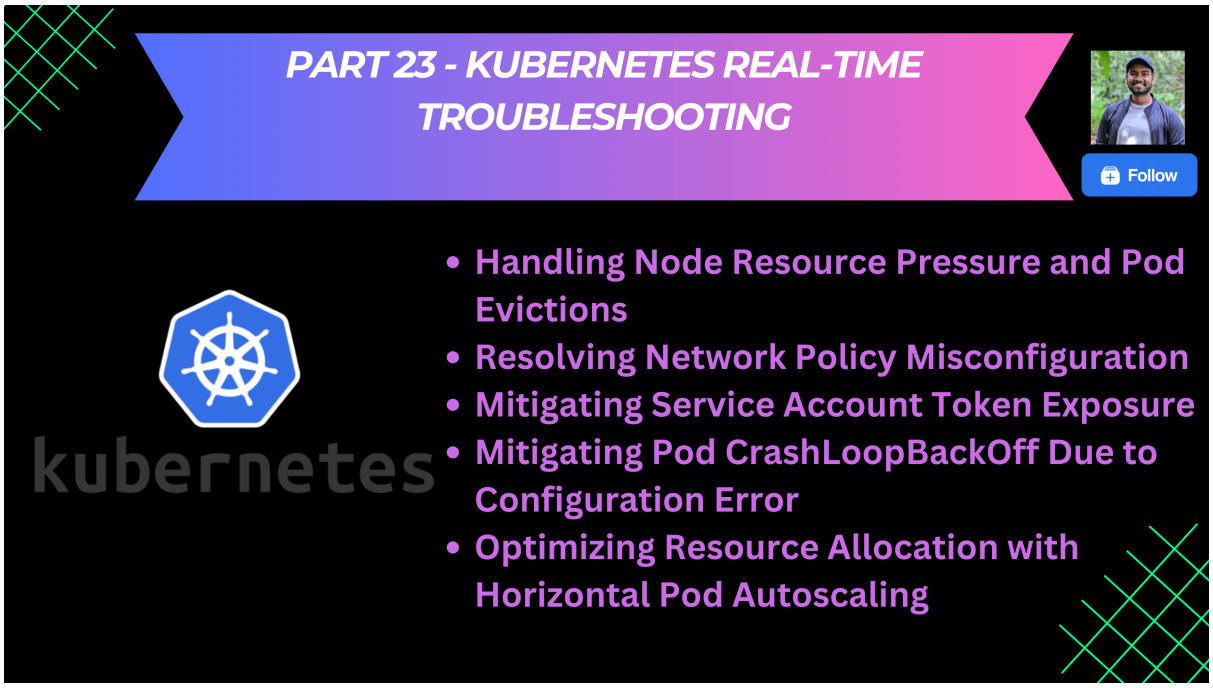
Part 23: Kubernetes Real-Time Troubleshooting

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 111: Handling Node Resource Pressure and Pod Evictions

Scenario
Your Kubernetes cluster is running critical workloads, but one node, cluster3-worker2, is reporting resource pressure due to high memory usage. As a result, the kubelet is evicting low-priority pods, causing service disruptions. You need to identify the cause, stabilize the node, and prevent future evictions without adding new nodes.

Solution
Context:

kubectl config use-context k8s-c3-prod

Steps:

1. **Identify the Node Under Pressure:**
    - Check node status to confirm resource pressure:

kubectl describe node cluster3-worker2

    - Look for conditions like MemoryPressure or DiskPressure in the output.
2. **Inspect Pod Resource Usage:**
    - List pods on the affected node and their resource consumption:

kubectl top pod --all-namespaces --field-selector=spec.nodeName=cluster3-worker2

    - Identify pods consuming excessive memory (e.g., a pod named resource-hog).
3. **Set Resource Limits for Problematic Pods:**
    - Edit the pod's deployment to enforce memory limits:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: resource-hog-deployment
  namespace: prod
spec:
  template:
    spec:
      containers:
      - name: resource-hog
        image: nginx:1.14
        resources:
          limits:
            memory: "500Mi"
          requests:
            memory: "200Mi"
```

    - Apply the updated configuration:

kubectl apply -f resource-hog-deployment.yaml

4. **Adjust Pod Priority to Protect Critical Workloads:**
    - Create a PriorityClass for critical pods:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "Priority class for critical workloads"
```

    - Apply the PriorityClass:

kubectl apply -f high-priority.yaml

    - Assign the PriorityClass to critical deployments:

```
spec:
  template:
    spec:
```

priorityClassName: high-priority

5. **Tune Kubelet Eviction Thresholds (Optional):**
    - SSH into cluster3-worker2:

ssh cluster3-worker2

    - Modify kubelet configuration at /var/lib/kubelet/config.yaml:

evictionHard:
  memory.available: "200Mi"

    - Restart the kubelet:

systemctl restart kubelet

6. **Verify Stability:**
    - Confirm no further evictions:

kubectl get events --field-selector involvedObject.kind=Pod

    - Check node status again:

kubectl describe node cluster3-worker2

Outcome

- The node cluster3-worker2 is stabilized with no ongoing resource pressure.
- Critical pods are protected from eviction due to their high priority.
- Resource-hungry pods are constrained, preventing future disruptions.

