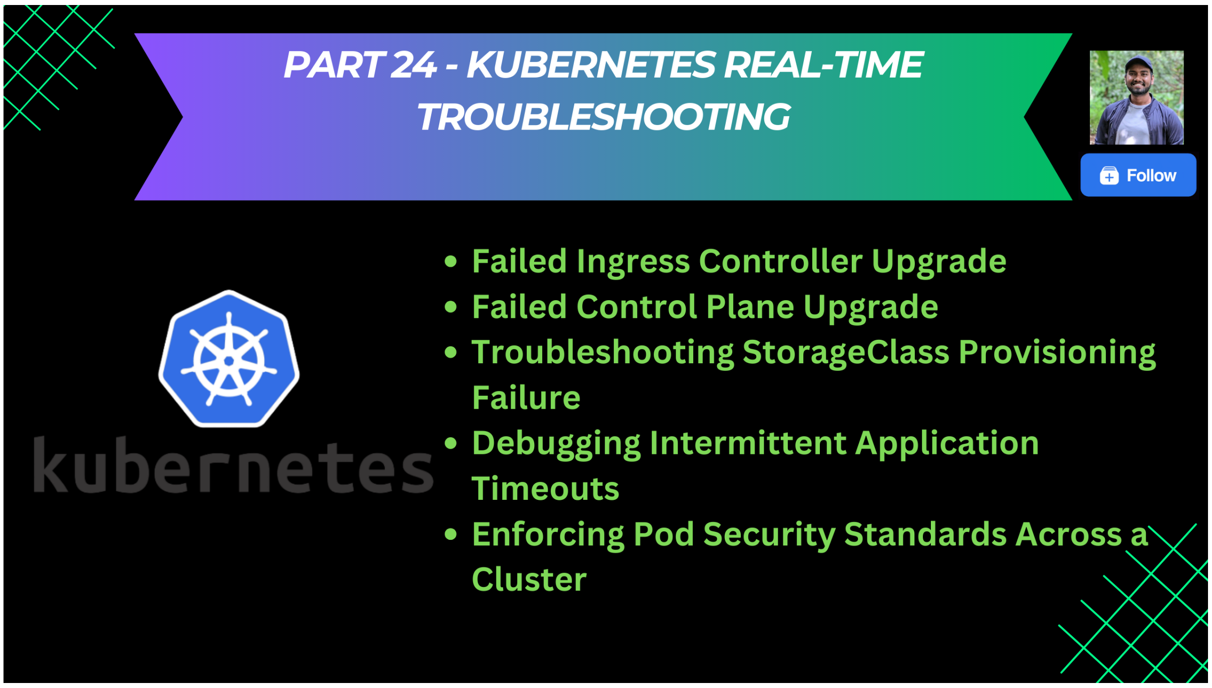
Part 24: Kubernetes Real-Time Troubleshooting

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.


PART 24 - KUBERNETES REAL-TIME TROUBLESHOOTING

- Failed Ingress Controller Upgrade
- Failed Control Plane Upgrade
- Troubleshooting StorageClass Provisioning Failure
- Debugging Intermittent Application Timeouts
- Enforcing Pod Security Standards Across a Cluster

kubernetes
Follow

Scenario 116: Recovering from Failed Ingress Controller Upgrade

Scenario

After upgrading the NGINX Ingress Controller in the ingress-nginx namespace, external traffic to services in the project-rhino namespace is failing with 502 errors. Logs indicate a configuration issue in the new controller version. You need to diagnose the problem, roll back to a stable version, and ensure traffic is restored without downtime.

