**CI/CD Flow**

Prod-

    Requester (Developer/QA) raise CR  ---> Devops will validate change Docs(Security/IT head approval, QA, Test-cases, SOnar/Code-coverage report, release notes) --->CM Team will validate and approve

    Jenkins  -- shared pipeline/ earlier scripted/declarative mixed ---

    CI Part # Pull code from Bit-bucket(release branch) ---> Build(Maven(95-97% java)/npm) ---SonarQube(code coverage. For new micro-service <70%, older <25%) ---> SUCCESS--->mail shoot(Releated vertical) --- Nexus (Artifact Upload)

Build- mvn clean package  | war||jar

Nexus- mvn clean upload

    CD Part 1 # Code pull Nexus --->Ansible (deploy on 1 node) -->Monitring(ELK,Prometheus/Grafana) metrix check -- If satisfactory , then deployment will happen on all pending node 1 by 1 --->Dashboard create -- SUCCESS-- > Code replication on DR ---> PUll request(release-master merge code).

    CD Part 2 # Docker image---->Ansible deploy on Prod ---> K8s cluster  --->Monitoring

# Shared Library Staructure

    vars/

        This directory holds all the global shared library code that can be called from a pipeline. It has all the library files with a .groovy extension.

    src/

        Its a regular java source directory. Its added to class path during every script compilation. Here you can add custom groovy code to extend your shared library code.

    resources/

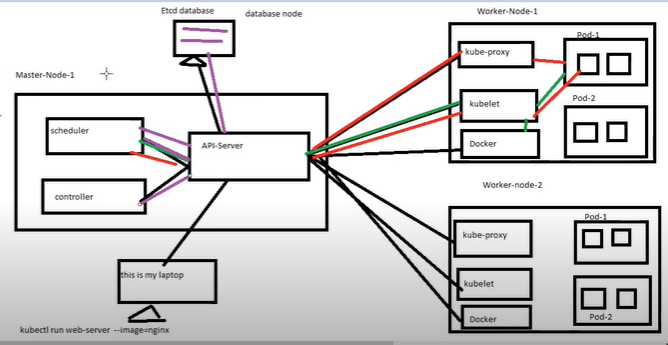
        All the non-groovy files required for your pipelines can be managed in this folder.

<https://confluence.airtel.com/display/AIR/APB-CICD-Flow>

<https://code.airtelworld.in:7990/bitbucket/projects/APB/repos/jenkins-shared-library/browse/vars/deployment.groovy>

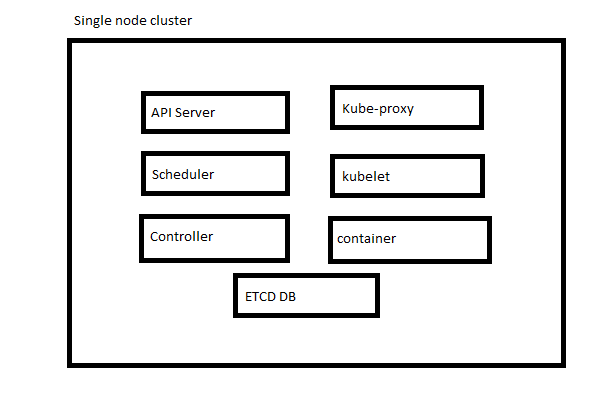
1. Kubernets Components-----

1. API Server
2. Scheduler
3. Kubelet
4. Kube-proxy
5. Container-Docker
6. Controller
7. ETCD Database
8. POD



2. What is single node cluster in kubernetes?

Ans--- In single node cluster, all component should be in one cluster like API Server, scheduler, kube-proxy, etcd etc. We are using this cluster in learning, testing environment.



3. **Prerequisites for multimode cluster?**

Ans:--- There are some Prerequisites for multimode cluster like,

1. Swap off
2. 2 GB of RAM (minimum)
3. 2 core CPU
4. Docker should be start or enable--- kubernetes use docker for creating container
5. Kubeadm----- Cluster will be setup by kubeadm
6. Kubelet start/Enable---- Kubelete will manage of all component of cluster
7. Kubectl---- operational work will be performed by kubectl
8. Run kubeadm init for creating cluster on master node.

**Control-plane node(s)**

| **Protocol** | **Direction** | **Port Range** | **Purpose** | **Used By** |
| --- | --- | --- | --- | --- |
| TCP | Inbound | 6443\* | Kubernetes API server | All |
| TCP | Inbound | 2379-2380 | etcd server client API | kube-apiserver, etcd |
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 10251 | kube-scheduler | Self |
| TCP | Inbound | 10252 | kube-controller-manager | Self |
|  |  |  |  |  |

**Worker node(s)**

| **Protocol** | **Direction** | **Port Range** | **Purpose** | **Used By** |
| --- | --- | --- | --- | --- |
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 30000-32767 | NodePort Services† | All |

**4. Access kubernetes cluster from our local machine?**

Ans:---- Install kubectl using below url,

https://storage.googleapis.com/kubernetes-release/release/v1.20.0/bin/windows/amd64/kubectl.exe

Then create a directory at C:\kubernetes and copy downloaded file in this directory then set environment variables.

Create C:\Users\devops\.kube directory and copy /etc/kubernetes/admin.conf file.

Then rename this file as a config then we can run kubectl command on cmd.

**For Linux :---**

Install kubectl on local linux machine

Download kubectl using below url,

https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

chmod +x ./kubectl

mv ./kubectl /usr/local/bin/kubectl

Now Test to ensure the version you installed is up-to-date:

kubectl version –client

Now create .kube directory and copy /etc/kubernetes/admin.conf file from master node to local machine in .kube directory then rename as config

**5. How to upgrade kubernetes cluster?**

Ans---- Before upgrade, we need to take backup of **/etc/kubernetes/manifests/** and /**var/lib/etcd/** in other drive or on network storage.

Kubernetes itself take backup for /etc/kubernetes/manifests and /var/lib/etcd in /etc/kubernetes/tmp directory on same machine (node) it means if that node will be crash then all backup will be lost.

When node will be upgrade then pods container will be restart.

When we are going to upgrade then we need to run that node in draining mode. All existing pods will be in running mode but we can’t create any new pods.

All containers are restarted after upgrade, because the container spec hash value is changed.

Commands:-----

1. Check which version is running,

yum list --showduplicates kubeadm --disableexcludes=kubernetes

# find the latest 1.20 version in the list

# it should look like 1.20.x-0, where x is the latest patch

The upgrade procedure on control plane nodes should be executed one node at a time.

