Kubernetes

The Early Cloud

In order to see clearly the benefits containers can give we're going to contrast a containerized deployment with a standard cloud infrastructure deployment that does not use any containers. A well automated cloud environment will still require the use of a number of different technologies. Fortunately, every role can be adequately filled with open source software.[12] See the table below for a list of each role and some corresponding software that can used to meet the needs of that role.

[12] While there are many closed source or otherwise non-free tools that are functional and usable, I want to ensure this book remains relevant to those on a shoestring budget.

Role / Function	Solution (non-container)	Solution (Kubernetes)
Provisioning of Virtualized Hardware / Cloud Components	Terraform, CloudFormation(AWS)	Kubernetes
Building Disk Images	Packer, Vagrant, VirtualBox	Docker
Installing and Maintaining Software	Puppet, Ansible, Chef	Docker
Monitoring Software and Hardware	Nagios, Icinga, Munin, Zabbix, Logstash, rsyslog, Grafana, Sysdig	Kubernetes, Prometheus, Sysdig
Integrating, Testing, and Deploying Changes	Jenkins, Buildbot	Jenkins

In what I'll self-servingly call the "old style", we all set up physical or virtual machines, assigned various roles and tasks to these machines, then ran programs to configure and maintain these machines based on the assigned role. Various support systems were set up for tasks like monitoring state and uptime, storing and accessing files on the network, and testing and deploying changes. Each of these systems typically required a unique and distinct role, and figuring out how to effectively and economically match computing resources to roles was a hard problem. One standard approach was to buy specialized hardware for each role, such as fast CPU high MEM boxes

for databases, and fast CPU large cache boxes for load balancing. This presented a number of problems as resources and needs were frequently mismatched. This caused overall usage to drop, and limited effective re-use of existing resources. Additionally, when your total set up time for a new node is 15 minutes, you can only respond to traffic spikes with at least a 15 minute delay. This generally results in running extra capacity to help absorb the 15 minutes of delay during a traffic spike. Finally, you can't control when hardware errors will occur, so all mission critical services must be run across multiple hardware nodes, to ensure that a single node failure can't shut down your entire system. Once you have multiple nodes, you need a way to balance traffic among them, then you need a way to balance your traffic balancers. At the end of this road, there is a great deal of specialized hardware set up for specific purposes. Capacity planning can make or break your ability to scale and stay online!

Contrast this with the Kubernetes style, which has first class resource types for many of the above specialized resources. Wherever you might need redundancy in the "old style", Kubernetes usually provides a mechanism for the indirection. For example, instead of running many load balancers to balance traffic across many physical nodes to achieve redundancy, Kubernetes provides the powerful `Service` abstraction. Kubernetes `Services` are capable of routing traffic in just about any way you require. Where before you had lots of spare capacity and specialized hardware/VMs, Kubernetes emphasizes use of larger clusters of general purpose hardware/VMs. This means you can pack tasks onto each node more effectively and get better resource usage and therefore lower costs. Additionally, the greatly increased startup speed of Kubernetes `Pods` (compared with provisioning entire VMs) means we can squeeze more usage from each node without sacrificing uptime. Kubernetes makes this easy by providing autoscaling and robust traffic routing.

Because Kubernetes emphasizes redundancy, nearly any resource can be easily distributed across multiple physical nodes. Kubernetes is built to be robust to node failure, and many of the complexities we'll face are a result of decisions we need to make about how to handle node failure. In the old style, you might need to think through and face many aspects of node failure directly and explicitly. In Kubernetes, most of these decisions are implicitly handled by choice of Kubernetes resource types. Instead of thinking about how to get back to a functioning system after experiencing node failure, Kubernetes wants you to consider that all nodes are volatile and allow it to handle matching workloads to nodes for you. The emphasis is on providing the end state and allowing Kubernetes to find the best way to get to that desired state. This means that when node

or process failure occurs, Kubernetes can automatically detect and respond to get back to our desired state.

