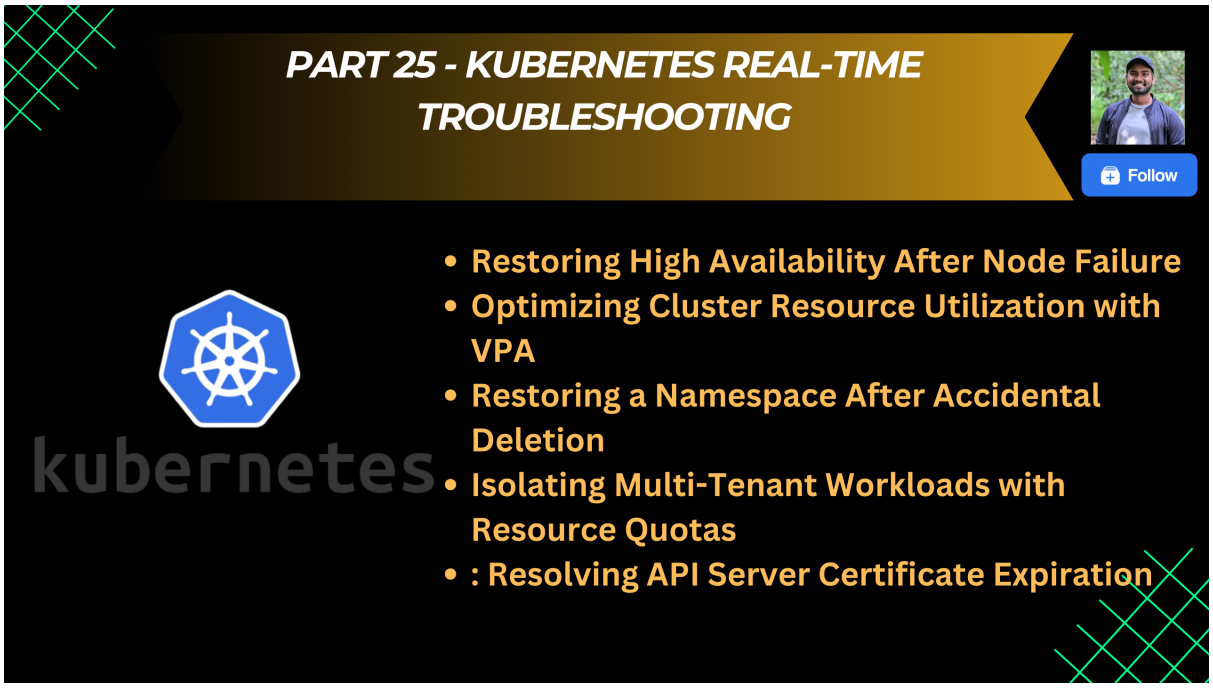
Part 25: Kubernetes Real-Time Troubleshooting

Introduction 🌐

Welcome to the world of Kubernetes troubleshooting, where every challenge is an opportunity to sharpen your skills and emerge victorious. Join us as we embark on a journey through common real-time scenarios, unraveling mysteries, and uncovering solutions along the way.



Scenario 121: Restoring High Availability After Node Failure

Scenario

A critical worker node, cluster5-worker3, in your production cluster crashes unexpectedly, causing several pods to become unschedulable. The cluster's high availability is compromised, and critical workloads in the project-owl namespace are at risk. You need to diagnose the node failure, restore it or mitigate its impact, and ensure workloads are rescheduled to maintain HA without adding new nodes.

Solution

Context:

kubectl config use-context k8s-c12-prod

Steps:

1. **Confirm Node Failure:**
    - Check node status:

kubectl get nodes

    - Look for cluster5-worker3 marked as NotReady.
    - Describe the node for details:

kubectl describe node cluster5-worker3

    - Example issue: Kubelet stopped posting node status.
2. **Inspect Affected Pods:**
    - List pods scheduled on the failed node:

kubectl get pods -A --field-selector=spec.nodeName=cluster5-worker3

    - Note critical pods in project-owl (e.g., owl-api).
3. **Attempt Node Recovery:**
    - SSH into cluster5-worker3 (if accessible):

ssh cluster5-worker3

    - Check kubelet status:

systemctl status kubelet

    - Restart kubelet if stopped:

sudo systemctl restart kubelet

    - If the node remains unresponsive, proceed to mitigation.
4. **Cordon and Drain the Failed Node:**
    - Mark the node as unschedulable:

kubectl cordon cluster5-worker3

    - Evict pods to other nodes:

kubectl drain cluster5-worker3 --ignore-daemonsets --delete-emptydir-data

5. **Ensure Pod Rescheduling:**
    - Verify pod distribution:

kubectl get pods -n project-owl -o wide

    - Update critical deployments with anti-affinity to prevent single-node failures:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: owl-api
  namespace: project-owl
spec:
  template:
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchLabels:
                  app: owl-api
              topologyKey: kubernetes.io/hostname
```

- Apply the update:

```
kubectl apply -f owl-api-deployment.yaml
```

6. **Monitor Cluster Health:**
   - Confirm all nodes are healthy (except cluster5-worker3):

```
kubectl get nodes
```

   - Check pod status:

```
kubectl get pods -n project-owl
```

   - Verify no pending pods:

```
kubectl get events -n project-owl
```

7. **Plan Node Replacement:**
   - If cluster5-worker3 cannot be recovered, label it for removal:

```
kubectl label node cluster5-worker3 node-role.kubernetes.io/failed=true
```

   - Document steps to replace the node later:

kubeadm join <control-plane-ip>:6443 --token <token> --discovery-token-ca-cert-hash <hash>

Outcome

- Critical workloads in project-owl are rescheduled to healthy nodes, restoring high availability.
- The failed node is isolated, preventing further scheduling issues.
- Anti-affinity rules ensure better workload distribution, reducing future risks.

Scenario 122: Optimizing Cluster Resource Utilization with VPA

Scenario

Your Kubernetes cluster in the project-tiger namespace is experiencing inefficient resource utilization. Some pods are overprovisioned, wasting CPU and memory, while others are underprovisioned, causing performance bottlenecks. You need to implement Vertical Pod Autoscaling (VPA) to dynamically adjust resource requests and limits, optimize utilization, and maintain application stability.

Solution

Context:

kubectl config use-context k8s-c13-dev

Steps:

1. **Assess Current Resource Utilization:**
    - Check pod resource usage:

kubectl top pod -n project-tiger

    - Identify overprovisioned (e.g., tiger-web with high limits) and underprovisioned pods (e.g., tiger-workerwith frequent OOM kills).
2. **Install VPA:**
    - Deploy the VPA controller:

kubectl apply -f https://github.com/kubernetes/autoscaler/releases/latest/download/vertical-pod-autoscaler.yaml

    - Verify VPA components:

kubectl get pods -n kube-system | grep vpa

3. **Create VPA for Critical Deployments:**
    - Define a VPA for tiger-web:

apiVersion: autoscaling.k8s.io/v1

```
kind: VerticalPodAutoscaler
metadata:
  name: tiger-web-vpa
  namespace: project-tiger
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: tiger-web
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
    - containerName: "*"
      minAllowed:
        cpu: 100m
        memory: 128Mi
      maxAllowed:
        cpu: 1
        memory: 1Gi
```

- Apply the VPA:

```
kubectl apply -f tiger-web-vpa.yaml
```

- Repeat for tiger-worker:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: tiger-worker-vpa
  namespace: project-tiger
spec:
  targetRef:
```

```
  apiVersion: apps/v1
  kind: Deployment
  name: tiger-worker
 updatePolicy:
  updateMode: "Auto"
 resourcePolicy:
  containerPolicies:
  - containerName: "*"
    minAllowed:
      cpu: 200m
      memory: 256Mi
    maxAllowed:
      cpu: 2
      memory: 2Gi
```

- Apply:

```
kubectl apply -f tiger-worker-vpa.yaml
```

4. **Monitor VPA Recommendations:**
   - Check VPA suggestions:

```
kubectl describe vpa tiger-web-vpa -n project-tiger
```

   - Example: Recommends reducing tiger-web CPU to 300m and increasing tiger-worker memory to 512Mi.

5. **Validate Autoscaling:**
   - Observe pod restarts with updated resources:

```
kubectl get pods -n project-tiger -o wide
```

   - Confirm resource allocation:

```
kubectl describe pod -n project-tiger -l app=tiger-web
```

6. **Test Application Stability:**
   - Simulate load on tiger-web:

```
kubectl run load-test --image=busybox -n project-tiger --rm -it -- /bin/sh
while true; do wget -q -O- http://tiger-web:8080; done
```

- Check for performance issues:

kubectl logs -n project-tiger -l app=tiger-web

7. **Fine-Tune VPA Policies:**
    - Adjust VPA to use “Recommendation” mode for manual review:

spec:

updatePolicy:

updateMode: "Off"

- Apply and review recommendations before applying:

kubectl apply -f tiger-web-vpa.yaml

kubectl describe vpa tiger-web-vpa -n project-tiger

Outcome

- Resource utilization in project-tiger is optimized with VPA adjusting CPU and memory dynamically.
- Overprovisioned pods are scaled down, and underprovisioned pods are scaled up, improving efficiency.
- Application performance is stable under varying loads, with no bottlenecks or waste.