1. Upgrade kubeadm---

replace x in 1.20.x-0 with the latest patch version

yum install -y kubeadm-1.20.x-0 --disableexcludes=kubernetes

1. Verify that the download works and has the expected version:

Kubeadm version

1. Verify the upgrade plan:

Kubeadm upgrade plan

This command checks that your cluster can be upgraded, and fetches the versions you can upgrade to. It also shows a table with the component config version states.

**Note:** kubeadm upgrade also automatically renews the certificates that it manages on this node. To opt-out of certificate renewal the flag --certificate-renewal=false can be used.

**Note:** If kubeadm upgrade plan shows any component config that require manual upgrade, users must provide a config file with replacement configs to kubeadm upgrade apply via the --config command line flag. Failing to do so will cause kubeadm upgrade apply to exit with an error and not perform an upgrade.

1. Choose a version to upgrade to, and run the appropriate command

# replace x with the patch version you picked for this upgrade

sudo kubeadm upgrade apply v1.20.x

**Once the command finishes you should see:**

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.20.x". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading your kubelets if you haven't already done so.

* Manually upgrade your CNI provider plugin.

Your Container Network Interface (CNI) provider may have its own upgrade instructions to follow. Check the [addons](https://kubernetes.io/docs/concepts/cluster-administration/addons/) page to find your CNI provider and see whether additional upgrade steps are required.

This step is not required on additional control plane nodes if the CNI provider runs as a DaemonSet.

**For the other control plane nodes**

Same as the first control plane node but use:

sudo kubeadm upgrade node

instead of:

sudo kubeadm upgrade apply

Also calling kubeadm upgrade plan and upgrading the CNI provider plugin is no longer needed.

Drain the node

* Prepare the node for maintenance by marking it unschedulable and evicting the workloads:
* # replace <node-to-drain> with the name of your node you are draining

kubectl drain <node-to-drain> --ignore-daemonsets

**Upgrade kubelet and kubectl**

# replace x in 1.20.x-0 with the latest patch version

yum install -y kubelet-1.20.x-0 kubectl-1.20.x-0 --disableexcludes=kubernetes

Restart the kubelet:

sudo systemctl daemon-reload

sudo systemctl restart kubelet

### Uncordon (remove draining mode) the node:

# replace <node-to-drain> with the name of your node

kubectl uncordon <node-to-drain>

## **Upgrade worker nodes**

The upgrade procedure on worker nodes should be executed one node at a time or few nodes at a time, without compromising the minimum required capacity for running your workloads.

1. Upgrade kubeadm:

# replace x in 1.20.x-0 with the latest patch version

yum install -y kubeadm-1.20.x-0 --disableexcludes=kubernetes

1. Drain the node

Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

# replace <node-to-drain> with the name of your node you are draining

kubectl drain <node-to-drain> --ignore-daemonsets

1. Call "kubeadm upgrade"

For worker nodes this upgrades the local kubelet configuration:

sudo kubeadm upgrade node

1. Upgrade the kubelet and kubectl:

# replace x in 1.20.x-0 with the latest patch version

yum install -y kubelet-1.20.x-0 kubectl-1.20.x-0 --disableexcludes=kubernetes

* Restart the kubelet:
* sudo systemctl daemon-reload

sudo systemctl restart kubelet

### Uncordon (remove draining mode) the node:

# replace <node-to-drain> with the name of your node

kubectl uncordon <node-to-drain>

## **Verify the status of the cluster**

After the kubelet is upgraded on all nodes verify that all nodes are available again by running the following command from anywhere kubectl can access the cluster:

kubectl get nodes

The STATUS column should show ready for all your nodes, and the version number should be updated.

## **Recovering from a failure state**

If kubeadm upgrade fails and does not roll back, for example because of an unexpected shutdown during execution, you can run kubeadm upgrade again. This command is idempotent and eventually makes sure that the actual state is the desired state you declare.

To recover from a bad state, you can also run kubeadm upgrade apply --force without changing the version that your cluster is running.

During upgrade kubeadm writes the following backup folders under /etc/kubernetes/tmp:

* kubeadm-backup-etcd-<date>-<time>
* kubeadm-backup-manifests-<date>-<time>

kubeadm-backup-etcd contains a backup of the local etcd member data for this control plane Node. In case of an etcd upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in /var/lib/etcd. In case external etcd is used this backup folder will be empty.

kubeadm-backup-manifests contains a backup of the static Pod manifest files for this control plane Node. In case of a upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in /etc/kubernetes/manifests. If for some reason there is no difference between a pre-upgrade and post-upgrade manifest file for a certain component, a backup file for it will not be written.

## **How it works**

kubeadm upgrade apply does the following:

* Checks that your cluster is in an upgradeable state:
  + The API server is reachable
  + All nodes are in the Ready state
  + The control plane is healthy
* Enforces the version skew policies.
* Makes sure the control plane images are available or available to pull to the machine.
* Generates replacements and/or uses user supplied overwrites if component configs require version upgrades.
* Upgrades the control plane components or rollbacks if any of them fails to come up.
* Applies the new kube-dns and kube-proxy manifests and makes sure that all necessary RBAC rules are created.
* Creates new certificate and key files of the API server and backs up old files if they're about to expire in 180 days.

kubeadm upgrade node does the following on additional control plane nodes:

* Fetches the kubeadm ClusterConfiguration from the cluster.
* Optionally backups the kube-apiserver certificate.
* Upgrades the static Pod manifests for the control plane components.
* Upgrades the kubelet configuration for this node.

kubeadm upgrade node does the following on worker nodes:

* Fetches the kubeadm ClusterConfiguration from the cluster.
* Upgrades the kubelet configuration for this node.

**Kubernetes Cluster Reset-------------**

**Command--- kubeadm reset**

**After reset join master node command----- kubeadm init**

**After reset join worker node command----- kubeadm join on worker node**

There are four phases of reset flow-------

1. Preflight
2. Update-cluster-status
3. Remove-etcd-member
4. Cleanup-node

Reset command----

Kubeadm --skip remove-etcd-member

It will remove everything except ETCD database.

kubeadm reset is responsible for cleaning up a node local file system from files that were created using the kubeadm init or kubeadm join commands. For control-plane nodes reset also removes the local stacked etcd member of this node from the etcd cluster and also removes this node's information from the kubeadm ClusterStatus object. ClusterStatus is a kubeadm managed Kubernetes API object that holds a list of kube-apiserver endpoints.

kubeadm reset phase can be used to execute the separate phases of the above workflow. To skip a list of phases you can use the --skip-phases flag, which works in a similar way to the kubeadm join and kubeadm init phase runners.