Kube Cluster Basics

Overview

In this section we will learn a high level overview of the parts of Kubernetes and how they interact. Because this information is all available in much greater and more up-to-date detail in the official Kubernetes documentation we're going to approach this section from a functionality perspective. If you need exhaustive lists of the resource types and what attributes they support, visit the Kubernetes documentation here `https://withku.be/links/kube-docs`

There are a number of basic Kubernetes resource types, but don't fret, it's rare that any deployment would use every resource type, and only a handful of resource types are broadly relevant. We're going to focus first on these widely applicable resource types: `Pods`, `Services`, `Deployments/StatefulSets`, `Volumes`, and `ConfigMaps/Secrets`. The other resources are important, we'll just cover them in a more ad-hoc manner later, when we're more equipped to understand what they're doing.

Pods

One of the most basic primitives within Kubernetes, a `Pod` is one or more containers running in a partially shared environment. The `Pod` is the most atomic reusable component of our Kubernetes object definitions. The shared networking environment allows different containers running inside the same `Pod` to communicate over the shared `localhost` without additional setup or configuration. Allowing more than one container to operate in this shared resource environment enables what we call sidecar containers. A typical `Pod` setup with multiple containers might have one primary container running a web server process, replying to connections on some specific port, while another secondary container, the sidecar, could simply watch a folder for configuration file changes and send a reload signal to the primary server process when necessary. Constructing our `Pods` in this way lets us make, use, and share simple modular containers for common tasks. See the list below for some examples broadly reusable containers.

Pause - gcr.io/google_containers/pause

: This container simply waits until it receives the quit signal and then exits. These are most commonly used as a low resource way

to make sure all of a `Pod`'s resources have been acquired before trying to initialize other containers in the `Pod`.

Config Reloader - github.com/coreos/prometheus-operator/tree/master/contrib/prometheus-config-reloader

watches a configuration folder for changes and sends a reload signal via http

Busybox - gcr.io/google_containers/busybox

small simple bash-like environment useful for debugging network connectivity or other I/O issues. Can also be useful for doing any manual volume management tasks that may be required, like formatting.

Services

The basic networking primitive, a service allows you to route traffic to specified `Endpoints`. They can also be used to track and expose a list of `Endpoints`, without performing the routing functions. This is useful for doing service discovery on internal services. `Services` can be configured in a number of ways, and there are four primary service types:

`ClusterIP` - A cluster internal IP address is created for the service, as well as internal dns entries mapping to this internal IP. Traffic from within the cluster being routed to this IP will be forwarded to an endpoint as specified by the service routing rules. [13]

`NodePort` - A specified port is opened on the Nodes, and they listen for any traffic on this port and then forward it to `Endpoints` as with the `ClusterIP` mode.

`LoadBalancer` - Can have different end results depending on how the Kubernetes cluster is setup, but the goal is to allocate some form of public address and then map it to the `Endpoints` as in the other modes. Unlike `ClusterIP` or `NodePort`, creating a `LoadBalancer` type service always requires talking to some external resources to allocate and finalize the service. For instance, on Google Cloud allocating a `LoadBalancer` type service automatically allocates a public IP address and creates routing rules to allow traffic to this service.[14]

`ExternalName` - This service type is unique in that it does not create any internal proxying rules. Instead, it only creates a CNAME record within the internal DNS so that requests for this service name map to the specified record.

[13] Setting the type to `ClusterIP` and providing a "none" value for the IP address will cause kube-dns to construct records for each individual endpoint. This can be useful for relying on kube-dns for service discovery, allowing us to expose a list of available `Endpoints` without any auth required as might be necessary when querying the Kubernetes api for an endpoint list. Using `ClusterIP` in this way is relatively uncommon.

[14] specifically, `LoadBalancer` type `Services` interact closely with `Ingress` resources to request and manage these public IPs, and an Ingress Controller will perform the heavy lifting

Once traffic has arrived to the service using one of the above methods, it needs a place to go. This is where endpoint selection and routing rules come into effect. `Endpoints` are selected by specifying label selectors, such as "app: my-nginx-webserver". Implementing specific routing rules depends on which proxy-mode your cluster is using, and is beyond the scope of this class. At present, more information can be found here https://kubernetes.io/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies

`Deployments` & `StatefulSets`

`Deployments` are a collection of `Pods` (defined as `Pod` templates) and some additional parameters describing the desired number of instances to run, triggers to restart, add, remove `Pods`, and other management operations. In short, `Deployments` let you take a bunch of `Pods` and bundle them into a trackable service deployment. `Pods` within `Deployments` are not given any kind of unique identifier or startup order, which can impose difficulties for processes that aren't designed to be stateless.

