engineering efforts to make the Google Cloud Platform the best place to run containers. He also has managed several other cloud teams, including Deployment Manager, Managed VMs, and Cloud DNS. Prior to Cloud, he was a lead engineer in Google's Web search infrastructure, building backends that powered social and personal search. Prior to working at Google, he was a professor at Union College in Schenectady, NY. He received a PhD in computer science from the University of Massachusetts Amherst, and a BA in computer science and studio art from Williams College. bburns@google.com

Container cluster managers are used by many Web-scale Internet companies, including Google's Borg and Omega, Facebook's Tupperware, Twitter's Aurora, and many others. At their core, these container orchestration systems schedule and manage ("orchestrate") collections of Linux application containers. In this article, I will explain the Kubernetes project.

Recently, interest in the Docker open source project has caused a significant growth in interest in Linux application containers in the general developer and operations community. Due to this growth in interest, Google launched the Kubernetes project, which makes Google's years of experience in running container clusters available to the larger world in a community-driven, open source project. The development of these internal container cluster managers was driven by real operational needs of operating software at "Google scale," but we have seen recently that their benefits apply even at a more modest scope and scale.

I illustrate how container orchestration systems change the operations tasks associated with running, maintaining, and upgrading highly scalable and reliable applications. At the heart of this change are two fundamental shifts. First, container orchestration systems provide and enforce significant decoupling between the layers of the serving stack: machine, operating system, application manager, and application code. This decoupling enables the development of specialized teams with agility and freedom to operate on their parts of the stack, thanks to separation of concerns. Second, container cluster APIs are inherently more application-oriented than traditional IaaS machine-centric APIs. This shift towards application-oriented primitives makes it easy to perform operation and maintenance tasks that were previously complicated, brittle, or both. In this article, I show how the formal boundaries introduced by containers and container cluster management enable the segmentation of traditional operations into multiple discrete roles.

In addition to a general discussion of container orchestration and operations, I also describe the Google Kubernetes container orchestrator, including the core resources in the Kubernetes system, and how they produce an inherently more stable, agile, and reliable foundation for application deployment.

Decoupling Operations Roles

Anyone who has tried to back up a trailer on a car knows that coupled, multi-component systems are hard to predict and control. Actions taken in one part of the system often cause unpredictable, user-visible problems in some other component of the system. A classic example might be upgrading a Web server, which includes updating the libc library, causing a database on the same machine to fail because the libc change introduces a bug that the database triggers.

Coupling increases the knowledge and skill set required to be a high-performing application administrator and requires operators to fully understand their entire application stack, including all dependencies, in comprehensive detail. In turn, this reduces the ability of operations teams to specialize, prevents the acquisition of true expertise, and reduces opportunities to introduce economies of scale.

As an example of this, in companies where every development team is responsible for their choice of operating system distribution (e.g., Debian or Red Hat), the operating systems in the fleet will inevitably be heterogeneous. Another example involves choosing to use SysV init vs. the systemd daemon. The resulting heterogeneity makes it difficult (if not impossible) to have a single team of administrators manage all of the machines in the fleet. It is also difficult to build a common set of tools and/or processes for performing maintenance and monitoring across all of the operating systems in the fleet. Being unable to share tooling and expertise means that fleet maintenance is more expensive and less reliable than if a single team and set of tools could manage the entire fleet of machines.

Container cluster management software makes it easier to avoid tight coupling, and the corresponding problems of heterogeneous environments, by introducing crisp boundaries and management APIs that decouple operations into discrete roles: hardware operations, kernel/OS operations, cluster operations, and application operations. The decoupling of these roles means that it is possible for each of the first three roles (hardware, kernel/OS, and cluster) to have a single team handle operations and administration, which enables lower costs and higher reliability. For application operations, it also enables the building of specialized, application-specific operations teams that can be deeply involved in the specifics of their application. The net result is a complete system that makes highly reliable applications cheaper to build and maintain.