### External etcd clean up

kubeadm reset will not delete any etcd data if external etcd is used. This means that if you run kubeadm init again using the same etcd endpoints, you will see state from previous clusters.

To wipe etcd data it is recommended you use a client like etcdctl, such as:

etcdctl del "" --prefix

**Kubernetes Token and CA public key-----------------**

kubeadm join 192.168.1.130:6443 --token qt57zu.wuvqh64un13trr7x --discovery-token-ca-cert-hash sha256:5ad014cad868fdfe9388d5b33796cf40fc1e8c2b3dccaebff0b066a0532e8723

**a. kubeadm join -----command**

**b. 192.168.1.130:6443---- API server IP and port**

**c. --token qt57zu.wuvqh64un13trr7x----- Token**

**d. --discovery-token-ca-cert-hash sha256:5ad014cad868fdfe9388d5b33796cf40fc1e8c2b3dccaebff0b066a0532e8723------**

**CA Public key**

**Check master node tocken---**

Kubeadm token list

**Check CA public key-----**

openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.\* //'

kubeadm token crete ------------ This command will generate token

kubeadm token list

kubeadm token create –ttl 10m ---- token will be valid for next 10 minutes

kubeadm token create –ttl 0(zero) ---- token will never expired

**Skip CA public key------**

kubeadm join 192.168.1.130:6443 --token qt57zu.wuvqh64un13trr7x –discovery-token-unsafe-skip-ca-verification

**Checking** auto approval

kubectl get csr

**Disable auto approval for join worker node----**

kubectl delete clusterrolebinding kubeadm:node-autoapprove-bootstrap

**Check which worker node wants to join---**

Kubectl describe csr <pending worker name>

**Approve manually worker node request for join---**

Kubectl certificate approve <pending worker name>

**Find token and CA certificate at a time**

kubeadm token create --print-join-command

#### **File or HTTPS-based discovery**

This provides an out-of-band way to establish a root of trust between the control-plane node and bootstrapping nodes. Consider using this mode if you are building automated provisioning using kubeadm. The format of the discovery file is a regular Kubernetes [kubeconfig](https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/) file.

In case the discovery file does not contain credentials, the TLS discovery token will be used.

**Example kubeadm join commands:**

* kubeadm join --discovery-file path/to/file.conf (local file)
* kubeadm join --discovery-file https://url/file.conf (remote HTTPS URL)

**Kubernetes Replication Controller, Replica Set and Deployments**

[**https://www.mirantis.com/blog/kubernetes-replication-controller-replica-set-and-deployments-understanding-replication-options/**](https://www.mirantis.com/blog/kubernetes-replication-controller-replica-set-and-deployments-understanding-replication-options/)

## **What is Kubernetes replication for?**

Before we go into the details on how you would do replication, let’s talk about why.  Typically you would want to replicate your containers (and thereby your applications) for several reasons, including:

* **Reliability**: By having multiple versions of an application, you prevent problems if one or more fails.  This is particularly true if the system replaces any containers that fail.
* **Load balancing**: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node. This is something that Kubernetes does out of the box, making it extremely convenient.
* **Scaling**: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed.

**Replication is appropriate for numerous use cases, including**:

* **Microservices-based applications**: In these cases, multiple small applications provide very specific functionality.
* **Cloud native applications**: Because cloud-native applications are based on the theory that any component can fail at any time, replication is a perfect environment for implementing them, as multiple instances are baked into the architecture.
* **Mobile applications**: Mobile applications can often be architected so that the mobile client interacts with an isolated version of the server application.

Kubernetes has multiple ways in which you can implement replication.

## Types of Kubernetes replication

In this article, we’ll discuss three different forms of replication: the Kubernetes Replication Controller (Replication Controller), Replica Sets, and Deployments.

### Kubernetes Replication Controller

The Replication Controller is the original form of replication in Kubernetes.  It’s being replaced by Replica Sets, but it’s still in wide use, so it’s worth understanding what it is and how it works. **A Replication Controller is a structure that enables you to easily create multiple pods**, then make sure that that number of pods always exists. **If a pod does crash, the Replication Controller replaces it**.  A Kubernetes controller such as the Replication Controller also provide other benefits, such as the ability to scale the number of pods, and to update or delete multiple pods with a single command. You can create a Replication Controller with an imperative command, or declaratively, from a file.  For example, create a new file called rc.yaml and add the following text:

apiVersion: v1

kind: ReplicationController

metadata:

 name: soaktestrc

spec:

 replicas: 3

 selector:

   app: soaktestrc

 template:

   metadata:

     name: soaktestrc

     labels:

       app: soaktestrc

   spec:

     containers:

     - name: soaktestrc

       image: nickchase/soaktest

       ports:

       - containerPort: 80

Most of this structure should look familiar from our discussion of how to create a [Kubernetes Deployment](https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/); we’ve got the name of the actual Kubernetes Replication Controller (soaktestrc) and we’re designating that we should have 3 replicas, each of which are defined by the template.  The selector defines how we know which pods belong to this Replication Controller. Now tell Kubernetes to create the Replication Controller based on that YAML file:

# **kubectl create -f rc.yaml**

replicationcontroller "soaktestrc" created

Let’s take a look at what we have using the describe command:

# **kubectl describe rc soaktestrc**

Name:           soaktestrc

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app=soaktestrc

Labels:         app=soaktestrc

Replicas:       3 current / 3 desired

Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type   Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                  -------

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-g5snq

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-cws05

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-ro2bl

As you can see, we’ve got the Replication Controller, and there are 3 replicas, of the 3 that we wanted.  All 3 of them are currently running.  You can also see the individual pods listed underneath, along with their names and other relevant fields.  If you ask Kubernetes to show you the pods, you can see those same names show up:

# **kubectl get pods**

NAME               READY     STATUS    RESTARTS   AGE

soaktestrc-cws05   1/1       Running   0          3m

soaktestrc-g5snq   1/1       Running   0          3m

soaktestrc-ro2bl   1/1       Running   0          3m

Next we’ll look at Replica Sets, but first let’s clean up:

# **kubectl delete rc soaktestrc**

replicationcontroller "soaktestrc" deleted

# **kubectl get pods**

As you can see, when you delete the Replication Controller, you also delete all of the pods that it created.