`StatefulSets` are effectively the same as `Deployments` with one important difference: every `Pod` gets a unique index number assigned to it and is able to access this number. We can use the index to define behaviors based on position within the startup order. For example, we might want to start a cron daemon in the background, but never on more than one `Pod`. Using a `Deployment` we would need to use some kind of external resource to manage this requirement and keep track of which `Pods` are responsible for executing which tasks. Using a `StatefulSet` we can just check if the index of our `Pod` is equal to a specific number, and know what the role of that `Pod` should be. `StatefulSets` guarantee you'll never have two `Pods` with the same index number, so you can simply query your index from within the `Pod` by examining the hostname. A sample shell script demonstrating this is below:

Listing 10: `entrypoint.sh`

```
#!/bin/sh

POD_INDEX=${HOSTNAME##*-}
if [[ "$POD_INDEX" = "0" ]]; then
# run cron along with the webserver
#  inside the first pod in the set
  /bin/crond -b
fi

```

```
10 python3 "${APP_DIR}/server.py"
```

`Deployments` and `StatefulSets` are very useful as the high level service target and we will continue to explore their key benefits and gotchas through this book. Instead of enumerating the details in depth here, we will be using them throughout this book and will introduce additional complexity as necessary. Just remember that one of these two resources will serve as the backbone for nearly all long-lived processes we'll set up.

Volumes

Sometimes your process needs to store or retrieve data from a disk or disk-like object. `Volumes` are an abstraction for disk-like objects. There are several `Volume` types you may use depending on what is required in terms of long term persistence, read-write capability, and cross-node access. The various `Volume` types will be explored by example throughout the book. This is another area where other than the few simplest `Volume` types (emptyDir, hostPath) you will likely need to find out from your cluster admin which `Volume` types are supported (`flocker`, `nfs`, `AzureDiskVolume`). On Google Cloud or Amazon Web Services for example, there are the `gcePersistentDisk` and `awsElasticBlockStore` types respectively.

We will attach various types of `Volumes` to our `Pods` as necessary in order to specify configuration, share files across `Pods`, or persist files across `Pod` restarts.

ConfigMaps & Secrets

Many times you will have a single `Pod`/container template and need to instantiate it with various underlying configurations, or you may just have some values that need to change over time but don't necessarily require rebuilding your containers. `ConfigMaps` let you store a set of key/value pairs and reference them across various resources. For example, when setting up feature flags[15], you could store a true/false value for each feature toggle, then automatically turn these `ConfigMap` values into environment variables to be used by your process upon startup. If you need to change the features for an environment, simply modify the `ConfigMap`, and Kubernetes takes care of the rest (provided you have it configured to do so!). Another common use would be specifying the locations of external resources, or specifying non-sensitive authentication information such as say a Google Analytics Site ID.

Sometimes you need to store and access more sensitive data, such as a username/password combination or a secret token. Se-

[15] Feature Flags are a method for deploying code changes. You wrap your changes in a conditional, which allows us to "flip" the feature on by changing configuration so our conditional evaluates to true, which causes our changed code to be executed

crets are essentially the same as `ConfigMaps` but with some extra precautions taken to prevent leakage. One of these precautions is to always mount `Secrets` on volatile storage such as RAM. By default, Kubernetes does not encrypt the secrets values for storage in etcd, so be aware that anyone with access to your etcd cluster also has access to your plaintext secrets. If you need a robust and secure solution for holding secrets values, it's probably time to consider setting up an external secrets storage service like Vault. We will cover configuring and using encrypted secrets storage in the Mini-Cookbook section.

Cluster Nodes

Kubernetes as a Service

For this course, we are going to demonstrate all examples on the google cloud environment. There are several commercial products or software packages that automate the process of setting up and maintaining the etcd and Kubernetes nodes, and my use of google cloud here shouldn't influence your decision about where and how to run Kubernetes.

