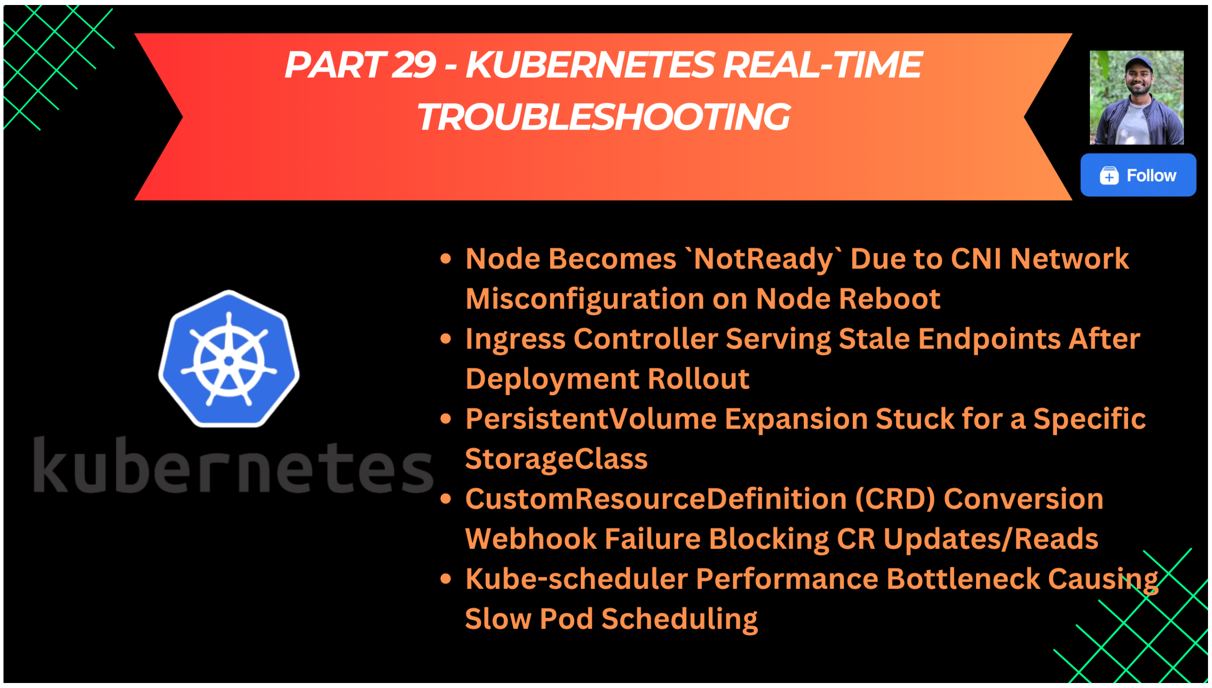
Part 29: Kubernetes Real-Time Troubleshooting

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.



- Node Becomes `NotReady` Due to CNI Network Misconfiguration on Node Reboot
- Ingress Controller Serving Stale Endpoints After Deployment Rollout
- PersistentVolume Expansion Stuck for a Specific StorageClass
- CustomResourceDefinition (CRD) Conversion Webhook Failure Blocking CR Updates/Reads
- Kube-scheduler Performance Bottleneck Causing Slow Pod Scheduling

---

Scenario 141: Node Becomes `NotReady` Due to CNI Network Misconfiguration on Node Reboot

Scenario: A worker node, `worker-node-5`, was rebooted for maintenance. After coming back online, it consistently fails to become `Ready`. `kubectl get nodes` shows its status as `NotReady`. Kubelet logs on `worker-node-5` show repeated errors like "CNI failed to set up pod network for pod default/test-pod-xyz: failed to determine pod CIDR: CNI plugin flannel failed to teardown network for pod default/test-pod-xyz" or "network plugin is not ready: cni config uninitialized".