Hardware Operations

The hardware operations role is responsible for racking and stacking machines, connecting network cables between racks and switches, and repairing or retiring machines. In modern public cloud providers, these roles have been wholly outsourced to the cloud provider, who can provide significantly greater expertise and economies of scale than the average user.

Kernel/OS Operations

The interface between a Docker container image and the underlying operating system is the Linux kernel syscall interface. Because each Docker container carries with it all of its dependencies (application binary, libraries, configuration files, etc.), it is wholly decoupled from the files that make up the machine image. An application developer can rely on two things from the kernel and operating system:

- Stability in the syscall API and operational characteristics
- A working Docker daemon

These requirements form an explicit contract between the kernel/OS and the applications that run on top of it. This means that the operations team responsible for the machine image (kernel, operating system able to boot the Docker daemon) can focus on qualifying those two generic requirements without understanding the details of any particular application. This decoupling enables release qualification, rollout, and management of a single, homogeneous kernel/OS across an entire fleet of machines. In managed container services like Google Container Engine, this kernel/operating system qualification and upgrading is outsourced to the cloud provider, enabling the application operations team (described below) to focus entirely on their application. The cluster management boundary imposes a discipline about the APIs available to application developers, as well as a single, shared implementation of this API. Because the implementation is shared between multiple, different applications, the discipline enforced by this API also acts as a counterweight to the natural tendency towards entropy and differences between the software stack supporting different applications.

Cluster Operations

If cluster users are allowed to deploy their container applications onto specific machines, then the resulting systems will be too tightly coupled because the applications will inevitably begin to rely on the specific characteristics of the machines on which they run. For example, if an application is coupled to the machine's network identity (hostname and IP address), the decoupling between application, hardware, and kernel has been broken. That machine cannot just be sent to repairs when the hardware operator determines it is failing. Nor can it be rebooted for an OS installation any time the OS operator decides one is needed. It is the container cluster manager's goal to decouple containerized applications from the specifics of any particular machine. For example, in Kubernetes, we give each pod an IP address that is independent of the IP address of the machine that it is running on. The pod does not have access to the machine's network identity. Furthermore, Kubernetes can restrict the set of file systems that can be mounted into a pod from the host file system, restricting access to things like raw block devices and other machine-specific hardware.

Additionally, container cluster managers, like Kubernetes, provide a declarative, programmable API that is the primary one by which developers schedule and deploy users' applications onto a fleet of machines. Consequently, developers are decoupled from the details of physical machines, because their mode of interaction is container and application-centric. The particular details of the machine that ends up running the application developer's containers become an implementation detail of the underlying cluster manager.

Indeed, many users forget that their applications are running on physical (or virtual) machines at all and, instead, deal solely with the logical compute substrate provided by the container cluster API. They ask that API for a certain set of application resource requirements (say, two cores and 100 GB of RAM), and it is the

container cluster manager's responsibility to find sufficient resources somewhere in the cluster and deploy the application onto those resources. The job of a container cluster administrator is to ensure that the services that provide the container cluster management API stay available and operationally healthy at all times.

Application Operations

Closest to the end user in these decoupled operations roles are application operators. These administrators are focused on managing and deploying applications: for example, the Google search backend or Gmail frontend. These administrators develop deep specialized knowledge of their applications, and rely on cluster, kernel, and hardware operations teams to provide them the infrastructure they need to do their job. Transferring work that is unrelated to their application (e.g., kernel and OS upgrades) onto specialized kernel operation teams allows the application operation teams to develop application-specific tooling for more reliable management of their application. The specialization of application administrators on a particular application also means that they can develop deep technical understanding of the specific application software, and form significant partnerships with the development teams to improve the reliability and performance of that software.