Those looking for a local test environment should look at running minikube [16] or for a more permanent multi-node consider running kube-deploy.[17]

[16] `https://github.com/kubernetes/minikube`

[17] `https://github.com/kubernetes/kube-deploy`

Cluster Setup

Initializing a Kubernetes cluster from scratch involves many trade-off decisions that are best made with an understanding of Kubernetes and how it's typically used. Even if your goal is to run and maintain your own cluster, I highly recommend touring around a bit on GKE or a similar cloud managed environment to get a feel for what a well setup cluster should look and act like. With the experience and knowledge under your belt, going through a resource like Kelsey Hightower's Kubernetes the Hard Way [18] should be much more informative and interesting. This document describes how to set up a Kubernetes cluster and the underlying dependencies manually.

[18] `https://github.com/kelseyhightower/kubernetes-the-hard-way`

Automatic Setup

In production, doing anything manually is almost never a good idea. Automation of the same setup tasks (see kube-deploy or similar) gives repeatability, auditability, and a lower chance of operator error[19]. The importance of auditability can't be overstated here. For example, if your setup process suddenly stops working correctly, you can refer to the audit logs to find the time the latest successful node

[19] Technically, what we get with automation is **more leverage on operator input**, meaning that while your intentional changes give you additional power to accomplish tasks with greater ease, you also run the risk of accidentally causing much more damage with an error. Fortunately, Kubernetes provides other mitigations for this increased risk

came online, then view all changes to the setup configuration since that date. Additionally, if a problematic change is found, you now know who authored and/or approved that change. This lets you construct a feedback system to pass the error signal all the way back to the engineer who checked in the problem lines and caused the error.

The repeatability and unattended nature of automating the setup task also allows us to test and verify changes to this script, and perhaps prevent problematic changes from ever being merged into our master branch in the first place. Your existing continuous integration infrastructure can attempt to spin up and then destroy a Kubernetes node on each potential code change as an automated integration test. Failure to create and destroy this node could automatically surface feedback to the developer attempting to change the code.

Developer Sanity / Quality of Life

One thing I've found particularly useful is to create a small shell script to automatically set the gcloud auth user and project, then fetch the Kubernetes credentials for use. Similarly this script could talk to your internal LDAP instance, or some other private resource to fetch Kubernetes credentials. The important thing here is to have a simple interface, this will not only speed up and help organize your management and ops tasks, but will allow inexperienced developers to use deploy scripts and such without having to learn details about Kubernetes auth. Remember to make the "right" way the easiest way!

Kubernetes Example

Now, we will continue developing our simple application and deployment. This source code is available online at `https://withku.be/links/example-kube`

All examples are available in a single archive at `https://withku.be/links/examples`

We will begin by writing Kubernetes manifests to deploy the simple docker container we created in the Container chapter. In the next chapter, Helm, we will convert our bare YAML files into a Helm chart for easier deployment management and reuse. Then, in the Jenkins chapter we will set up a service and use it to perform automatic testing and deployment using configuration files from our repository. Finally, we'll set up some simple monitoring to measure and alert on our deployments.

Listing 11: Kubernetes example file listing

```
kube_example/
├── app-config.yaml
├── app-deployment.yaml
├── app-svc.yaml
└── make_secret.sh
```

