1

Outline

- Managing Deployments
- Deployment/Release strategies Compared
- Implementing Deployment/Release strategies
- Scalability and HPA

2

Managing Deployments

3

Creating a Deployment

- **Deployment provide rollback functionality and update control.**
  - Updates are managed through the pod-template-hash label.
  - Each iteration creates a unique label that is assigned to both the ReplicaSet and subsequent Pods
- **Creating a declarative deployment of nginx 1.19.0**
  - kubectl apply -f deploy1-19-2.yaml
  - To see the Deployment Rollout status, run : kubectl rollout status deploy nginx-deployment
  - To see the Replicaset(rs) run: kubectl get rs
  - To see the labels automatically generated for each Pod, run: kubectl get pods --show-labels.
- **The pod-template-hash label is added by the Deployment controller to every ReplicaSet that a Deployment creates or adopts.**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19.0
        ports:
        - containerPort: 80
```


4

Updating a Deployment

- **Let's follow the steps given below to update your Deployment:**
  1. Let's update the nginx Pods to use the nginx:1.19.8 image instead of the nginx:1.19.0 image.

     ```
     kubectl set image deployment/nginx-deployment nginx=nginx:1.19.8 –record
     ```

     - Alternatively, you can edit the Deployment and change

       ```
       kubectl edit deployment.v1.apps/nginx-deployment
       ```

  2. To see the rollout status, run:

     ```
     kubectl rollout status deployment.v1.apps/nginx-deployment
     ```

     - Get details of your Deployment and look at the events section

       ```
       kubectl describe deployments
       ```

       - When you updated the Deployment, it created a new and scaled it up to 1 and then scaled down the old ReplicaSet to 2, so that at least 2 Pods were available and at most 4 Pods were created at all times.
       - It then continued scaling up and down the new and the old ReplicaSet, with the same rolling update strategy.
       - Finally, you'll have 3 available replicas in the new ReplicaSet, and the old ReplicaSet is scaled down to 0

5

Recording deployment actions

- **Some commands that modify a Deployment accept an optional --record flag**
  - Example: `kubectl set image deployment worker worker=alpine --record`
  - **The flag will store the command line in the Deployment**
    - Technically, using the annotation kubernetes.io/change-cause
    - It gets copied to the corresponding ReplicaSet (Allowing to keep track of which command created or promoted this ReplicaSet)
- **We can view this information with** `kubectl rollout history`
- **Updating the annotation directly**

  ```
  kubectl annotate deployment worker kubernetes.io/change-cause="Just for fun"
  ```

  - Check that our annotation shows up in the change history:

    ```
    kubectl rollout history deployment worker
    ```


6

Pausing & Resuming

- **You can pause a Deployment before triggering one or more updates and then resume it.**
  - This allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts
  - Use Case example :
    - Pause a running deployment
    - Update the image (no rollout started)
    - Resume the Deployment and observe a new ReplicaSet coming up with all the new updates
- **Pause by running the following command:**

```
kubectl rollout pause deployment.v1.apps/nginx-deployment
```

- **Resume the Deployment and observe a new ReplicaSet coming up with all the new updates:**

```
kubectl rollout resume deployment.v1.apps/nginx-deployment
```


7

Rolling Back a Deployment

- **You may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping.**
  - By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want.
- **Suppose that you made a typo while updating the Deployment, by putting the image name as nginx:1.161 instead of nginx:1.16.1:**

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161 --record=true
```

  - The output is similar to this:

```
deployment.apps/nginx-deployment image updated
```

  - The rollout gets stuck. You can verify it by checking the rollout status:

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

- **Rolling Back to a Previous Revision**
  - First, check the revisions of this Deployment:

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

  - Now you've decided to undo the current rollout and rollback to the previous revision:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

  - Check if the rollback was successful and the Deployment is running as expected, run:

```
kubectl get deployment nginx-deployment
```

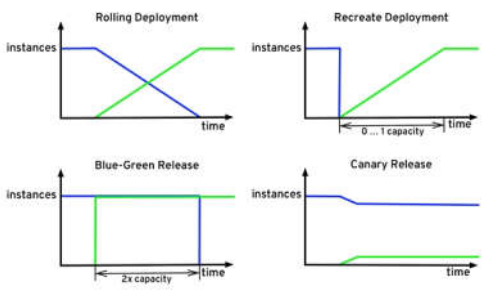

8

Deployment/Release Strategies Compared

9

Deployment/release Strategies

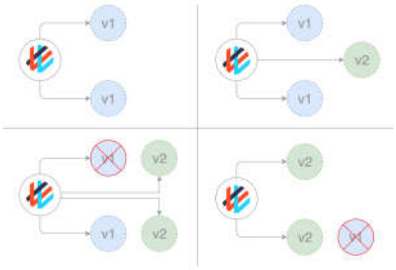
- **Choosing the right deployment procedure depends on the needs, we listed below some of the possible strategies to adopt:**
  - **Rolling Deployment**: works by slowly, one by one, replacing pods of the previous version of your application with pods of the new version without any cluster downtime.
  - **Recreate**: terminate the old version and release the new one
  - **Blue/green**: release a new version alongside the old version then switch traffic
  - **Canary**: release a new version to a subset of users, then proceed to a full rollout



10

Rolling Strategy

- **ReplicaSet is created with the new version of the application, then the number of replicas of the old version is decreased and the new version is increased until the correct number of replicas is reached.**

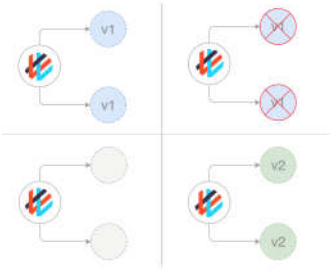


  - Pro:
    - version is slowly released across instances
    - convenient for stateful applications that can handle rebalancing of the data
  - Cons:
    - rollout/rollback can take time
    - supporting multiple APIs is hard
    - no control over traffic