The Kubernetes Cluster Manager

Having described how containers and cluster management APIs enable the decoupling of operations roles, I will now discuss some specifics of the Kubernetes API to provide a deeper understanding of the functionality that Kubernetes provides. Beginning with a description of pods, the atomic unit of scheduling in the Kubernetes system and the basic building block for running containers in a Kubernetes cluster, I will go on to cover generic software patterns for building applications with pods. I'll show how Kubernetes resources are organized into dynamic sets with labels and how those labels are used to automatically manage replicated microservices using Replication Controllers and Services.

Pods

Pods are the most fundamental API object in Kubernetes. A pod is a group of containers that is scheduled together onto one machine. All of the containers within a pod share the same network namespace, so the containers within a pod can easily find each other on "localhost." This eliminates the need for a complicated discovery service (more on that later). The containers in a pod also share the same IPC namespace, which means that they can use traditional UNIX IPC, such as pipes. As Kubernetes matures, we expect that pods will come to share all of the available kernel namespaces, including group ID namespaces, process ID namespaces, and more.

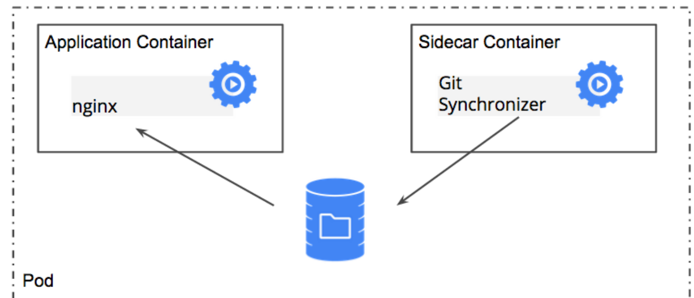

Figure 1: Example of a sidecar container: a pod where an Nginx Web server is being augmented by a Git synchronizing container

Pods also encapsulate node-level health checking and reliability for their constituent containers. In Kubernetes, there are two different types of checks:

First, each container has a *liveness* check. By default, this is a simple process-based one ("is the process running"), but it can be extended to include several other application-specific health checks: *HTTP* (healthy if the container endpoint returns an HTTP 200), *TCP* (healthy if a TCP socket can be opened), or *exec* (healthy if a user-supplied binary executed in the context of the container returns an exit code of zero). If any liveness test fails, the container is automatically restarted by Kubernetes.

The second check is a *readiness* check, which is applied to an entire pod. Readiness checks indicate whether the pod is ready to serve end-user traffic. In many situations, a pod may take some time to start up, due to network downloads, migrations, or other long, computational initialization steps. During this time, the pod is *alive*: it should not be restarted by Kubernetes. However, it is not *ready*: it should not serve traffic. Readiness checks are used to implement service load balancers, described below.

Pod Patterns

When you start using pods, some general patterns naturally start to recur. The three common ones are sidecar containers, ambassador containers, and adapter containers.

SIDECAR CONTAINERS

Sidecar containers extend and enhance the "main" container; they take existing containers and make them better.

As an example, consider a container that runs the Nginx Web server. Add a different container that syncs a directory with a Git repository, share the file system between the containers, and you have built a non-atomic, push-to-deploy Git. But you've done it in a modular manner where the Git synchronizer can be built by a different team and reused across many different Web servers (Apache, Python, Tomcat, etc.). Because of this modularity, you only have to write and test your Git synchronizer once to reuse it across numerous apps. If someone else writes it, you don't even need to do that.