Scenario 123: Restoring a Namespace After Accidental Deletion

Scenario

A junior administrator accidentally deletes the project-hawk namespace, which hosts a critical application with persistent data. The application pods, services, and persistent volume claims (PVCs) are gone, causing a service outage. You need to restore the namespace and its resources using a backup, recover the persistent data, and verify the application is fully operational without redeploying from scratch.

Solution

Context:

kubectl config use-context k8s-c14-prod

Steps:

1. **Confirm Namespace Deletion:**
    - Verify the namespace is missing:

kubectl get namespaces

- Check for lingering resources:

kubectl get pvc,pv -A | grep project-hawk

2. **Locate the Latest Backup:**
   - Assuming Velero is used for backups, list available backups:

velero backup get

   - Identify the most recent backup for project-hawk (e.g., hawk-backup-20250412).

3. **Restore the Namespace:**
   - Initiate a Velero restore for project-hawk:

velero restore create hawk-restore-20250413 --from-backup hawk-backup-20250412 --include-namespaces project-hawk

   - Monitor the restore process:

velero restore describe hawk-restore-20250413

4. **Verify Restored Resources:**
   - Check the namespace:

kubectl get namespace project-hawk

   - List restored resources:

kubectl get pods,services,deployments,pvc -n project-hawk

   - Confirm persistent volumes are bound:

kubectl get pvc -n project-hawk -o wide

5. **Validate Persistent Data:**
   - Access a restored pod (e.g., hawk-db):

kubectl exec -n project-hawk -it hawk-db-0 -- /bin/bash

   - Verify data integrity (e.g., check database tables):

psql -U hawk_user -d hawk_db -c "\dt"

6. **Restart Application Pods:**
   - If pods are not running, delete to trigger recreation:

kubectl delete pod -n project-hawk -l app=hawk

   - Verify pod status:

kubectl get pods -n project-hawk

7. **Test Application Functionality:**
    - Check service accessibility:

kubectl get svc -n project-hawk

    - Test the application endpoint:

curl http://hawk-app.project-hawk.svc.cluster.local:8080

    - Monitor logs for errors:

kubectl logs -n project-hawk -l app=hawk

8. **Prevent Future Accidents:**
    - Apply a namespace-level RBAC to restrict deletions:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: restricted-admin
  namespace: project-hawk
rules:
- apiGroups: [""]
  resources: ["pods", "services", "deployments"]
  verbs: ["get", "list", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: restricted-admin-binding
  namespace: project-hawk
subjects:
```

- kind: Group

  name: junior-admins

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: Role

  name: restricted-admin

  apiGroup: rbac.authorization.k8s.io

    - Apply the RBAC:

kubectl apply -f restricted-admin-rbac.yaml

Outcome

- The project-hawk namespace and its resources are fully restored from the backup.
- Persistent data is intact, and the application is operational with no downtime post-restore.
- RBAC policies prevent unauthorized namespace deletions, enhancing cluster security.

Scenario 124: Isolating Multi-Tenant Workloads with Resource Quotas

Scenario

Your cluster hosts multiple teams in the project-lemur and project-gorilla namespaces, but resource contention is causing performance issues. The project-lemur namespace is consuming excessive CPU and memory, starving project-gorilla workloads. You need to implement resource quotas and limits to isolate tenant workloads, ensure fair resource allocation, and stabilize both namespaces without restarting existing pods.

Solution

Context:

kubectl config use-context k8s-c15-stage

Steps:

1. **Analyze Current Resource Usage:**
    - Check resource consumption by namespace:

kubectl top pod -n project-lemur

kubectl top pod -n project-gorilla

    - Confirm project-lemur is overconsuming (e.g., high CPU usage).

2. **Define Resource Quotas for project-lemur:**
    - Create a quota to cap resource usage:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: lemur-quota
  namespace: project-lemur
spec:
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "6"
    limits.memory: "12Gi"
    pods: "20"
```