11

Recreate Strategy

- **A deployment defined with a strategy of type Recreate will terminate all the running instances then recreate them with the newer version.**

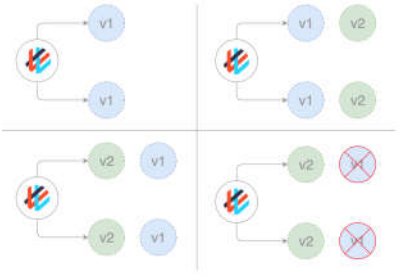


  - Pro:
    - version released for a subset of users
    - convenient for error rate and performance monitoring
    - fast rollback
  - Cons:
    - slow rollout
    - fine tuned traffic distribution can be expensive (99% A/ 1%B = 99 pod A, 1 pod B)

12

Blue/Green Strategy

- **After testing that the new version meets the requirements, we update the Kubernetes Service object that plays the role of load balancer to send traffic to the new version by replacing the version label in the selector field..**

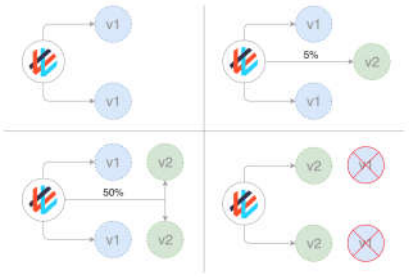


  - Pro:
    - instant rollout/rollback
    - avoid versioning issue, change the entire cluster state in one go
  - Cons:
    - requires double the resources
    - proper test of the entire platform should be done before releasing to production
    - handling stateful applications can be hard

13

Canary Strategy

- **Let the consumer do the testing. A canary deployment consists of routing a subset of users to a new functionality. Then after some time and if no error is detected, scale up the number of replicas of the new version and delete the old deployment..**



  - Pro:
    - instant rollout/rollback
    - avoid versioning issue, change the entire cluster state in one go
  - Cons:
    - requires double the resources
    - proper test of the entire platform should be done before releasing to production
    - handling stateful applications can be hard

14

Implementing Deployment Strategies

15

The strategy field of a deployment

- **Strategy: describes the method used to update the deployment**
  - **Recreate** is pretty self explanatory, All existing Pods are killed before new ones are created
  - **RollingUpdate** cycles through updating the Pods according to the parameters: **maxSurge** and **maxUnavailable**
- **maxSurge**
  - Optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The default value is 25%.
- **maxUnavailable**
  - Optional field that specifies the maximum number of Pods that can be unavailable during the update process. The default value is **25%**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
  template:
    <pod template>
```


16

Implemnting Green/Blue strategy

- **In a blue/green deployment strategy the old version of the application (green) and the new version (blue) get deployed at the same time.**
  - After testing that the new version meets the requirements, we update the Kubernetes Service object that plays the role of load balancer to send traffic to the new version by replacing the version label in the selector field.

```yaml
apiVersion: v1
kind: Service
metadata:
 name: my-app
 labels:
   app: my-app
spec:
 type: NodePort
 ports:
 - name: http
   port: 8080
   targetPort: 8080

 # Note here that we match both the app and the version.
 # When switching traffic, we update the label “version” with
 # the appropriate value, ie: v2.0.0
 selector:
   app: my-app
   version: v1.0.0
```


Implementing Canary Strategy

- **Canary deployments are a bit like blue/green deployments, but are more controlled and use a more ‘progressive delivery’ phased-in approach.**
  - While this strategy can be done just using Kubernetes resources by replacing old and new pods, it is much more convenient and easier to implement this strategy with a service mesh like Istio.
- **In the following example we use two ReplicaSets side by side, version A with three replicas (75% of the traffic), version B with one replica (25% of the traffic).**
  - Truncated deployment manifest version A:

```yaml
spec:
  replicas: 3
```

  - Truncated deployment manifest version B, note that we only start one replica of the application:

```yaml
spec:
  replicas: 1
```

Scalability and HPA

19

Scaling Up and Down Applications

- **Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources.**
  - Scaling will increase the number of Pods to the new desired state. Kubernetes also supports autoscaling of Pods
- **You can scale a Deployment by using the following command:**

```
kubectl scale deployment nginx-deployment --replicas=10
```

- **To check the deployment, use the get command**

```
kubectl get deployments nginx-deployment
```

- **Running multiple instances of an application will require a way to distribute the traffic to all of them.**
  - Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment.
  - Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

20

Managing Rollouts

What is the Horizontal Pod Autoscaler, or HPA?

- **It is a controller that can perform horizontal scaling automatically**
- **Horizontal scaling**
  - Changing the number of replicas (adding/removing pods)
- **Vertical scaling**
  - Changing the size of individual replicas (increasing/reducing CPU and RAM per pod)
- **Cluster scaling**
  - Changing the size of the cluster (adding/removing nodes
- **HPA's Principle of operation**
  - Each HPA resource (or "policy") specifies:
    - which object to monitor and scale (e.g. a Deployment, ReplicaSet...)
    - min/max scaling ranges (the max is a safety limit!)
    - a target resource usage (e.g. the default is CPU=80%)
  - The HPA continuously monitors the CPU usage for the related object
  - It scales the related object up/down to this target number of pods

21

Managing Rollouts

Creating Horizontal Pod Autoscaler

- **Imperative style**

kubectl autoscale deployment nginx-deployment --cpu-percent=50 --min=1 --max=10

  - This command creates an Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment.
  - Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50%

- **The declative style use the object HorizontalPodAutoscaler**

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 100Mi
```


22