### Kubernetes Replica Sets

It can be tricky to compare a replica controller vs replica set (ReplicalSet), because the latter is a sort of a hybrid. They are in some ways more powerful than ReplicationControllers, and in others they are less powerful. ReplicaSets are declared in essentially the same way as ReplicationControllers, except that they have more options for the selector.  For example, we could create a ReplicaSet like this:

apiVersion: extensions/v1beta1

kind: ReplicaSet

metadata:

 name: soaktestrs

spec:

 replicas: 3

 selector:

   matchLabels:

     app: soaktestrs

 template:

   metadata:

     labels:

       app: soaktestrs

  environment: dev

   spec:

     containers:

     - name: soaktestrs

       image: nickchase/soaktest

       ports:

       - containerPort: 80

In this case, it’s more or less the same as when we were creating the Replication Controller, except we’re using matchLabels instead of label.  But we could just as easily have said:

...

spec:

 replicas: 3

 selector:

**matchExpressions:**

**- {key: app, operator: In, values: [soaktestrs, soaktestrs, soaktest]}**

**- {key: teir, operator: NotIn, values: [production]}**

 template:

   metadata:

...

In this case, we’re looking at two different conditions:

1. The app label must be soaktestrc, soaktestrs, or soaktest
2. The tier label (if it exists) must not be production

Let’s go ahead and create the Replica Set and get a look at it:

# **kubectl create -f replicaset.yaml**

replicaset "soaktestrs" created

# **kubectl describe rs soaktestrs**

Name:           soaktestrs

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app in (soaktest,soaktestrs),teir notin (production)

Labels:         app=soaktestrs

Replicas:       3 current / 3 desired

Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type    Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                   -------

 1m            1m              1       {replicaset-controller }                        Normal  SuccessfulCreate Created pod: soaktestrs-it2hf

 1m            1m              1       {replicaset-controller }                       Normal  SuccessfulCreate Created pod: soaktestrs-kimmm

 1m            1m              1       {replicaset-controller }                        Normal  SuccessfulCreate Created pod: soaktestrs-8i4ra

# **kubectl get pods**

NAME               READY     STATUS    RESTARTS   AGE

soaktestrs-8i4ra   1/1       Running   0          1m

soaktestrs-it2hf   1/1       Running   0          1m

soaktestrs-kimmm   1/1       Running   0          1m

As you can see, the output is pretty much the same as for a Replication Controller (except for the selector), and for most intents and purposes, they are similar.  **The major difference difference between a replication controller and replica set is that the rolling-update command works with Replication Controllers, but won’t work with a Replica Set**.  This is because Replica Sets are meant to be used as the backend for Deployments. Let’s clean up before we move on.

# **kubectl delete rs soaktestrs**

replicaset "soaktestrs" deleted

# **kubectl get pods**

Again, the pods that were created are deleted when we delete the Replica Set.

### Deployments

Deployments are intended to replace Replication Controllers. When comparing a Deployment vs Replica Set, the former provides  the same replication functions (through Replica Sets) and also the ability to rollout changes and roll them back if necessary. Let’s create a simple Deployment using the same image we’ve been using.  First create a new configuration file, deployment.yaml, and add the following:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

 name: soaktest

spec:

 replicas: 5

 template:

   metadata:

     labels:

       app: soaktest

   spec:

     containers:

     - name: soaktest

       image: nickchase/soaktest

       ports:

       - containerPort: 80

Now go ahead and create the Deployment:

# **kubectl create -f deployment.yaml**

deployment "soaktest" created

Now let’s go ahead and describe the Deployment:

# **kubectl describe deployment soaktest**

Name:                   soaktest

Namespace:              default

CreationTimestamp:      Sun, 05 Mar 2017 16:21:19 +0000

Labels:                 app=soaktest

Selector:               app=soaktest

Replicas:               5 updated | 5 total | 5 available | 0 unavailable

StrategyType:           RollingUpdate

MinReadySeconds:        0

RollingUpdateStrategy:  1 max unavailable, 1 max surge

OldReplicaSets:         <none>

NewReplicaSet:          soaktest-3914185155 (5/5 replicas created)

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type    Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                   -------

 38s           38s             1       {deployment-controller }                        Normal  ScalingReplicaSet        Scaled up replica set soaktest-3914185155 to 3

 36s           36s             1       {deployment-controller }                        Normal  ScalingReplicaSet        Scaled up replica set soaktest-3914185155 to 5

As you can see, rather than listing the individual pods, Kubernetes shows us the Replica Set.  Notice that the name of the Replica Set is the Deployment name and a hash value. A complete discussion of updates is out of scope for this article — we’ll cover it in the future — but couple of interesting things here:

* The StrategyType is RollingUpdate. This value can also be set to Recreate.
* By default we have a minReadySeconds value of 0; we can change that value if we want pods to be up and running for a certain amount of time — say, to load resources — before they’re truly considered “ready”.
* The RollingUpdateStrategy shows that we have a limit of 1 maxUnavailable — meaning that when we’re updating the Deployment, we can have up to 1 missing pod before it’s replaced, and 1 maxSurge, meaning we can have one extra pod as we scale the new pods back up.

As you can see, the Deployment is backed, in this case, by Replica Set soaktest-3914185155. If we go ahead and look at the list of actual pods…

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3914185155-7gyja   1/1       Running   0          2m

soaktest-3914185155-lrm20   1/1       Running   0          2m

soaktest-3914185155-o28px   1/1       Running   0          2m

soaktest-3914185155-ojzn8   1/1       Running   0          2m

soaktest-3914185155-r2pt7   1/1       Running   0          2m

… you can see that their names consist of the Replica Set name and an additional identifier.