Solution

Context: kubectl config use-context k8s-c36-clusterinfra

Steps:

1. **Confirm Node Status and Kubelet Errors:**

   Verify `NotReady` status:

   kubectl get node worker-node-5

   SSH into `worker-node-5`.

   Examine Kubelet logs for CNI errors:

   sudo journalctl -u kubelet -f

   # Look for "cni", "network plugin", "pod CIDR" errors. Note the exact CNI plugin mentioned (e.g., flannel, calico, cilium).

2. **Check CNI DaemonSet Pod Status on the Affected Node:**

   The CNI plugin usually runs as a DaemonSet. Identify its pod on `worker-node-5`:

   kubectl get pods -A --field-selector spec.nodeName=worker-node-5 -l <cni-daemonset-label> # e.g., k8s-app=flannel

   If the CNI pod is CrashLoopBackOff, Pending, or has errors, describe it and check its logs:

   kubectl describe pod <cni-pod-on-worker-node-5> -n <cni-namespace>

   kubectl logs <cni-pod-on-worker-node-5> -n <cni-namespace> -p # Check previous logs too.

   Common CNI pod issues: missing ConfigMap/Secret, RBAC issues, failure to communicate with API server, or internal errors.

3. **Inspect CNI Configuration Files on the Node:**

   CNI configuration files are typically in `/etc/cni/net.d/`.

   ls -l /etc/cni/net.d/

   cat /etc/cni/net.d/<cni-config-file>.conf # (e.g., 10-flannel.conf)

   Verify:

The configuration file exists and is valid JSON/conflist.

It points to the correct CNI plugin binary (e.g., in `/opt/cni/bin/`).

Settings like `name`, `type`, `bridge`, `subnetFile` (for flannel), `datastore_type` (for calico with etcd) are correct.

Compare with a CNI config file from a healthy node. Was it corrupted or not properly laid down by the CNI DaemonSet?

4. Check Kubelet Configuration for Pod CIDR:

The Kubelet needs to know the Pod CIDR assigned to the node. This is usually assigned by the controller manager and written to the Node object.

Check Kubelet startup arguments (e.g., `/var/lib/kubelet/kubeadm-flags.env` or systemd unit file):

`--pod-cidr`: If statically set, is it correct and not overlapping?

`--configure-cbr0` or `--cni-bin-dir` and `--cni-conf-dir`.

Describe the Node object from the API server:

```
kubectl describe node worker-node-5
```

Look for `spec.podCIDR` and `spec.podCIDRs`. Is it populated? If not, the controller manager might have issues assigning it, or the Kubelet isn't picking it up.

5. Investigate IPAM (IP Address Management) Issues:

If the CNI plugin uses a specific IPAM method (e.g., `host-local`, flannel's subnet manager, calico-ipam):

Flannel: Check `/run/flannel/subnet.env` on the node. Does it contain the correct `FLANNEL_SUBNET` and `FLANNEL_MTU`? The flannel CNI pod is responsible for creating this.

Calico: Check `calico-node` logs for IPAM errors. If using Calico IPAM with etcd, ensure connectivity.

Host-local: Check for exhaustion of IPs in the configured ranges in `/etc/cni/net.d/`.

6. Examine Node Networking (Routes, Interfaces):

Check the node's routing table: `ip route show`. Are there routes for the pod CIDR and service CIDR?

Check network interfaces: `ip link show`. Is the CNI bridge (e.g., `cni0`, `docker0`) or other virtual interfaces (e.g., `flannel.1`, `vxlan.calico`) created and up?

7. Resolve the Issue (Example: Corrupted Flannel Subnet File):

Assume Kubelet logs show "failed to determine pod CIDR," and the Flannel CNI pod logs on `worker-node-5` show it can't write to or read `/run/flannel/subnet.env` due to a transient disk issue during boot, or the file is missing/corrupted.