    - Apply the quota:

```
bash

kubectl apply -f lemur-quota.yaml
```

3. **Define Resource Quotas for project-gorilla:**
    - Create a balanced quota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gorilla-quota
  namespace: project-gorilla
spec:
  hard:
    requests.cpu: "3"
```

requests.memory: "6Gi"

limits.cpu: "5"

limits.memory: "10Gi"

pods: "15"

- Apply the quota:

kubectl apply -f gorilla-quota.yaml

4. **Set Default LimitRanges:**
   - Apply a LimitRange to enforce per-pod constraints in both namespaces:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: project-lemur
spec:
  limits:
  - type: Container
    default:
      cpu: "500m"
      memory: "512Mi"
    defaultRequest:
      cpu: "100m"
      memory: "128Mi"
---
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: project-gorilla
spec:
```

```
  limits:
  - type: Container
    default:
      cpu: "400m"
      memory: "384Mi"
    defaultRequest:
      cpu: "80m"
      memory: "96Mi"
```

- Apply the LimitRanges:

```
kubectl apply -f default-limits.yaml
```

5. **Validate Quota Enforcement:**
   - Attempt to deploy a pod exceeding the quota in project-lemur:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-overquota
  namespace: project-lemur
spec:
  containers:
  - name: test
    image: nginx
    resources:
      requests:
        cpu: "3"
        memory: "4Gi"
```

- Confirm it's rejected:

```
kubectl apply -f test-overquota.yaml
```

- Check quota status:

kubectl describe quota lemur-quota -n project-lemur

6. **Monitor Performance:**
    - Verify resource allocation:

kubectl top pod -n project-lemur

kubectl top pod -n project-gorilla

    - Ensure project-gorilla pods are no longer starved:

kubectl get pods -n project-gorilla -o wide

7. **Educate Teams:**
    - Document quota policies for transparency:

echo "Resource Quotas: project-lemur (CPU: 4-6, Mem: 8-12Gi), project-gorilla (CPU: 3-5, Mem: 6-10Gi)" > tenant-guidelines.txt

Outcome

- Resource quotas isolate project-lemur and project-gorilla, preventing contention.
- Both namespaces operate within defined limits, ensuring fair resource distribution.
- Performance issues are resolved, and teams can deploy workloads confidently.

Scenario 125: Resolving API Server Certificate Expiration

Scenario

The Kubernetes API server certificate in your cluster has expired, causing kubectl commands to fail with errors like x509: certificate has expired or is not yet valid. Applications are still running, but administrative access is blocked. You need to renew the certificate, restore API server access, and verify cluster functionality without downtime.

Solution

Context:

bash

kubectl config use-context k8s-c16-prod

Steps:

1. **Confirm Certificate Issue:**
    - Attempt to access the API server:

kubectl get nodes

    - Expected error: x509: certificate has expired.

- Check certificate validity:

ssh cluster6-master1

openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -dates

- Note the expiration date.

2. **Back Up Current Certificates:**
   - Archive existing certificates:

sudo mkdir /etc/kubernetes/pki/backup

sudo cp -r /etc/kubernetes/pki/* /etc/kubernetes/pki/backup/

3. **Renew API Server Certificate:**
   - Generate a new certificate with kubeadm:

sudo kubeadm certs renew apiserver

   - Verify the new certificate:

openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -dates

4. **Restart Control Plane Components:**
   - Restart the API server, controller manager, and scheduler:

sudo docker ps | grep kube-apiserver | awk '{print $1}' | xargs docker restart

sudo docker ps | grep kube-controller-manager | awk '{print $1}' | xargs docker restart

sudo docker ps | grep kube-scheduler | awk '{print $1}' | xargs docker restart

   - If using systemd:

sudo systemctl restart kubelet

5. **Update Kubeconfig:**
   - Regenerate the admin kubeconfig:

sudo kubeadm kubeconfig user --org system:masters --client-name kubernetes-admin > ~/.kube/config

   - Verify access:

kubectl get nodes

6. **Validate Cluster Health:**
   - Check control plane pods:

kubectl get pods -n kube-system

   - Ensure workloads are unaffected:

kubectl get pods -A

- Test application connectivity:

curl http://app.example.com

7. **Automate Future Renewals:**
    - Schedule certificate renewal checks:

echo "0 0 1 * * kubeadm certs check-expiration" | crontab -

- Enable auto-renewal (if supported):

sudo kubeadm certs renew all --auto-renew

Outcome

- The API server certificate is renewed, restoring administrative access.
- Cluster operations continue without downtime or application disruption.
- Automated checks prevent future certificate expiration issues.

In the up-coming parts, we will discussion on more troubleshooting steps for the different Kubernetes based scenarios. So, stay tuned for the and follow @Prasad Suman Mohan for more such posts.