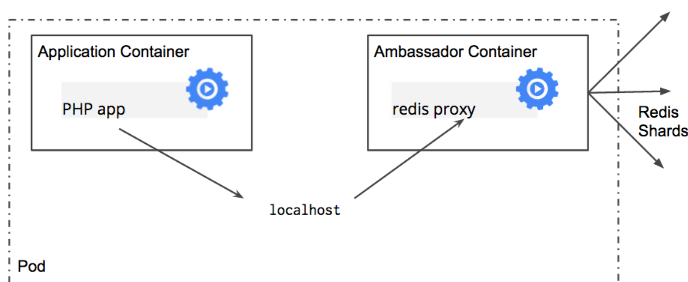

Figure 2: Example ambassador container: a pod where a Redis proxy ambassador is used to proxy connections from a PHP application to a set of Redis shards

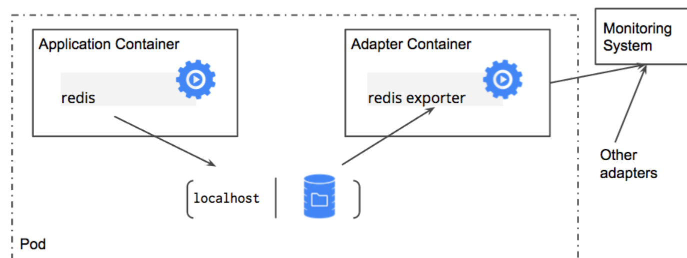

Figure 3: Example of an adapter container: a pod where the Redis key-value store is adapted to provide a consistent monitoring interface (e.g., https://github.com/oliver006/redis_exporter)

AMBASSADOR CONTAINERS

Ambassador containers proxy the outside world via a local connection in the same pod.

As an example, consider a Redis cluster with read replicas and a single write master. You can create a pod that groups your Redis client with a Redis ambassador container. The ambassador is a proxy; it is responsible for splitting reads and writes to Redis and sending them on to the appropriate Redis servers. Because these two containers share a network namespace, they share an IP address, and your application can open a connection on "localhost" and find the proxy without any service discovery. Note that this is "localhost" for the network of the pod, *not* "localhost" on the host machine.

ADAPTER CONTAINERS

Adapter containers standardize and normalize output.

In any real-world application, the application's software comes from a heterogeneous set of sources (open source, off-the-shelf software, home brew), and monitoring system developers cannot be expected to understand, build, maintain, and deploy for all of them. Consequently, you often need to wrap applications to enable communication with auditing or monitoring services.

Using a modular *adapter* container co-located in the same pod as your application gives you a simple unit of deployment that combines both application and adapter. Using adapters enables each application developer to supply a common interface. The modularity of using two different containers (the application and the adapter) means that despite making the adapter the application owner's responsibility, adapters can be reused (e.g., a Java JMX adapter).

The adapter pattern creates pods that group the application containers with adapters that know how to do the transformation. Again, because these pods share namespaces and file systems, the coordination of these two containers is simple and straightforward.

Labels

Experience operating large, complicated systems has taught us that requiring applications and their parts to be grouped into fixed, disjoint sets is overly restrictive.

As an example of this, consider the canonical search stack. There is a set of replicas that are responsible for serving end-user requests {frontend, middleware, backend servers}, and then there are the jobs that are responsible for building, pushing, and loading a new search index {crawler, index-builder, backend servers}. The presence of "backend servers" in both of these organizations reflects the problem with fixed sets. We need an organizational mechanism that can flexibly represent both of these organizational sets (and any other useful sets). If the cluster management infrastructure can't represent the overlapping sets of organizations that are present in the cluster, then additional tooling, which is opaque to the cluster manager, will get built to represent these organizations. The additional complexity required to make these systems interact well with the cluster management software makes the system harder to maintain and extend.

Additionally, we need a representation that is dynamic. For example, at different times, pods may be added or removed from sets; during a rolling update of a service to a new version of its software, pods are dynamically added and removed from the set of backends of a load balancer. We need a representation that can easily capture this dynamism without requiring constant action from the user to maintain these sets.

In Kubernetes, *labels* and *label queries* provide flexible, dynamic sets of resources. Rather than encode any specific grouping primitive into the Kubernetes API, every resource in the Kubernetes API can have labels attached to that resource. These labels are arbitrary, key-value pairs that help define the object. For example, a production Web server might have the labels `{role=frontend, stage=production, version=v1, machine=m1, rack=r2}`, and a production backend might have the labels `{role=backend, stage=production, version=v1, machine=m2, rack=r1}`.