Action:

1. Restart CNI Pod: Delete the Flannel CNI pod on `worker-node-5`. The DaemonSet controller will recreate it, which should re-initialize its configuration and attempt to create `/run/flannel/subnet.env` again.

```
kubectl delete pod <flannel-pod-on-worker-node-5> -n kube-system
```

2. Clear Stale CNI State (If Necessary): In some rare cases, stale CNI network interface configurations or IPAM leases might persist. This is highly CNI-specific. For Flannel, ensure `/run/flannel/subnet.env` is removed if the pod is gone, so it can be recreated cleanly. For other CNIs, consult their documentation for troubleshooting node initialization.

3. Reboot (If Simpler and Safe): Sometimes, a clean reboot after ensuring the CNI DaemonSet is healthy can resolve transient state issues.

4. Check for Persisted Incorrect Config: If `/etc/cni/net.d/` was manually modified or corrupted and not managed by the CNI DaemonSet's `initContainer` or lifecycle, ensure it's corrected.

8. Monitor Node Status:

After actions, monitor `worker-node-5`:

```
kubectl get node worker-node-5 -w
```

And Kubelet logs for CNI errors to cease. The node should transition to `Ready`.

Outcome

The root cause of CNI failure on `worker-node-5` after reboot (e.g., corrupted CNI config file, stale IPAM lease, CNI daemonset pod failing to start correctly, Kubelet missing pod CIDR info) is identified.

The CNI configuration is restored (e.g., by restarting the CNI pod, correcting files, ensuring pod CIDR is available to Kubelet).

The worker node successfully initializes its pod network and transitions to the `Ready` state, rejoining the cluster.

Scenario 142: Ingress Controller Serving Stale Endpoints After Deployment Rollout

Scenario: After a rolling update of a Deployment (e.g., `api-gateway` in `production` namespace), the NGINX Ingress controller continues to route a small percentage of requests to pods from the old ReplicaSet for about 30-60 seconds after those old pods have started their termination process. This leads to intermittent 502/503 errors for users. The pods have a `readinessProbe` and `terminationGracePeriodSeconds: 60`.

Solution

Context: kubectl config use-context k8s-c34-ingress

Steps:

1. **Reproduce and Observe:**

    Initiate a rolling update for the `api-gateway` Deployment.

    Simultaneously, send a continuous stream of requests to the Ingress endpoint for `api-gateway`.

    Monitor for HTTP 502/503 errors and correlate their timing with the rollout process.

2. **Verify Service, Endpoints, and Pod Lifecycle:**

    Watch the Service's Endpoints object during the rollout:

    ```
    kubectl get endpoints api-gateway-svc -n production -w
    ```

    Observe when IPs of terminating pods are removed.

    Describe a terminating pod from the old ReplicaSet:

    ```
    kubectl describe pod <old-api-gateway-pod-name> -n production
    ```

    Note the `DeletingTimestamp` and events related to readiness probe failures and SIGTERM.

    Confirm `readinessProbe` is correctly failing on terminating pods, making them `NotReady`.

3. **Inspect NGINX Ingress Controller Logs:**

    Find the NGINX Ingress controller pod(s).

Enable verbose logging if necessary (e.g., via ConfigMap, setting `log-level: "debug"` or higher for NGINX).

Tail the logs during the rollout:

```
kubectl logs -n <ingress-nginx-namespace> <nginx-ingress-pod-name> -f
```

Look for log entries related to:

Endpoint updates for the `api-gateway-svc`.

The specific upstream IPs it's routing to during the error period.

Any errors related to connecting to those upstreams.

4. **Check NGINX Ingress Controller Configuration:**

Inspect the NGINX Ingress controller's ConfigMap for relevant settings:

```
kubectl get configmap nginx-configuration -n <ingress-nginx-namespace> -o yaml # Name might vary
```

Look for settings related to:

`upstream-keepalive-timeout`

`proxy-connect-timeout`, `proxy-read-timeout`

Service synchronization interval (though NGINX Ingress typically uses watches, not polling).