Solution

Context:

```
kubectl config use-context k8s-c8-stage
```

Steps:

1. **Verify Ingress Controller Status:**
   - Check the Ingress Controller pods:

```
kubectl get pods -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx
```

   - Confirm the version (e.g., 1.9.0):

```
kubectl describe pod -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx | grep Image
```

2. **Inspect Logs for Errors:**
   - View logs of the Ingress Controller pod:

```
kubectl logs -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx
```

   - Example error: Invalid configuration: missing default backend.
3. **Check Ingress Resources:**
   - List Ingress resources in the affected namespace:

```
kubectl get ingress -n project-rhino
```

   - Inspect a specific Ingress (e.g., rhino-app-ingress):

```
kubectl describe ingress rhino-app-ingress -n project-rhino
```

   - Note any misconfigurations (e.g., outdated annotations).
4. **Roll Back the Ingress Controller:**
   - Identify the previous stable version (e.g., 1.8.0) from documentation or deployment history.
   - Update the Ingress Controller deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  template:
    spec:
      containers:
      - name: controller
        image: k8s.gcr.io/ingress-nginx/controller:v1.8.0
```

   - Apply the rollback:

```
kubectl apply -f ingress-nginx-deployment.yaml
```

5. **Fix Ingress Configuration:**
   - Update the Ingress resource to align with the older controller's requirements:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: rhino-app-ingress
  namespace: project-rhino
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: rhino-app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: rhino-app
            port:
              number: 80
```

   - Apply the updated Ingress:

```
kubectl apply -f rhino-app-ingress.yaml
```

6. **Test Traffic Restoration:**
   - Verify external access:

```
curl http://rhino-app.example.com
```

   - Monitor Ingress events:

```
kubectl get events -n project-rhino
```

7. **Plan for Future Upgrades:**
   - Test the new version (1.9.0) in a staging environment before retrying the upgrade.
   - Update Helm chart values (if using Helm):

```
helm upgrade ingress-nginx ingress-nginx/ingress-nginx --namespace ingress-nginx --version 4.7.0
```

Outcome

- Traffic to rhino-app is restored with no 502 errors.
- The Ingress Controller is rolled back to a stable version (1.8.0).
- The cluster is prepared for a controlled upgrade in the future.

Scenario 117: Recovering from a Failed Control Plane Upgrade

Scenario

During an upgrade of your Kubernetes cluster from version 1.24 to 1.25, the control plane components on cluster4-master1 fail to start, leaving the cluster partially operational. Worker nodes are still running, but API requests are intermittent. You need to diagnose the issue, roll back the upgrade, and restore full control plane functionality without data loss.

Solution

Context:

```
kubectl config use-context k8s-c4-prod
```

Steps:

1. **Check Control Plane Status:**
   - SSH into the master node:

```
ssh cluster4-master1
```

   - Verify the status of control plane components:

```
kubectl get pods -n kube-system -l component=kube-apiserver,kube-controller-manager,kube-scheduler
```

   - Note any pods in CrashLoopBackOff or Pending state.

2. **Inspect Component Logs:**
   - Check logs for the API server pod:

```
kubectl logs -n kube-system -l component=kube-apiserver
```

- Example error: Incompatible configuration: etcd version mismatch.

3. **Identify Upgrade Issue:**
   - Confirm the current Kubernetes version:

```
kubeadm version
```

   - Check etcd version compatibility:

```
kubectl get pods -n kube-system -l component=etcd -o yaml | grep image
```

   - Note: Upgrading to 1.25 may have introduced an incompatible etcd version.
4. **Roll Back the Control Plane:**
   - Drain the master node to prevent scheduling:

```
kubectl drain cluster4-master1 --ignore-daemonsets
```

   - Revert kubeadm to the previous version (1.24):

```
sudo apt-get install -y kubeadm=1.24.10-00
```

   - Restore control plane components:

```
sudo kubeadm upgrade apply v1.24.10 --force
```

   - Uncordon the node:

```
kubectl uncordon cluster4-master1
```

5. **Verify etcd Compatibility:**
   - Ensure etcd is running the correct version:

```
kubectl get pods -n kube-system -l component=etcd -o yaml | grep image
```

   - Expected: etcd:3.5.6 (compatible with 1.24).
6. **Test Cluster Functionality:**
   - Confirm API server responsiveness:

```
kubectl get nodes
```

   - Check control plane pod health:

```
kubectl get pods -n kube-system
```

   - Verify no errors in events:

```
kubectl get events -n kube-system
```

7. **Plan for Future Upgrades:**
   - Test the 1.25 upgrade in a staging cluster.
   - Update etcd to a compatible version (e.g., 3.5.9) before retrying:

```
kubectl edit deployment etcd -n kube-system
```

Outcome

- The control plane is restored to version 1.24, ensuring full cluster operability.
- API requests are stable, and no data is lost.
- A plan is in place to safely retry the 1.25 upgrade after testing.