### Passing environment information: identifying a specific pod

Before we look at the different ways that we can affect replicas, let’s set up our deployment so that we can see what pod we’re actually hitting with a particular request.  To do that, the image we’ve been using displays the pod name when it outputs:

<?php

$limit = $\_GET['limit'];

if (!isset($limit)) $limit = 250;

for ($i; $i < $limit; $i++){

    $d = tan(atan(tan(atan(tan(atan(tan(atan(tan(atan(123456789.123456789))))))))));

}

**echo "Pod ".$\_SERVER['POD\_NAME']." has finished!\n";**

?>

As you can see, we’re displaying an environment variable, POD\_NAME.  Since each container is essentially its own server, this will display the name of the pod when we execute the PHP. Now we just have to pass that information to the pod. We do that through the use of the Kubernetes Downward API, which lets us pass environment variables into the containers:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

 name: soaktest

spec:

 replicas: 3

 template:

   metadata:

     labels:

       app: soaktest

   spec:

     containers:

     - name: soaktest

       image: nickchase/soaktest

       ports:

       - containerPort: 80

**env:**

**- name: POD\_NAME**

**valueFrom:**

**fieldRef:**

**fieldPath: metadata.name**

As you can see, we’re passing an environment variable and assigning it a value from the Deployment’s metadata.  (You can find more information on metadata [here](https://kubernetes.io/docs/tasks/configure-pod-container/downward-api-volume-expose-pod-information/).) So let’s go ahead and clean up the Deployment we created earlier…

# **kubectl delete deployment soaktest**

deployment "soaktest" deleted

# **kubectl get pods**

… and recreate it with the new definition:

# **kubectl create -f deployment.yaml**

deployment "soaktest" created

Next let’s go ahead and [expose the pods to outside network requests](https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/) so we can call the nginx server that is inside the containers:

# **kubectl expose deployment soaktest --port=80 --target-port=80 --type=NodePort**

service "soaktest" exposed

Now let’s describe the services we just created so we can find out what port the Deployment is listening on:

# **kubectl describe services soaktest**

Name:                   soaktest

Namespace:              default

Labels:                 app=soaktest

Selector:               app=soaktest

Type:                   NodePort

IP:                     11.1.32.105

Port:                   <unset> 80/TCP

NodePort:               <unset> **30800**/TCP

Endpoints:              10.200.18.2:80,10.200.18.3:80,10.200.18.4:80 + 2 more...

Session Affinity:       None

No events.

As you can see, the NodePort is 30800 in this case; in your case it will be different, so make sure to check.  That means that each of the servers involved is listening on port 30800, and requests are being forwarded to port 80 of the containers.  That means we can call the PHP script with:

http://[HOST\_NAME OR HOST\_IP]:[PROVIDED PORT]

In my case, I’ve set the IP for my Kubernetes hosts to hostnames to make my life easier, and the PHP file is the default for nginx, so I can simply call:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-xnfme has finished!

So as you can see, this time the request was served by pod soaktest-3869910569-xnfme.

## Recovering from crashes: Creating a specified number of replicas

Now that we know everything is running, let’s take a look at some replication use cases. The first thing we think of when it comes to replication is recovering from crashes. If there are 5 (or 50, or 500) copies of an application running, and one or more crashes, it’s not a catastrophe.  Kubernetes improves the situation further by ensuring that if a pod goes down, it’s replaced. Let’s see this in action.  Start by refreshing our memory about the pods we’ve got running:

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-qqwqc   1/1       Running   0          11m

soaktest-3869910569-qu8k7   1/1       Running   0          11m

soaktest-3869910569-uzjxu   1/1       Running   0          11m

soaktest-3869910569-x6vmp   1/1       Running   0          11m

soaktest-3869910569-xnfme   1/1       Running   0          11m

If we repeatedly call the Deployment, we can see that we get different pods on a random basis:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-xnfme has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-x6vmp has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-uzjxu has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-x6vmp has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-uzjxu has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-qu8k7 has finished!

To simulate a pod crashing, let’s go ahead and delete one:

# **kubectl delete pod soaktest-3869910569-x6vmp**

pod "soaktest-3869910569-x6vmp" deleted

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-516kx   1/1       Running   0          18s

soaktest-3869910569-qqwqc   1/1       Running   0          27m

soaktest-3869910569-qu8k7   1/1       Running   0          27m

soaktest-3869910569-uzjxu   1/1       Running   0          27m

soaktest-3869910569-xnfme   1/1       Running   0          27m

As you can see, pod \*x6vmp is gone, and it’s been replaced by \*516kx.  (You can easily find the new pod by looking at the AGE column.) If we once again call the Deployment, we can (eventually) see the new pod:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-516kx has finished!

Now let’s look at changing the number of pods.

## Scaling up or down: Manually changing the number of replicas

One common task is to scale up a Deployment in response to additional load. Kubernetes has autoscaling, but we’ll talk about that in another article.  For now, let’s look at how to do this task manually. The most straightforward way is to simply use the scale command:

# **kubectl scale --replicas=7 deployment/soaktest**

deployment "soaktest" scaled

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-2w8i6   1/1       Running   0          6s

soaktest-3869910569-516kx   1/1       Running   0          11m

soaktest-3869910569-qqwqc   1/1       Running   0          39m

soaktest-3869910569-qu8k7   1/1       Running   0          39m

soaktest-3869910569-uzjxu   1/1       Running   0          39m

soaktest-3869910569-xnfme   1/1       Running   0          39m

soaktest-3869910569-z4rx9   1/1       Running   0          6s

In this case, we specify a new number of replicas, and Kubernetes adds enough to bring it to the desired level, as you can see. One thing to keep in mind is that Kubernetes isn’t going to scale the Deployment down to be below the level at which you first started it up.  For example, if we try to scale back down to 4…

# **kubectl scale --replicas=4 -f deployment.yaml**

deployment "soaktest" scaled

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-l5wx8   1/1       Running   0          11s

soaktest-3869910569-qqwqc   1/1       Running   0          40m

soaktest-3869910569-qu8k7   1/1       Running   0          40m

soaktest-3869910569-uzjxu   1/1       Running   0          40m

soaktest-3869910569-xnfme   1/1       Running   0          40m

… Kubernetes only brings us back down to 5, because that’s what was specified by the original deployment.