Consider if any custom annotations on the Ingress resource for `api-gateway` might influence upstream handling (e.g., session affinity, custom load balancing).

5. **Analyze the Timing Mismatch:**

The core issue is often a race condition:

1. Pod receives SIGTERM. `readinessProbe` starts failing.
2. Pod enters `terminating` state but is still in the Service Endpoints list as `Ready=false`.
3. NGINX Ingress controller might take some time to process the Endpoint update where the pod is marked `NotReady` or fully removed.
4. If NGINX's internal upstream list is not updated quickly enough, or if it has keep-alive connections to the old pod, it might still attempt to route requests there.
5. `terminationGracePeriodSeconds` allows the pod to finish in-flight requests, but if new requests are still sent to it, they will fail.

6. Implement Solutions:

A) `preStop` Hook with Delay: Add a `preStop` lifecycle hook to the application container in the Deployment. This hook executes before SIGTERM is sent to the main container process.

```
lifecycle:
  preStop:
    exec:
      command: ["/bin/sh", "-c", "sleep 30"] # Delay for N seconds
```

The `sleep` allows time for endpoint controllers and the Ingress controller to remove the pod from their load balancing pools before the application stops accepting new connections or the readiness probe definitively fails. The sleep duration should be tuned based on observed propagation delays.

B) Tune NGINX Ingress Controller Settings (with caution):

Reduce `upstream-keepalive-timeout` if very long, but this can impact performance.

NGINX Ingress has an annotation `nginx.ingress.kubernetes.io/service-upstream: "true"` which makes NGINX use the Service ClusterIP as the upstream. This can sometimes help by relying more on `kube-proxy`'s endpoint handling, but it adds an extra hop and has other implications. Test thoroughly.

C) Ensure Readiness Probe is Effective: The `readinessProbe` should quickly fail once the pod intends to stop serving traffic. It should not rely on SIGTERM alone.

7. Test and Verify:

After applying a solution (e.g., `preStop` hook), re-run the rollout test (Step 1).

Monitor for 502/503 errors. The goal is zero errors due to routing to terminating pods.

Outcome

The race condition between pod termination, endpoint updates, and Ingress controller endpoint synchronization is understood.

A solution, typically a `preStop` hook with a carefully tuned delay, is implemented.

The Ingress controller no longer routes traffic to terminating pods during rollouts, eliminating the intermittent 502/503 errors and ensuring smoother deployments.

Scenario 143: PersistentVolume Expansion Stuck for a Specific StorageClass

Scenario: You are attempting to expand a PersistentVolumeClaim (PVC) named `cache-db-pvc` in the `services` namespace. The StorageClass `fast-expandable-ssd` has `allowVolumeExpansion: true`. After editing the PVC to request a larger size, the PVC's `status.conditions` shows `FileSystemResizePending: true`, but it never resolves. The pod using the PVC (`cache-db-0`) continues to run but with the old volume size. Attempts to expand other PVCs using different StorageClasses work fine.

Solution

Context: kubectl config use-context k8s-c33-storageops

Steps:

1. **Verify PVC and PV Status:**

   Get the PVC and note its status, conditions, and requested/actual capacity:

   ```
   kubectl get pvc cache-db-pvc -n services -o yaml
   ```

   Look for `status.capacity` (old size) vs. `spec.resources.requests.storage` (new size).

   Confirm `status.conditions` includes `FileSystemResizePending` with status `True`.

   Get the corresponding PersistentVolume (PV):

   ```
   kubectl get pv <pv-name-for-cache-db-pvc> -o yaml
   ```

   Check its `spec.capacity.storage` (should reflect new size after CSI driver controller part is done) and `status.phase`.

2. **Inspect CSI External-Resizer Logs:**

   Find the pod running the `csi-resizer` sidecar for your `fast-expandable-ssd` StorageClass's CSI driver (often part of the CSI controller deployment).