A label query dynamically organizes objects into a group by constructing a set of objects that match its conditions. For example, we might query "`stage=production`" to see all production pods, "`rack=r2`" to see all containers on a particular rack, or even conjugate queries like "`stage=production, machine=m1`" for all production jobs on a particular machine. Label queries are used to list particular RESTful resources in the Kubernetes API. A label query for a resource of a particular type (e.g., pods) will only return the pods whose labels match the query.

Reconciliation

The third key concept in Kubernetes, after pods and labels, is reconciliation loops.

The basic premise is that there are three states of the world: an idealized *desired state*, which is a declarative statement of what the world should be like; a *current state*, which approximates the actual state, and might be noisy, incomplete, or out of date; and an *actual state*. Unfortunately, the actual state isn't directly observable, thanks to the vagaries of distributed systems, delays, and failures, so we must make do with the observed state.

The role of the reconciliation loop is to repeatedly compare the current state against the desired state, and take action to drive the actual state to match the desired state. This is just a control loop, like the one in your thermostat. It is what transforms Kubernetes into a self-healing, dynamic system, by automatically causing it to restore the system to the desired state without needing operator intervention. Only if this fails does the system need to invoke help from an administrator, e.g., by triggering an alert.

Replication Controllers

In any real production system, replicating the components in the system is the only way to achieve reliable operation. Each replica is an independent unit of failure, and thus, multiple replicas reduce the probability of a total failure. They also allow a service to be scaled up as traffic grows. However, the complexity of managing a replicated system must not be linear in the number of replicas, or else the system is not truly scalable.

In Kubernetes, replication controllers provide an API for managing replicated sets of pods. Replication controllers use a pod template, a label query, and a desired number of replicas to create a replicated set of pods. The operation is as follows:

```
Repeat forever
   1. Select pods matching Label Query.
   2. Subtract number of pods found from the desired number of
replicas.
   3. If this difference is negative, destroy a pod.
   4. If this difference is positive, create a pod using the pod
Template.
```

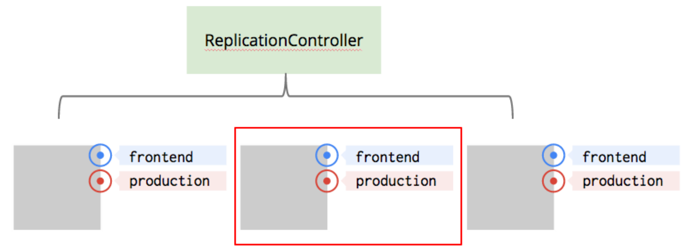

Figure 4. A replicated set of pods with a misbehaving replica (pod within rectangle). Solid boxes are pods; circles indicate labels attached to them.

Note that this is a reconciliation loop. No matter why a pod disappears—whether due to node failure, accidental deletion, or network partition—the replication controller attempts to ensure that the correct number of replicas exists. Likewise, if a user or automated process resizes the number of replicas up or down, these adjustments to the number of replicas are also materialized by this simple reconciliation loop.

Services

A recent, popular trend in distributed systems is microservice architectures, which decouple different pieces of a distributed system into independently managed and scaled microservices. This decoupling helps microservice architectures to be reliable and scalable.

In Kubernetes, the *Service* API object represents a load balancer for a microservice. Like replication controllers, services are based on a dynamic label query that identifies the set of backends that the service connects to.

To enable service discovery, a service is assigned a static virtual IP address (VIP). This address is constant, and has the same lifespan as the service. Consequently, the VIP can be populated into DNS for service discovery. Because the VIP is not the address of any particular pod, the VIP can be kept constant, even as pods are scaled up or down behind the service.