Scenario 118: Troubleshooting StorageClass Provisioning Failure

Scenario

A new StatefulSet in the project-eagle namespace fails to provision persistent volumes due to an issue with the fast-ssd StorageClass. Pods are stuck in Pending with errors indicating "failed to provision volume." You need to diagnose the StorageClass issue, fix the provisioning, and ensure the StatefulSet runs successfully.

Solution

Context:

```
kubectl config use-context k8s-c9-dev
```

Steps:

1. **Check StatefulSet Status:**
   - Inspect the StatefulSet:

```
kubectl get statefulset -n project-eagle
```

   - Confirm pods are Pending:

```
kubectl get pods -n project-eagle
```

2. **Investigate Provisioning Errors:**
   - Describe a stuck pod:

```
kubectl describe pod eagle-db-0 -n project-eagle
```

   - Example error: Failed to provision volume with StorageClass "fast-ssd": invalid provisioner.

3. **Verify StorageClass Configuration:**
   - Inspect the fast-ssd StorageClass:

```
kubectl get storageclass fast-ssd -o yaml
```

   - Note: The provisioner field may be incorrect (e.g., csi.storage.k8s.io instead of csi-driver-name).

4. **Fix the StorageClass:**

- Update the StorageClass with the correct provisioner:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: disk.csi.cloud-provider.com
parameters:
  type: ssd
reclaimPolicy: Retain
volumeBindingMode: WaitForFirstConsumer
```

- Apply the updated StorageClass:

```
kubectl apply -f fast-ssd-storageclass.yaml
```

5. **Validate CSI Driver:**
   - Check if the CSI driver is running:

```
kubectl get pods -n kube-system -l app=csi-driver
```

   - If missing, install the correct CSI driver:

```
kubectl apply -f https://raw.githubusercontent.com/cloud-provider/csi-driver/release-1.0/deployment.yaml
```

6. **Restart the StatefulSet:**
   - Delete stuck pods to trigger reprovisioning:

```
kubectl delete pod -n project-eagle -l app=eagle-db
```

   - Verify volume provisioning:

```
kubectl get pvc -n project-eagle
```

7. **Confirm StatefulSet Health:**
   - Check pod status:

```
kubectl get pods -n project-eagle
```

   - Ensure all pods are Running and volumes are bound:

```
kubectl describe pvc -n project-eagle
```

Outcome

- The fast-ssd StorageClass is corrected, enabling successful volume provisioning.
- The eagle-db StatefulSet is fully operational with bound persistent volumes.

- The cluster is configured to avoid future provisioning issues.

Scenario 119: Debugging Intermittent Application Timeouts

Scenario

Users report intermittent timeouts when accessing a service, frontend-app, in the project-panther namespace. The application logs show occasional connection failures to a backend database service, db-service. You need to diagnose the root cause, resolve the timeouts, and ensure reliable communication between services.

Solution

Context:

```
kubectl config use-context k8s-c10-stage
```

Steps:

1. **Verify Service Accessibility:**
   - Check the frontend-app service:

```
kubectl get svc frontend-app -n project-panther
```

   - Test connectivity to db-service from a temporary pod:

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

wget -O- http://db-service:5432
```

   - Note intermittent failures.

2. **Inspect Application Logs:**
   - View logs for frontend-app:

```
kubectl logs -l app=frontend -n project-panther
```

   - Example error: Connection timeout to db-service:5432.

3. **Check DNS Resolution:**
   - Verify DNS functionality in the cluster:

```
kubectl run dns-test --image=busybox -n project-panther --rm -it -- /bin/sh

nslookup db-service.project-panther.svc.cluster.local
```

   - Confirm CoreDNS is running:

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
```

4. **Analyze Network Traffic:**
   - Deploy a debugging pod with network tools:

```
kubectl run net-debug --image=nicolaka/netshoot -n project-panther --rm -it -- /bin/
```

- Run a tcpdump to capture traffic:

```
tcpdump -i eth0 port 5432
```

- Observe dropped packets or timeouts.

5. **Fix Service Configuration:**

- Inspect the db-service definition:

```
kubectl get svc db-service -n project-panther -o yaml
```

- Update to ensure stable endpoint selection:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db-service
  namespace: project-panther
spec:
  selector:
    app: db
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    clientIP:
      timeoutSeconds: 10800
```

- Apply the updated service:

```
kubectl apply -f db-service.yaml
```

6. **Scale Backend for Reliability:**

- Increase replicas for db-service deployment:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: db-deployment

  namespace: project-panther

spec:

  replicas: 3
```

   - Apply the update:

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

7. **Test and Monitor:**
   - Retest connectivity:

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


wget -O- http://db-service:5432
```

   - Monitor application metrics:

```
kubectl top pod -n project-panther
```

Outcome

- The frontend-app communicates reliably with db-service, eliminating timeouts.
- Service configuration ensures stable connections with session affinity.
- The backend is scaled to handle load, improving overall reliability.

Scenario 120: Enforcing Pod Security Standards Across a Cluster

Scenario

A compliance audit flags that pods in the project-falcon namespace are running with excessive privileges, violating your organization’s Pod Security Standards (PSS). Some pods have privileged: true or lack resource limits, posing a security risk. You need to enforce the baseline Pod Security Standard across the namespace, update non-compliant pods, and ensure future deployments adhere to the policy without disrupting running workloads.

Solution

Context:

```
kubectl config use-context k8s-c11-prod
```

Steps:

1. **Verify Current Pod Security Status:**
   - Check existing pod configurations in the namespace:

```
kubectl get pods -n project-falcon -o yaml | grep -E "privileged|securityContext"
```

   - Identify pods with risky settings (e.g., privileged: true).

2. **Apply Pod Security Standard:**
   - Label the project-falcon namespace to enforce the baseline PSS:

```
kubectl label namespace project-falcon pod-security.kubernetes.io/enforce=baseline --
overwrite
```

   - Verify the label:

```
kubectl get namespace project-falcon -o yaml
```

3. **Audit Non-Compliant Pods:**
   - Run an audit to identify violations:

```
kubectl get pods -n project-falcon --show-labels
```

   - Check events for PSS enforcement failures:

```
kubectl get events -n project-falcon | grep PodSecurity
```

   - Example: Pods like falcon-worker may fail due to privileged settings.

4. **Update Non-Compliant Deployments:**
   - Edit the offending deployment (e.g., falcon-worker):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: falcon-worker
  namespace: project-falcon
spec:
  template:
    spec:
      containers:
      - name: worker
        image: worker:1.3
        securityContext:
```

```
      privileged: false
      allowPrivilegeEscalation: false
      runAsNonRoot: true
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "500m"
        memory: "256Mi"
```

- Apply the updated deployment:

```
kubectl apply -f falcon-worker-deployment.yaml
```

5. **Restart Affected Pods:**
   - Delete non-compliant pods to enforce the new configuration:

```
kubectl delete pod -n project-falcon -l app=falcon-worker
```

   - Verify pods are recreated and compliant:

```
kubectl get pods -n project-falcon -o yaml | grep securityContext
```

6. **Monitor Compliance:**
   - Set up a recurring audit with a policy enforcement tool (e.g., Kyverno or OPA Gatekeeper):

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: enforce-baseline-pss
spec:
  validationFailureAction: Enforce
  rules:
  - name: baseline-security
    match:
```

```
    resources:
      kinds:
      - Pod
  validate:
    podSecurity:
      level: baseline
```

- Apply the policy:

```
kubectl apply -f baseline-pss-policy.yaml
```

7. **Validate Enforcement:**
   - Attempt to deploy a non-compliant pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-privileged
  namespace: project-falcon
spec:
  containers:
  - name: test
    image: nginx
    securityContext:
      privileged: true
```

   - Confirm it’s blocked:

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

   - Check running pods:

```
kubectl get pods -n project-falcon
```

Outcome

- The project-falcon namespace enforces the baseline Pod Security Standard.
- Non-compliant pods are updated to meet security requirements without downtime.

- Future deployments are automatically checked for compliance, reducing security risks.

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.

https://www.linkedin.com/in/prasad-suman-mohan