   ```
   kubectl get pods -n <csi-driver-ns> -l app=<csi-controller-label>
   ```

Examine logs of the `csi-resizer` container:

```
kubectl logs -n <csi-driver-ns> <csi-controller-pod-name> -c csi-resizer -f
```

Look for errors related to `cache-db-pvc` or its PV. It might indicate issues calling `ControllerExpandVolume` on the CSI driver or that the driver reported success but the node-stage expansion is pending.

3. **Inspect CSI Driver Controller Plugin Logs:**

In the same CSI controller pod, check logs of the main driver plugin:

```
kubectl logs -n <csi-driver-ns> <csi-controller-pod-name> -c <csi-driver-container-name> -f
```

These logs should show if the `ControllerExpandVolume` gRPC call was received and if it succeeded or failed at the storage backend level.

4. **Inspect Kubelet Logs on the Node Hosting the Pod:**

The `FileSystemResizePending` condition means the volume was expanded at the storage backend level, but the filesystem on the node needs to be resized. Kubelet is responsible for triggering this if the CSI driver supports node-stage expansion.

SSH into the node where `cache-db-0` is running.

Check Kubelet logs:

```
sudo journalctl -u kubelet -f | grep -i 'resize\|expand'
```

Look for messages related to resizing the filesystem for the specific PV.

5. **Inspect CSI Driver Node Plugin Logs:**

On the node hosting `cache-db-0`, find the CSI driver's node plugin pod.

Examine its logs:

```
kubectl logs -n <csi-driver-ns> <csi-node-pod-on-correct-node> -c <csi-driver-node-container-name> -f
```

Look for errors related to `NodeStageVolume` or `NodeExpandVolume` (if supported) for the PV in question.

6. **Verify StorageClass and CSI Driver Capabilities:**

Describe the `StorageClass fast-expandable-ssd`:

```
kubectl get sc fast-expandable-ssd -o yaml
```

Confirm `allowVolumeExpansion: true`.

Check the `CSIDriver` object for `com.example.csi-driver-name` (the provisioner for the SC):

```
kubectl get csidriver com.example.csi-driver-name -o yaml
```

Verify `spec.podInfoOnMount: true` (often needed for node operations) and `spec.volumeLifecycleModes` includes `Persistent` and potentially `Ephemeral`. Check if `spec.fsGroupPolicy` is relevant.

The crucial part is whether the CSI driver implements `NodeExpandVolume` capability. If not, Kubelet might not automatically resize, or online expansion might not be fully supported by this driver.

7. Attempt Manual Filesystem Resize (If Safe and Driver Lacks Auto Node Resize):

Caution: This is risky and depends on the filesystem type and whether the application can tolerate it.

If the CSI driver expanded the block device but doesn't handle filesystem resize on the node, and Kubelet isn't doing it:

1. Exec into the `cache-db-0` pod: `kubectl exec -it cache-db-0 -n services -- /bin/bash`
2. Identify the device and mount point (`df -h`, `lsblk`).
3. Use appropriate filesystem resize tool (e.g., `resize2fs /dev/sdXN` for ext4, `xfs_growfs /mount/point` for XFS).

This will not clear the `FileSystemResizePending` condition, as Kubernetes expects the driver/Kubelet to do it.

8. Resolve the Issue (Example: CSI Driver Node Plugin Bug or Missing Capability):

Assume logs indicate the CSI driver controller successfully expanded the volume, but the node plugin or Kubelet is not performing the filesystem resize. The `CSIDriver` object might show that `NodeExpandVolume` is not implemented or advertised by this specific driver version.

Action:

1. Check Driver Documentation: Verify if the current CSI driver version for `fast-expandable-ssd` supports online filesystem resizing (node-stage expansion).

2. Upgrade CSI Driver: If a newer version supports it, plan an upgrade of the CSI driver.

3. Offline Resize (Workaround): If online resize is not supported by the driver, the typical workaround is:

Scale down the application using the PVC (`replicas: 0`).