## Deploying a new version: Replacing replicas by changing their label

Another way you can use deployments is to make use of the selector.  In other words, if a Deployment controls all the pods with a tier value of dev, changing a pod’s teir label to prod will remove it from the Deployment’s sphere of influence. This mechanism enables you to selectively replace individual pods. For example, you might move pods from a dev environment to a production environment, perform debugging, documentation, and data recovery operations, or you might do a manual rolling update, updating the image, then removing some fraction of pods from the Deployment; when they’re replaced, it will be with the new image. If you’re happy with the changes, you can then replace the rest of the pods. Let’s see this in action.  As you recall, this is our Deployment:

# **kubectl describe deployment soaktest**

Name:                   soaktest

Namespace:              default

CreationTimestamp:      Sun, 05 Mar 2017 19:31:04 +0000

Labels:                 app=soaktest

Selector:               app=soaktest

Replicas:               3 updated | 3 total | 3 available | 0 unavailable

StrategyType:           RollingUpdate

MinReadySeconds:        0

RollingUpdateStrategy:  1 max unavailable, 1 max surge

OldReplicaSets:         <none>

NewReplicaSet:         **soaktest-3869910569** (3/3 replicas created)

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type              Reason                  Message

 ---------     --------        -----   ----                            -------------   --------  ------                  -------

 50s           50s             1       {deployment-controller }                        Normal            ScalingReplicaSet       Scaled up replica set soaktest-3869910569 to 3

And these are our pods:

# **kubectl describe replicaset soaktest-3869910569**

Name:           soaktest-3869910569

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app=soaktest,pod-template-hash=3869910569

Labels:         app=soaktest

               pod-template-hash=3869910569

Replicas:       5 current / 5 desired

Pods Status:    5 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type              Reason                  Message

 ---------     --------        -----   ----                            -------------   --------  ------                  -------

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-0577c

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-wje85

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-xuhwl

 1m            1m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-8cbo2

 1m            1m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-pwlm4

We can also get a list of pods by label:

# **kubectl get pods -l app=soaktest**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          7m

soaktest-3869910569-8cbo2   1/1       Running   0          6m

soaktest-3869910569-pwlm4   1/1       Running   0          6m

soaktest-3869910569-wje85   1/1       Running   0          7m

soaktest-3869910569-xuhwl   1/1       Running   0          7m

So those are our original soaktest pods; what if we wanted to add a new label?  We can do that on the command line:

# **kubectl label pods soaktest-3869910569-xuhwl experimental=true**

pod "soaktest-3869910569-xuhwl" labeled

# **kubectl get pods -l experimental=true**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-xuhwl   1/1       Running   0          14m

So now we have one experimental pod.  But since the experimental label has nothing to do with the selector for the Deployment, it doesn’t affect anything. So what if we change the value of the app label, which the Deployment **is** looking at?

# **kubectl label pods soaktest-3869910569-wje85 app=notsoaktest --overwrite**

pod "soaktest-3869910569-wje85" labeled

In this case, we need to use the overwrite flag because the app label already exists. Now let’s look at the existing pods.

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          17m

soaktest-3869910569-4cedq   1/1       Running   0          4s

soaktest-3869910569-8cbo2   1/1       Running   0          16m

soaktest-3869910569-pwlm4   1/1       Running   0          16m

soaktest-3869910569-wje85   1/1       Running   0          17m

soaktest-3869910569-xuhwl   1/1       Running   0          17m

As you can see, we now have six pods instead of five, with a new pod having been created to replace \*wje85, which was removed from the deployment. We can see the changes by requesting pods by label:

# **kubectl get pods -l app=soaktest**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          17m

soaktest-3869910569-4cedq   1/1       Running   0          20s

soaktest-3869910569-8cbo2   1/1       Running   0          16m

soaktest-3869910569-pwlm4   1/1       Running   0          16m

soaktest-3869910569-xuhwl   1/1       Running   0          17m

Now, there is one wrinkle that you have to take into account; because we’ve removed this pod from the Deployment, the Deployment no longer manages it.  So if we were to delete the Deployment…

# **kubectl delete deployment soaktest**

deployment "soaktest" deleted

The pod remains:

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-wje85   1/1       Running   0          19m

You can also easily replace all of the pods in a Deployment using the –all flag, as in:

# **kubectl label pods --all app=notsoaktesteither --overwrite**

But remember that you’ll have to delete them all manually!

## Conclusion

Replication is a large part of Kubernetes’ purpose in life, so it’s no surprise that we’ve just scratched the surface of what it can do, and how to use it. It is one of the most useful features for reliability purposes, for scalability, and even as a basis for your architecture. What do you anticipate using replication for, and what would you like to know more about? Let us know in the comments.

**LAB--- Create pod through command or yml file.**

* 1. kubectl get nodes -----List nodes
  2. kubectl describe controlplane --- Describe master node (controleplane)
  3. kubectl describe nodes node01 --- describe any particular node
  4. kubectl run web-server --image=nginx:1.7.1 – create pod
  5. kubectl get pod --- list pods
  6. kubectl get pod -o wide – check pod IP and node details
  7. watch kubectl get pod -o wide--- check pod in watch command
  8. kubectl describe pod web-server--- Describe pods
  9. kubectl run webserver --image=wordpress --- create pod
  10. kubectl get pod – describe pod
  11. kubectl taint --all node-role.kubernetes.io/master-node/master untainted ---- Removed taint for creating pods on master node.
  12. kubectl delete pod webserver-5db7895b57-c6dmd --- delete pod
  13. kubectl apply -f pod.yml---- create pod using yml file
  14. Syntax for making yml,

apiVersion: v1 kind: Pod

metadata:

name: web-server-1

spec:

containers:

- name: web-server-1

image: nginx

* 1. kubectl delete -f pod.yml---- delete pod using yml file

**LAB---- Create replication controller**

rc.yml

**apiVersion: v1**

**kind: ReplicationController**

**metadata:**

**name: nginx**

**spec:**

**replicas: 3**

**selector:**

**team: dev**

**template:**

**metadata:**

**name: nginx**

**labels:**

**team: dev**

**spec:**

**containers:**

**- name: nginx**

**image: nginx**

**ports:**

**- containerPort: 80**

* 1. kubectl apply –f rc.yml---create replication controller
  2. kubectl describe pod pod\_name
  3. kubectl lable pod pod\_name team=dev --- add label in pods
  4. kubectl lable pod pod\_name team- --- remove label from pods
  5. kubectl delete rc --cascade=false nginx --- remove replication controller from pods
  6. kubectl describe rc replication\_controller\_name--- describe rc
  7. kubectl delete –f rc.yml --- delete rc
  8. kubectl get all

**LAB---- Create replica Set**

1. Kubectl explain rs
2. kubectl label pod web-server-1 team=prod
3. kubectl label pod web-server-1 team=prod
4. kubectl label pod web-server-1 team-
5. kubectl get pods --show-labels
6. kubectl describe rs
7. kubectl describe pod

**replicasets.yml**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: cloudknowledge

spec:

replicas: 3

selector:

matchExpressions:

-

key: team

operator: In

values:

- dev

- prod

- test

-

key: team

operator: In

values:

- test

template:

metadata:

labels:

team: dev

name: dev

spec:

containers:

-

image: "nginx:1.7.1"

name: cloudknowledge

ports:

-

containerPort: 80

**Other rs.yml**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: cloudknowledge

spec:

replicas: 3

selector:

matchExpressions:

- {key: team, operator: In, values: [ dev, prod,test ]}

- {key: team, operator: NotIn, values: [ test ]}

template:

metadata:

labels:

team: dev

name: dev

spec:

containers:

-

image: "nginx:1.7.1"

name: cloudknowledge

ports:

-

containerPort: 80

**LAB---- Create Deployment:-**

apiVersion: apps/v1

kind: Deployment

metadata:

name: deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- image: nginx:1.7.1

name: cloudknowlwdge

ports:

- containerPort: 80

watch kubectl get pod,rs,deployment --show-labels

Kubecet get deployment

kubectl rollout history deployment

kubectl apply -f deployment.yml --record=true

kubectl describe deployment deployment (deployment\_name)

kubectl rollout history deployment deployment(deployment\_name)

kubectl set image deployments.app/deployment nginx=nginx:1.16.1 --record=true ------ **Rollout from command line.**

kubectl edit deployment----- **Edit deployment from command line**

kubectl rollout history deployment --revision=1----- See --revision=1 history

kubectl rollout undo deployment --to-revision=2---- **If you will enter without --to-revision=2 then it will be undo for last changes**

kubectl rollout pause deployment\_name **------ If you want that no one can rollout then oause your deployment**

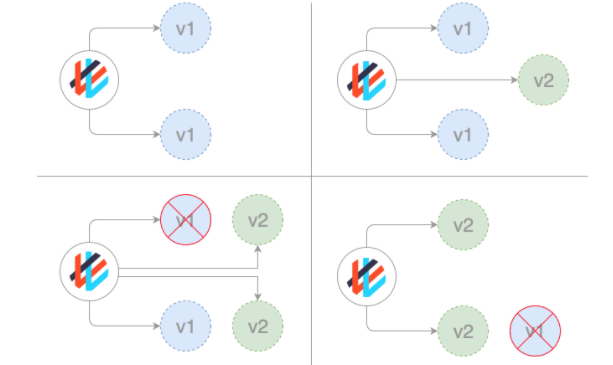
kubectl rollout resume deployment\_name ----- **remove pause from deployment.**

## **Deployment Strategies**

There are several different types of deployment strategies you can take advantage of depending on your goal. For example, you may need to roll out changes to a specific environment for more testing, or a subset of users/customers or you may want to do some user testing before making a feature 'Generally Available'.

### Rolling Deployment

The rolling deployment is the standard default deployment to Kubernetes. It works by slowly, one by one, replacing pods of the previous version of your application with pods of the new version without any cluster downtime.



A rolling update waits for new pods to become ready via your [readiness probe](https://www.weave.works/blog/resilient-apps-with-liveness-and-readiness-probes-in-kubernetes) before it starts scaling down the old ones. If there is a problem, the rolling update or deployment can be aborted without ringing the whole cluster down. In the YAML definition file for this type of deployment, a new image replaces the old image.

### apiVersion: apps/v1

### kind: Deployment

### metadata:

### name: deployment

### labels:

### app: nginx

### spec:

### replicas: 4

### strategy:

### type: RollingUpdate

### rollingUpdate:

### maxSurge: 4

### maxUnavailable: 0

### selector:

### matchLabels:

### app: nginx

### template:

### metadata:

### labels:

### app: nginx

### spec:

### containers:

### - image: nginx:1.11.1

### name: cloudknowlwdge

### ports:

### - containerPort: 80

Rolling updates can be further refined by adjusting the parameters in the manifest file:

### replicas: 4

### strategy:

### type: RollingUpdate

### rollingUpdate:

### maxSurge: 25%

### maxUnavailable: 10%

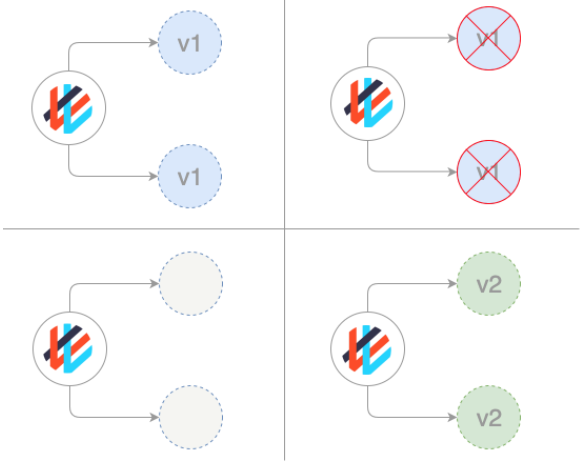
### selector:

* **maxSurge: the maximum number of pods that can be created over the desired number of pods. Again this can be an absolute number or a percentage of the replicas count; the default is 25%.**
* **maxUnavailable: the maximum number of pods that can be unavailable during the update process. This can be an absolute number or percentage of the replicas count; the default is 25%.**

## **Recreate**

**kubectl explain deployment --recursive | less ---explain deployment syntax**

In this type of very simple deployment, all of the old pods are killed all at once and get replaced all at once with the new ones.



**This manifest looks something like this:**

### apiVersion: apps/v1

### kind: Deployment

### metadata:

### name: deployment

### labels:

### app: nginx

### spec:

### replicas: 4

### strategy:

### type: Recreate

### selector:

### matchLabels:

### app: nginx

### template:

### metadata:

### labels:

### app: nginx

### spec:

### containers:

### - image: nginx:1.18.1

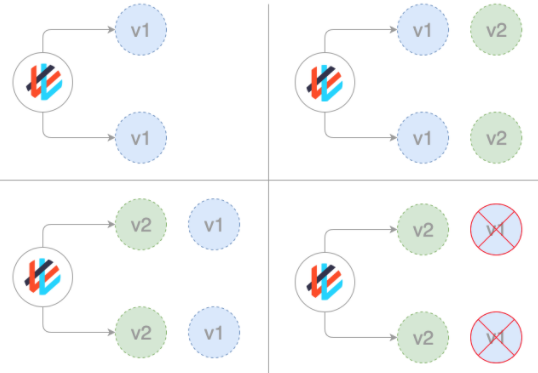
### name: cloudknowlwdge

### ports:

### - containerPort: 80

### Blue/ Green (or Red / Black) deployments

In a blue/green deployment strategy (sometimes referred to as red/black) the old version of the application (green) and the new version (blue) get deployed at the same time. When both of these are deployed, users only have access to the green; whereas, the blue is available to your QA team for test automation on a separate service or via direct port-forwarding.



apiVersion: apps/v1beta1

kind: Deployment

metadata:

name: awesomeapp-02

spec:

template:

metadata:

labels:

app: awesomeapp

version: "02"

After the new version has been tested and is signed off for release, the service is switched to the blue version with the old green version being scaled down:

apiVersion: v1

kind: Service

metadata:

name: awesomeapp

spec:

selector:

app: awesomeapp

version: "02"

...