Kubernetes itself ships with a simple, default load-balancer implementation, but the Kubernetes API also makes *Endpoint* objects available. These endpoints are the current members of the service's load balancing group—i.e., the pod IP addresses across which it spreads incoming requests. Advanced users can use these endpoints to populate a third-party load balancer (e.g., Nginx, HAProxy) or even to implement thick clients that do balancing without a proxy.

The maintenance of the service's endpoints is another example of a reconciliation loop. In this case, the loop looks like:

```
Repeat forever
   1. Select pods matching Service Selector Label Query
   2. For each matching pod
      a. If the pod is Ready (see 'Readiness Checks' above)
         i. Add the pod to the Endpoint set for this Service
```

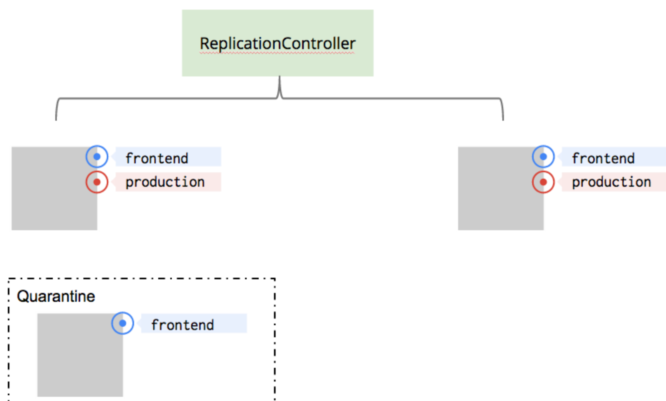

Figure 5. After the "production" label is removed from the misbehaving replica, the replica is now quarantined.

Operations in Kubernetes

It is easier to operate systems that are deployed into a Kubernetes cluster than systems deployed into traditional virtual machines. This section describes two operations scenarios that demonstrate this.

Quarantining a Replica

One of the common tasks that occur in operations is quarantining a misbehaving replica of an application. Oftentimes, sadly, this means simply killing the misbehaving replica, collecting logs for retrospective analysis, and restarting the process. While this restores the service to health quickly, it is much harder to debug a problem from (possibly incomplete) logs than it is with a running server. It would be better to remove a misbehaving replica from the service but maintain it as a running server so that it can be debugged. This is precisely what Kubernetes services and labels allow. This is illustrated in the following example.

We start with an existing Kubernetes replicated service that shares load across three pods. The pod in the middle is determined to be misbehaving.

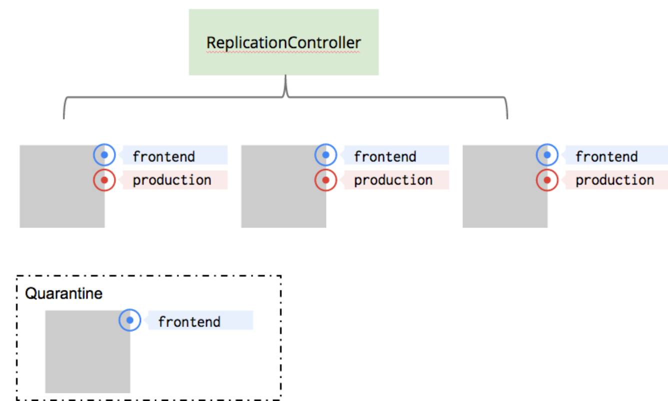

Figure 6. The replication controller replaces the misbehaving pod with a new replica.

The operator removes the "production" label from the misbehaving pod. Because Kubernetes dynamically queries label selectors, the pod is now removed from the corresponding ReplicationController and the service.

The reconciliation loop in `ReplicationController` detects that a pod is missing from the replica set and creates a new pod, restoring the service to full health. The misbehaving pod is retained for future debugging.

Rolling Update

Another common operation is rolling out new software. Kubernetes achieves this through manipulating replication controllers and labels.