Wait for the pod to terminate.

The filesystem resize might then occur when the volume is unmounted/remounted, or Kubelet might attempt it while the volume is not in use.

If still stuck, detach the volume (if possible via storage backend) and manually resize on a utility VM, then reattach.

Scale up the application.

4. Report Bug: If it's a bug in a supposedly supported feature, report it to the CSI driver vendor.

Outcome

The reason for the stuck volume expansion (e.g., CSI driver lacks node-stage expansion capability, bug in CSI node plugin, Kubelet issue, or incompatibility) is identified.

A workaround (like offline resize) is implemented, or the CSI driver is upgraded/fixed.

PVCs can be successfully expanded, and the `FileSystemResizePending` condition is cleared appropriately, with the filesystem reflecting the new size.

Scenario 144: CustomResourceDefinition (CRD) Conversion Webhook Failure Blocking CR Updates/Reads

Scenario: You have a CustomResourceDefinition (CRD) for `MyPlatformConfig` which previously only had a `v1alpha1` version. You've introduced a `v1` version and implemented a conversion webhook to handle transformations between `v1alpha1` and `v1`. After deploying the new CRD and the webhook, attempts to `kubectl get myplatformconfigs.stable.example.com -n my-namespace` (which should serve `v1` as the storage version) or update existing `v1alpha1` CRs start failing with errors like: `Internal error occurred: failed calling webhook "crd-conversion.stable.example.com": Post "https://myplatformconfig-converter.webhook-ns.svc:443/convert?timeout=30s": context deadline exceeded` or a specific conversion logic error.

Solution

Context: kubectl config use-context k8s-c35-crddev

Steps:

1. **Identify Failing Operation and Error:**

   Perform the failing `kubectl get` or `kubectl apply` operation and capture the exact error.

   kubectl get myplatformconfigs.stable.example.com my-config -n my-namespace -v=7 # Increased verbosity

   # Error: Internal error occurred: failed calling webhook "crd-conversion.stable.example.com": ...

   Note the webhook name (`crd-conversion.stable.example.com`) and the specific error (timeout, TLS issue, specific denial message from webhook).

2. **Inspect the CRD Definition:**

   Get the full CRD definition:

   kubectl get crd myplatformconfigs.stable.example.com -o yaml

   Verify:

   `spec.conversion.strategy` is `Webhook`.

   `spec.conversion.webhook.clientConfig.service` correctly points to your webhook's Service (`myplatformconfig-converter.webhook-ns.svc`).

   `spec.conversion.webhook.clientConfig.caBundle` is correctly populated if using custom CAs.

   `spec.conversion.webhook.conversionReviewVersions` includes `v1` (and `v1alpha1` if your webhook logic expects it, though typically `v1` is for AdmissionReview/ConversionReview).

3. **Examine Conversion Webhook Service and Pods:**

   Verify the webhook service (`myplatformconfig-converter` in `webhook-ns`) exists, has correct port (443/HTTPS), and has active Endpoints.

```
kubectl get svc myplatformconfig-converter -n webhook-ns -o wide
kubectl get endpoints myplatformconfig-converter -n webhook-ns
```

Check the status and logs of the pod(s) backing the webhook:

```
kubectl get pods -n webhook-ns -l app=myplatformconfig-converter
kubectl logs -n webhook-ns <webhook-pod-name> -f
```

The webhook logs are critical. They should show incoming `ConversionReview` requests and the `ConversionResponse`. Look for:

Errors in deserializing the `ConversionReview` request.

Logic errors during the conversion of objects between `v1alpha1` and `v1`.

Errors serializing the `ConversionResponse`.

Panics or crashes.

4. Test Webhook Connectivity and TLS:

Ensure the API server can reach the webhook service endpoint. Check NetworkPolicies.