Listing 12: app-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-app-configmap
data:
  GOOGLE_APPLICATION_CREDENTIALS: /keys/storage-reader-key.json
  GCLOUD_STORAGE_BUCKET_TARGET: my-gcloud-storage-bucket
```

First, we'll need a `ConfigMap` to hold our two env variables. This `ConfigMap` is very simple and only holds two keys. Note that the key names specified here don't necessarily need to be the final file or env name, but for simplicity here I'm just going to use the final names. We will be using `ConfigMaps` frequently throughout the examples in this book to store and aggregate configuration values and scripts.

Make sure you specify a `name` for the `ConfigMap` under the `metadata` section. The `metadata` section can also hold a list of key:value `labels`, or any other metadata you want to attach under `annotations`. Typically `labels` are used to select and find resources, while `annotations` are not intended to be used for resource selection, and have fewer restrictions.

Listing 13: app-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: demo-app
  labels:
    app: demo-app
    env: prod
spec:
  replicas: 2
  revisionHistoryLimit: 5
  template:
    metadata:
      labels:
        app: demo-app
        env: prod
    spec:
      containers:
      - name: app
        image: docker.io/limnick/gcloud_storage_list_example:latest
        imagePullPolicy: Always
        volumeMounts:
          - name: gcs-keys
            mountPath: /keys
            readOnly: true
        env:
        - name: GOOGLE_APPLICATION_CREDENTIALS
          valueFrom:
            configMapKeyRef:
              name: demo-app-configmap
              key: GOOGLE_APPLICATION_CREDENTIALS
        - name: GCLOUD_STORAGE_BUCKET_TARGET
          valueFrom:
            configMapKeyRef:
              name: demo-app-configmap
              key: GCLOUD_STORAGE_BUCKET_TARGET
        ports:
        - containerPort: 8888
        livenessProbe:
          httpGet:
            path: /
            port: 8888
          periodSeconds: 10
```

```
          timeoutSeconds: 3
        readinessProbe:
          httpGet:
            path: /
            port: 8888
          periodSeconds: 10
          timeoutSeconds: 3
        resources:
          requests:
            cpu: 10m
            memory: 128Mi
      volumes:
        - name: gcs-keys
          secret:
            secretName: "demo-app-gcs-keys"
            items:
              - key: "storage-reader-key.json"
                path: "storage-reader-key.json"
```

This deployment named `demo-app` specifies that we spin up 2 replicas of our example pod. A deployment consists of a `Pod` template and optional configuration for performing updates. In our example, we are leaving the update `strategy` as the default value of `RollingUpdate`, which will try to keep our deployment online as we perform changes.

Inside the `Pod` template, we specify the information for our server container and mount the gcloud storage key from our `Secret`. This `Secret` gets mounted as a volume, which means we can access the keys of the secret as files in the mounted directory. Additionally, we specify two required environment variables that will pull their values from the `ConfigMap` we defined above.

There are also liveness and readiness probes specified to perform a http request to the root path on port 8888. The readiness probe runs during the startup process until it returns successful, at which point the `Pod` is marked ready and traffic will be routed to it. After the `Pod` is running, we switch to the liveness probe and check to make sure the `Pod` is still alive and running. If at any point we detect the `Pod` is no longer responding to our http get request, Kubernetes will mark it unhealthy and begin the process of repairing it (typically by deleting and recreating the `Pod`).

Listing 14: `app-svc.yaml`

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: demo-app
5   labels:
6     app: demo-app
7     env: prod
8 spec:
9   type: LoadBalancer
10  ports:
11  - port: 80
12    targetPort: 8888
13    protocol: TCP
14    name: http
15  selector:
16    app: demo-app
17    env: prod
```

This `Service` sets up a public IP address and routes traffic to all `Pods` marked Ready with labels that match our `selector` (Lines 15-17). You'll see they match the labels from the `Pod` template in our `Deployment`. We accept traffic on port 80 externally, but we route it to port 8888 internally. You can route traffic at Layer 4 as we do here (protocol: TCP), or at Layer 7 (protocol: HTTP). Routing at Layer 7 allows us to route and match based on the HTTP headers in the request, while Routing at Layer 4 simply passes the traffic to any functioning endpoint.

You can check if a service has any place to route traffic to by running a `kubectl describe svc <service-name>` and checking the `Endpoints` list. If you don't see any entries, ensure you have running `Pods` and double check for typos in your label selector.

Listing 15: `make_secret.sh`

```
#!/bin/sh

kubectl create secret generic demo-app-gcs-keys \
    --from-file=$1
```

Finally, we'll need to create a secret to hold our gcloud storage key. Instead of simply running the command required, we're going to put this command into a simple script and check it into our repository. This lets us keep track of the process and potentially provide a consistent interface for our developers as we modify our deploy process. Later, we'll see how to use Helm to automatically fetch secret values and package them appropriately. In our case we just call `./make\_secret.sh path/to/gcloud.key`.

The command we run here is `kubectl create secret generic`. The `-from-file` argument grabs a file by reference, then creates a key with the filename and assigns a value with the contents of that file. You may pass `-from-file` multiple times to include multiple files.