At the start of the update, there is a single replication controller. It has three replicas, and is using version 1 (v1) of the application. There is also a Kubernetes Service that is defined to serve traffic to pods with the "frontend" and "production" labels. To perform a rolling update, a second replication controller is created. This replication controller is identical to the first replication controller in all ways, except the image in its template has been updated to version 2 (v2). Initially, the desired replica count for this controller is set to zero (Figure 7).

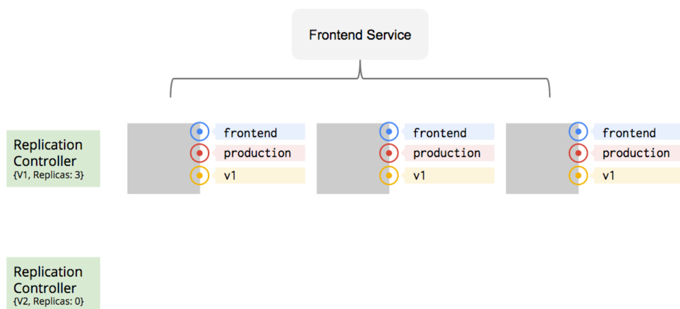

Figure 7. The initial state of the rolling update. A second replica controller has been created but has no replicas yet.

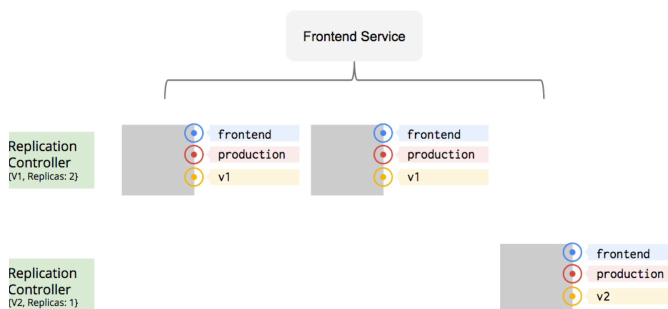

Figure 8. The first "canarying" step of a rolling update: replica count for the original controller is set to two, and to one for the second controller.

To perform the rolling update, the desired number of replicas on the v1 replication controller is dialed down by one (in this case, to two replicas), and the desired replicas for the v2 replication controller is increased by one (to one replica, Figure 8).

This process of one up, one down proceeds until the desired number of replicas for v1 is zero and the desired number of replicas for v2 is three. Because the Kubernetes Service is defined by the label query `{role=frontend, stage=production}`, which ignores the version, the load balancer seamlessly spreads traffic across version 1 and version 2 as the rollout proceeds. If failures occur during the rollout, and a rollback is necessary, it is simple to reverse the roles of the replication controllers and restore the number of replicas for v1 to be three.

Conclusion

Containers have grown in popularity because they decouple user applications from the underlying operating system/kernel, and allow the development of kernel/OS-specific operations teams. Container cluster orchestration systems, like Kubernetes, further allow the decoupling of operations into hardware operations, kernel operations, cluster operations, and application operations. This decoupling enables specialization and focus, which increases the reliability and scalability of those operations teams. Furthermore, Kubernetes provides a set of objects that makes it easier for application developers to design and develop services that are easier to operate and scale. Container cluster management systems are the backbone of most large-scale Web service companies, and with the advent of open source solutions like Docker and Kubernetes, we believe there is an industry-wide shift underway to this new style of decoupled infrastructure.

Acknowledgments

I would like to thank John Wilkes, Jessie Yang, Robert van Gent, and Seth Hettich for providing significant feedback and revisions to this article.

Do you know about the USENIX Open Access Policy?

USENIX is the first computing association to offer free and open access to all of our conferences proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your financial support plays a major role in making this endeavor successful.

Please help to us to sustain and grow our open access program. Donate to the USENIX Annual Fund, renew your membership, and ask your colleagues to join or renew today.

www.usenix.org/annual-fund