From a pod within the cluster (or using `kubectl debug node/...`), try to `curl` the webhook's `/convert` endpoint (requires setting up auth/certs if testing manually, or just testing connectivity to the port).

```
# From a debug pod in cluster
curl -k https://myplatformconfig-converter.webhook-ns.svc:443/convert # -k to ignore cert validation for a quick test
# Expect a 400 Bad Request if no payload, but not connection refused/timeout
```

If TLS errors are suspected (from API server logs or webhook logs), verify the webhook server certificate is valid, matches the service name, and is trusted by the CA bundle configured in the CRD.

5. Simulate a ConversionReview Request (Advanced):

Manually craft a `ConversionReview` JSON object.

`request.uid` (a unique ID)

`request.desiredAPIVersion` (e.g., `stable.example.com/v1`)

`request.objects` (an array of raw `runtime.RawExtension` objects to be converted, e.g., a `v1alpha1` CR)

Send this JSON to your webhook's `/convert` endpoint using `curl` from a pod that can reach it, ensuring correct headers (`Content-Type: application/json`).

Examine the `ConversionReview` response. It should contain `response.uid` (matching request), `response.convertedObjects` (the objects in the desired API version), and `response.result.status` ("Success" or "Failure" with a message).

6. Address the Root Cause (Example: Bug in Conversion Logic):

Assume webhook logs show a panic or a specific error message like "field 'spec.oldField' not found in v1alpha1 object during conversion to v1" when processing certain CRs.

Action:

1. Debug Webhook Code: Identify the bug in the conversion function (e.g., mishandling of optional fields, incorrect type assertions, failure to correctly map a field from `v1alpha1` to `v1` or vice-versa).
2. Deploy Fixed Webhook: Build and deploy the corrected webhook image.
3. Consider CR Data: If some existing CRs have data that the buggy (or even fixed) webhook cannot handle, you might need a data migration strategy or to manually correct those CRs (e.g., by editing them via `v1alpha1` if reads are still possible at that version, or directly in etcd as a last resort).

7. Retry CRD Operations:

After deploying the fixed webhook, retry `kubectl get` and `kubectl apply` operations.

```
kubectl get myplatformconfigs.stable.example.com my-config -n my-namespace
```

Monitor webhook logs and API server logs.

Outcome

The issue with the CRD conversion webhook (e.g., connectivity problem, TLS misconfiguration, bug in conversion logic, malformed `ConversionReview` handling) is identified.

The webhook deployment, its code, or CRD configuration is corrected.

Kubernetes can now successfully convert CRs between `v1alpha1` and `v1` (or other configured versions), allowing users to read and update CRs using their preferred/storage API versions.

Scenario 145: Kube-scheduler Performance Bottleneck Causing Slow Pod Scheduling

Scenario: Newly created pods across various namespaces are taking an unusually long time (several minutes) to transition from `Pending` to a scheduled state, even though `kubectl get nodes` shows ample allocatable resources (CPU, memory). There are no obvious affinity/anti-affinity rules or taints/tolerations that would explain the delay for all these pods. The cluster has a large number of nodes (~500) and active pods.

Solution:

Context: kubectl config use-context k8s-c32-large

Steps:

1. **Confirm Widespread Scheduling Delay:**

   Deploy a simple test pod without specific constraints and observe its scheduling time:

   ```
   time kubectl run nginx-test --image=nginx --restart=Never
   # Monitor 'kubectl get pod nginx-test -w' for the 'PodScheduled' condition
   ```

   Check events for a pending pod:

   ```
   kubectl describe pod <a-pending-pod-name> -n <namespace>
   # Look for 'FailedScheduling' events, but in this case, expect a long delay before 'Scheduled'
   ```

2. **Inspect Kube-scheduler Logs:**

   Find the `kube-scheduler` pod(s) (usually in `kube-system`):

   ```
   kubectl get pods -n kube-system -l component=kube-scheduler
   ```

   Examine the logs, potentially increasing verbosity (e.g., by editing the static pod manifest `/etc/kubernetes/manifests/kube-scheduler.yaml` to add `--v=4` and restarting the scheduler).