Scenario 112: Resolving Network Policy Misconfiguration

Scenario

A new network policy in the project-zebra namespace is blocking traffic to a critical application pod, api-service, causing API requests to fail. You need to diagnose the issue, fix the network policy, and ensure the application is accessible only to specific pods within the same namespace.

Solution

Context:

kubectl config use-context k8s-c4-stage

Steps:

1. **Verify Network Policy Impact:**
    - Check existing network policies in the namespace:

kubectl get networkpolicy -n project-zebra

    - Inspect the policy (e.g., restrict-api-access):

kubectl describe networkpolicy restrict-api-access -n project-zebra

2. **Test Connectivity:**
    - Deploy a temporary pod to test connectivity to api-service:

kubectl run test-pod --image=busybox -n project-zebra --rm -it -- /bin/sh

    - Attempt to reach api-service:

wget http://api-service:8080

    - Confirm the connection fails due to the policy.

3. **Update the Network Policy:**

- Edit the network policy to allow traffic from specific pods (e.g., those labeled app: frontend):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-api-access
  namespace: project-zebra
spec:
  podSelector:
    matchLabels:
      app: api-service
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 8080
```

- Apply the updated policy:

```
kubectl apply -f restrict-api-access.yaml
```

4. **Retest Connectivity:**
   - Label the test pod to match the policy:

```
kubectl label pod test-pod app=frontend -n project-zebra
```

   - Retry the connection:

```
wget http://api-service:8080
```

   - Confirm success.
5. **Validate Policy Scope:**
   - Ensure pods without the app: frontend label cannot connect:

```
kubectl run unauthorized-pod --image=busybox -n project-zebra --rm -it -- /bin/sh
wget http://api-service:8080
```

   - Confirm failure for unauthorized access.

Outcome

- The api-service pod is accessible only to pods labeled app: frontend in the project-zebra namespace.
- The network policy is correctly enforced, restoring application functionality while maintaining security.

Scenario 113: Mitigating Pod CrashLoopBackOff Due to Configuration Error

Scenario

A pod, data-processor, in the project-lion namespace is stuck in a CrashLoopBackOff state. Logs indicate a configuration error in the application's environment variables. You need to diagnose the issue, fix the configuration, and ensure the pod runs stably.

Solution

Context:

kubectl config use-context k8s-c5-dev

Steps:

1. **Check Pod Status:**
    - Inspect the pod's status:

kubectl get pods -n project-lion

    - Confirm data-processor is in CrashLoopBackOff.
2. **Review Pod Logs:**
    - Fetch logs to identify the error:

kubectl logs data-processor -n project-lion

    - Example error: Invalid DATABASE_URL format.
3. **Inspect Pod Configuration:**
    - View the pod's spec:

kubectl describe pod data-processor -n project-lion

    - Note the environment variable DATABASE_URL is malformed (e.g., missing protocol).
4. **Update the Deployment:**
    - Edit the deployment to fix the environment variable:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-processor
  namespace: project-lion
spec:
  template:
    spec:
      containers:
      - name: data-processor
        image: processor:1.2
        env:
        - name: DATABASE_URL
          value: "postgres://user:password@db-host:5432/dbname"
```

    - Apply the updated deployment:

kubectl apply -f data-processor-deployment.yaml

5. **Verify Pod Stability:**
    - Check the pod's status:

kubectl get pods -n project-lion

    - Confirm the pod is Running.
    - Recheck logs to ensure no errors:

kubectl logs data-processor -n project-lion

6. **Prevent Future Issues:**
    - Store sensitive variables in a Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
```

```
  namespace: project-lion
type: Opaque
data:
  database-url:
cG9zdGdyZXM6Ly91c2VyOnBhc3N3b3JkQGRiLWhvc3Q6NTQzMi9kYm5hbWU= # base64 encoded
```

- Reference the Secret in the deployment:

```
env:
- name: DATABASE_URL
  valueFrom:
    secretKeyRef:
      name: db-credentials
      key: database-url
```

- Apply the Secret and updated deployment:

```
kubectl apply -f db-credentials.yaml
kubectl apply -f data-processor-deployment.yaml
```

Outcome

- The data-processor pod is running stably without crashing.
- Configuration errors are resolved, and sensitive data is securely managed via a Kubernetes Secret.

Scenario 114: Mitigating Service Account Token Exposure

Scenario

A security audit reveals that a ServiceAccount token for deployer-sa in the project-giraffe namespace is exposed in a pod's logs, posing a risk of unauthorized cluster access. You need to rotate the token, secure the pod's logging, and prevent future exposures while ensuring the application remains functional.

Solution

Context:

```
kubectl config use-context k8s-c6-sec
```

Steps:

1. **Identify the Affected ServiceAccount:**
   - Verify the ServiceAccount in the namespace:

```
kubectl get serviceaccount deployer-sa -n project-giraffe
```

   - Check which pods use this ServiceAccount:

```
kubectl get pods -n project-giraffe -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.spec.serviceAccountName}{"\n"}{end}' | grep deployer-sa
```

   - Example output: deployer-pod deployer-sa.

2. **Inspect Pod Logs for Token Exposure:**
    - View logs of the affected pod (e.g., deployer-pod):

```
kubectl logs deployer-pod -n project-giraffe
```

    - Confirm the token appears (e.g., a JWT string like eyJhb...).
3. **Rotate the ServiceAccount Token:**
    - Delete the existing secret associated with the ServiceAccount:

```
kubectl delete secret -n project-giraffe $(kubectl get secret -n project-giraffe -o name | grep deployer-sa)
```

    - Kubernetes automatically regenerates a new token. Verify:

```
kubectl get secret -n project-giraffe | grep deployer-sa
```

4. **Secure Pod Logging:**
    - Edit the pod's deployment to prevent token exposure:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployer-deployment
  namespace: project-giraffe
spec:
  template:
    spec:
      containers:
      - name: deployer
        image: deployer:1.0
        env:
        - name: LOG_LEVEL
          value: "INFO" # Ensure sensitive data is not logged
      automountServiceAccountToken: false # Disable token mounting
```

    - Apply the updated deployment:

```
kubectl apply -f deployer-deployment.yaml
```

5. **Restrict ServiceAccount Permissions:**

- Create a minimal Role to limit deployer-sa permissions:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: deployer-restricted
  namespace: project-giraffe
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "update"]
```

- Bind the Role to the ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: deployer-restricted-binding
  namespace: project-giraffe
subjects:
- kind: ServiceAccount
  name: deployer-sa
  namespace: project-giraffe
roleRef:
  kind: Role
  name: deployer-restricted
  apiGroup: rbac.authorization.k8s.io
```

- Apply both:

```
kubectl apply -f deployer-role.yaml
kubectl apply -f deployer-rolebinding.yaml
```

6. **Verify Security:**
   - Restart the pod and check logs:

kubectl delete pod -l app=deployer -n project-giraffe

kubectl logs -l app=deployer -n project-giraffe

- Confirm no token appears.
- Test ServiceAccount access:

kubectl auth can-i get deployments --as=system:serviceaccount:project-giraffe:deployer-sa -n project-giraffe

Outcome

- The deployer-sa token is rotated, eliminating the security risk.
- The pod no longer logs sensitive data or mounts the ServiceAccount token.
- The ServiceAccount has minimal permissions, enhancing cluster security.

Scenario 115: Optimizing Resource Allocation with Horizontal Pod Autoscaling

Scenario

A web application in the project-cheetah namespace experiences sporadic performance issues due to fluctuating traffic. The deployment web-app lacks autoscaling, causing slowdowns during peak loads and wasted resources during low traffic. You need to implement Horizontal Pod Autoscaling (HPA) to dynamically adjust replicas based on CPU usage, ensuring optimal performance and cost efficiency.

Solution

Context:

kubectl config use-context k8s-c7-prod

Steps:

1. **Verify Current Resource Usage:**
   - Check the deployment's resource consumption:

kubectl top pod -l app=web-app -n project-cheetah

   - Confirm Metrics Server is running:

kubectl get deployment metrics-server -n kube-system

2. **Set Resource Requests and Limits:**
   - Update the web-app deployment to define CPU requests and limits:

apiVersion: apps/v1

kind: Deployment

metadata:

```
  name: web-app
  namespace: project-cheetah
spec:
  template:
    spec:
      containers:
      - name: web-app
        image: web-app:2.1
        resources:
          requests:
            cpu: "200m"
            memory: "256Mi"
          limits:
            cpu: "500m"
            memory: "512Mi"
```

- Apply the updated deployment:

```
kubectl apply -f web-app-deployment.yaml
```

3. **Create a Horizontal Pod Autoscaler:**
   - Define an HPA to scale based on CPU utilization:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
  namespace: project-cheetah
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
```

```
minReplicas: 2
maxReplicas: 10
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 70
```

- Apply the HPA:

```
kubectl apply -f web-app-hpa.yaml
```

4. **Simulate Load to Test Autoscaling:**
   - Deploy a load generator pod:

```
kubectl run load-generator --image=busybox -n project-cheetah --rm -it -- /bin/sh
```

   - Generate traffic to web-app:

```
while true; do wget -q -O- http://web-app.project-cheetah.svc.cluster.local; done
```

   - Monitor HPA status:

```
kubectl get hpa web-app-hpa -n project-cheetah --watch
```

5. **Validate Autoscaling Behavior:**
   - Check pod scaling:

```
kubectl get pods -n project-cheetah -l app=web-app
```

   - Stop the load generator and verify scale-down:

```
kubectl delete pod load-generator -n project-cheetah
kubectl get hpa web-app-hpa -n project-cheetah
```

Outcome

- The web-app deployment dynamically scales between 2 and 10 replicas based on CPU usage.
- Performance issues are resolved during peak traffic, and resources are conserved during low traffic.
- The application remains stable and cost-efficient.

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.