### Canary

A canary deployment, or canary release, is a deployment pattern that allows you to roll out new code/features to a subset of users as an initial test.

## **Implement Canary Releases**

The initial steps for implementing [canary deployment](https://www.split.io/blog/canary-release-feature-flags/) are: create two clones of the production environment, have a load balancer that initially sends all traffic to one version, and creates new functionality in the other version. When you deploy the new software version, you shift some percentage – say, 10% – of your user base to the new version while maintaining 90% of users on the old version. If that 10% reports no errors, you can roll it out to gradually more users, until the new version is being used by everyone. If the 10% has problems, though, you can roll it right back, and 90% of your users will have never even seen the problem.

Note that infrastructure changes and configuration changes should always be tested with canaries because of their sensitivity.

## **Why Canary Deployment?**

Canary deployment benefits include zero downtime, easy rollout and quick rollback – plus the added safety from the gradual rollout process. It also has some drawbacks – the expense of maintaining multiple server instances, the difficult clone-or-don’t-clone database decision.

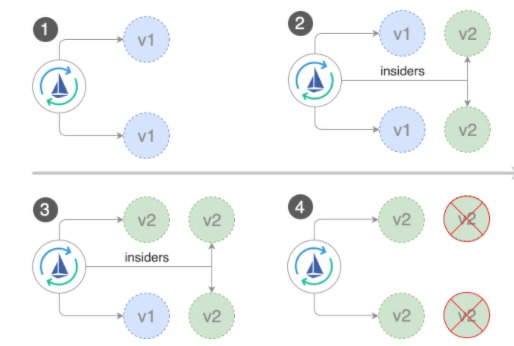
Typically, software development teams implement blue/green deployment when they’re sure the new version will work properly and want a simple, fast strategy to deploy it. Conversely, canary deployment is most useful when the development team isn’t as sure about the new version and they don’t mind a slower rollout if it means they’ll be able to catch the bugs.

**Dark deployments or A/B Deployments**

A dark deployment is another variation on the canary (that incidentally can also be handled by [Flagger)](https://docs.flagger.app/usage/ab-testing). The difference between a dark deployment and a canary is that dark deployments deal with features in the front-end rather than the backend as is the case with canaries.

Another name for dark deployment is A/B testing. Rather than launch a new feature for all users, you can release it to a small set of users. The users are typically unaware they are being used as testers for the new feature, hence the term “dark” deployment.

With the use of feature toggles and other tools, you can monitor how your user is interacting with the new feature and whether it is converting your users, or whether they find the new UI confusing and other types of metrics.



#### Flagger and A/B deployments

Besides weighted routing, Flagger can also route traffic to the canary based on HTTP match conditions. In an A/B testing scenario, you'll be using HTTP headers or cookies to target a certain segment of your users. This is particularly useful for front-end applications that require session affinity. Find out more in the Flagger docs.

**Kubernetes Services:**

**What is a Service in Kubernetes?**

A Service enables network access to a set of Pods in Kubernetes.

Services select Pods based on their labels. When a network request is made to the service, it selects all Pods in the cluster matching the service's selector, chooses one of them, and forwards the network request to it.

**What's the difference between a Service and a Deployment in Kubernetes?**

A deployment is responsible for keeping a set of pods running.

We could use a deployment without a service to keep a set of identical pods running in the Kubernetes cluster. The deployment could be scaled up and down and pods could be replicated. Each pod could be accessed individually via direct network requests (rather than abstracting them behind a service), but keeping track of this for a lot of pods is difficult.

A service is responsible for enabling network access to a set of pods.

We could also use a service without a deployment. We'd need to create each pod individually (rather than "all-at-once" like a deployment). Then our service could route network requests to those pods via selecting them based on their labels.

Services and Deployments are different, but they work together nicely

**What does ClusterIP, NodePort, and LoadBalancer mean?**

The type property in the Service's spec determines how the service is exposed to the network. It changes where a Service is able to be accessed from. The possible types are ClusterIP, NodePort, and LoadBalancer

* ClusterIP – The default value. The service is only accessible from within the Kubernetes cluster – you can’t make requests to your Pods from outside the cluster!
* NodePort – This makes the service accessible on a static port on each Node in the cluster. This means that the service can handle requests that originate from outside the cluster.
* LoadBalancer – The service becomes accessible externally through a cloud provider's load balancer functionality. GCP, AWS, Azure, and OpenStack offer this functionality. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service

**Web-deployment.yml----**

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-server

labels:

app: web-server

spec:

replicas: 1

strategy:

type: Recreate

selector:

matchLabels:

app: web-server

template:

metadata:

labels:

app: web-server

spec:

containers:

- image: nginx:1.7.1

name: nginx

ports:

- containerPort: 80

**DB deployment----**

apiVersion: apps/v1

kind: Deployment

metadata:

name: db-server

labels:

app: db-server

spec:

replicas: 1

strategy:

type: Recreate

selector:

matchLabels:

app: db-server

template:

metadata:

labels:

app: db-server

spec:

containers:

- image: nginx:1.7.1

name: nginx

ports:

- containerPort: 80

**kubectl exec -it web-server-84c8867645-dpvzm /bin/bash---run this command for access container**

**apt-get update**

**apt-get install curl**

**Then ping db-server ip from web-server**

**Curl db-server\_ip:port**

**kubectl expose deployment db-server --type=ClusterIP --port 80 --target-port=80---- create service**

**web-service---**

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: web-server

ports:

- protocol: TCP

port: 80

targetPort: 9376

**Install MetalLB-----**

kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.5/manifests/namespace.yaml

kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.5/manifests/metallb.yaml

# On first install only

kubectl create secret generic -n metallb-system memberlist --from-literal=secretkey="$(openssl rand -base64 128)"

**matallb.yml**

apiVersion: v1

kind: ConfigMap

metadata:

namespace: metallb-system

name: config

data:

config: |

address-pools:

- name: default

protocol: layer2

addresses:

- 192.168.43.206-192.168.43.250