   ```
   kubectl logs -n kube-system <kube-scheduler-pod-name> -f
   ```

   Look for:

   Repeated lines evaluating a large number of nodes for each pod.

   Messages indicating long processing times for predicates or priorities.

   Any errors or warnings related to plugin execution.

3. Examine Kube-scheduler Metrics:

If Prometheus is scraping `kube-scheduler` metrics, analyze:

`scheduler_scheduling_algorithm_duration_seconds_bucket` (or `scheduler_framework_extension_point_duration_seconds_bucket` for newer versions): High latencies here indicate slowdowns in the core scheduling loop (filter, score, bind).

`scheduler_e2e_scheduling_duration_seconds_bucket`: Overall pod scheduling latency from scheduler's perspective.

`scheduler_pod_scheduling_attempts_total`: High number of attempts per pod.

`scheduler_schedule_goroutines`: Number of active goroutines for scheduling.

Without Prometheus, you can `curl` the `/metrics` endpoint on the scheduler pod (e.g., port-forward).

4. Review Scheduler Configuration:

Check the scheduler configuration (often passed via `--config` in its manifest or a ConfigMap).

```
# Check kube-scheduler manifest for --config argument

# kubectl get configmap kube-scheduler-config -n kube-system -o yaml (if applicable)
```

Look for:

Custom or a large number of enabled scheduler plugins (predicates/priorities in older versions).

`percentageOfNodesToScore`: If set very low, it might not find optimal nodes quickly in a large cluster, but setting it too high can increase scoring overhead. Default is usually fine.

Complex `InterPodAffinity` or `TopologySpreadConstraints` policies, if widely used, can significantly increase scheduling complexity.

5. Consider Cluster Scale and Resource Complexity:

In very large clusters, the sheer number of nodes and pods can strain the scheduler if its algorithms are not efficient enough or if many complex scheduling constraints are in play.

A high rate of pod churn (creations/deletions) also adds load.

6. Profile the Scheduler (Advanced):

If metrics point to a specific bottleneck, consider enabling profiling on the scheduler (`--profiling=true` in manifest) and capturing CPU/memory profiles via its HTTP pprof endpoints (e.g., `/debug/pprof/profile`). This requires expertise to analyze.

7. Address the Issue (Example: Inefficient Custom Plugin or Widespread Complex Affinity):

Assume scheduler logs and metrics (high `scheduler_framework_extension_point_duration_seconds` for a specific filter or score plugin) point to an inefficient custom scheduling plugin or very widespread use of computationally expensive `podAntiAffinity` rules across many nodes.

Action:

1. Optimize/Disable Custom Plugin: If a custom plugin is the culprit, optimize its logic or temporarily disable it via scheduler configuration to confirm impact.

2. Review Affinity Rules: Audit the usage of complex `podAntiAffinity` or `topologySpreadConstraints`. Can they be simplified, reduced in scope, or replaced with less expensive alternatives for some workloads?

3. Scale Scheduler Resources: Ensure the `kube-scheduler` pod itself has adequate CPU/memory requests and limits.

4. Consider Scheduler Profiles: For different types of workloads, consider defining multiple scheduler profiles with different sets of plugins to optimize for specific needs, rather than one-size-fits-all.

8. Monitor Scheduling Performance:

After applying changes, re-run test pod deployments (Step 1) and monitor scheduler metrics (Step 3) to confirm scheduling latencies have improved.

Outcome

The bottleneck in the `kube-scheduler` (e.g., inefficient plugin, overwhelming number of complex constraints, insufficient scheduler resources for cluster scale) is identified.

Scheduler configuration, workload scheduling policies, or custom plugins are optimized.

Pod scheduling times return to acceptable levels, improving cluster responsiveness and application deployment speed.

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