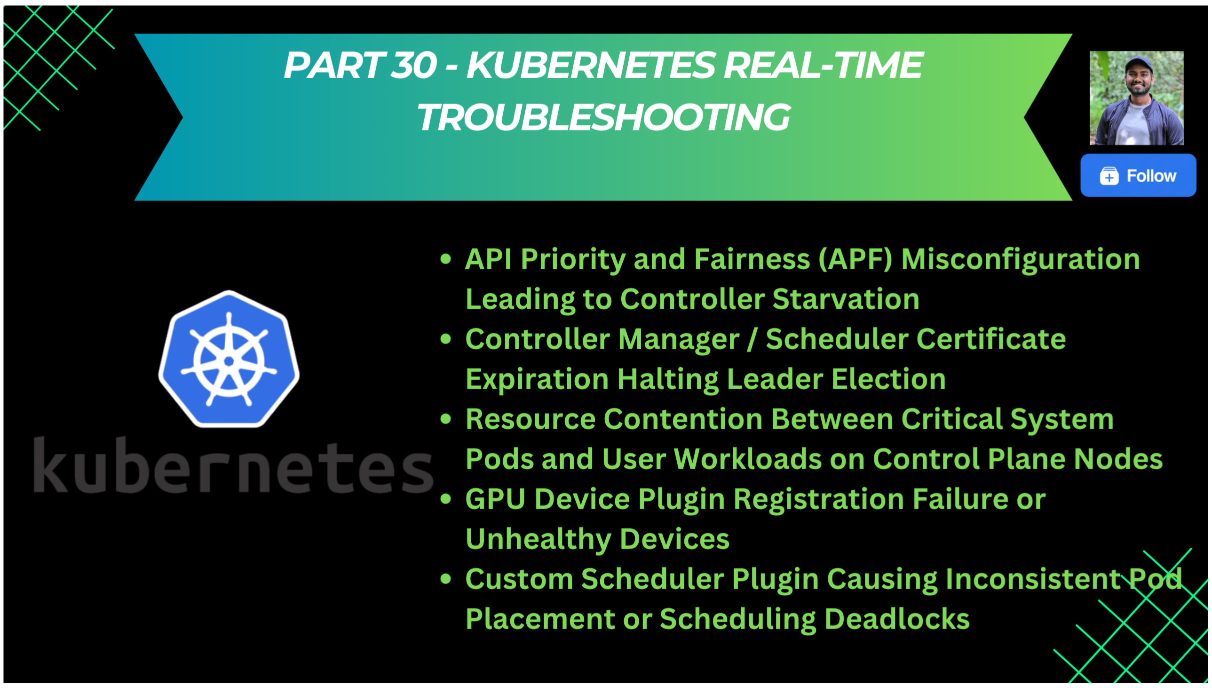
Part 30: Kubernetes Real-Time Troubleshooting

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 30 - KUBERNETES REAL-TIME TROUBLESHOOTING

- API Priority and Fairness (APF) Misconfiguration Leading to Controller Starvation
- Controller Manager / Scheduler Certificate Expiration Halting Leader Election
- Resource Contention Between Critical System Pods and User Workloads on Control Plane Nodes
- GPU Device Plugin Registration Failure or Unhealthy Devices
- Custom Scheduler Plugin Causing Inconsistent Pod Placement or Scheduling Deadlocks

---

<u>Scenario 146: API Priority and Fairness (APF) Misconfiguration Leading to Controller Starvation</u>

Scenario: A custom controller (`resource-optimizer-controller`) responsible for managing `OptimizedResource` CRs is observed to be extremely slow in its reconciliations. Its logs show it's taking a long time to get or list `OptimizedResource` CRs, even though the number of CRs is not excessively large. Other cluster operations seem mostly fine, but this specific controller is lagging. The cluster has API Priority and Fairness (APF) enabled.

Solution:

Context:

kubectl config use-context k8s-c37-apf

Steps:

1. **Confirm Controller Slowness and API Latency:**

Examine `resource-optimizer-controller` logs for timestamps indicating slow API calls (GETs, LISTs, WATCHes for `OptimizedResource` CRs).

If the controller exposes metrics, check its workqueue depth and reconciliation loop latency.

2. **Inspect API Server Metrics for APF:**

Examine API server metrics (e.g., via Prometheus) related to APF:

`apiserver_flowcontrol_request_wait_duration_seconds_bucket`: High wait times for specific FlowSchemas or PriorityLevelConfigurations.

`apiserver_flowcontrol_rejected_requests_total`: Are requests from the controller's UserAgent or ServiceAccount being rejected?

`apiserver_flowcontrol_current_executing_requests`: Number of requests currently executing per PriorityLevel.

`apiserver_flowcontrol_priority_level_seat_utilization`: Utilization of concurrency seats for each priority level.

3. **Identify FlowSchema and PriorityLevelConfiguration for the Controller:**

The `resource-optimizer-controller`'s requests will be matched by a `FlowSchema` and assigned to a `PriorityLevelConfiguration`.

List `FlowSchemas`:

```
kubectl get flowschemas -o wide
```

Look for a `FlowSchema` whose `matchingPrecedence` and `rules` (matching on user, group, or service account of the controller, or the `OptimizedResource` API group/resource) would classify requests from `resource-optimizer-controller`.

Describe the matched `FlowSchema` to find its `priorityLevelConfiguration.name`.

```
kubectl describe flowschema <matched-flowschema-name>
```

Describe the identified `PriorityLevelConfiguration`:

```
kubectl describe prioritylevelconfiguration <plc-name>
```

Note its `spec.limited.assuredConcurrencyShares` (ACS) and `spec.limited.limitResponse.queuing.queues` and `queueLengthLimit`.

4. Analyze APF Configuration for Bottlenecks:

Low ACS: If the `PriorityLevelConfiguration` used by the controller has very low `assuredConcurrencyShares` (e.g., 1 or 5) and other, possibly less important, requests are also mapped to this same priority level, the controller's requests might be frequently queued or time out waiting for a "seat."

Incorrect FlowSchema Matching: The controller's requests might be unintentionally falling into a lower-priority FlowSchema (e.g., a catch-all) due to misconfigured rules or incorrect matching precedence.

Shared Priority Level Overload: Even if ACS seems reasonable, if too many distinct, high-volume clients are mapped to the same `PriorityLevelConfiguration`, they can exhaust its concurrency and queue capacity.

5. Check API Server Audit Logs:

If audit logging is enabled, filter logs for requests from the `resource-optimizer-controller`'s ServiceAccount or UserAgent.

Look at the `annotations` in the audit events, specifically `apf.kubernetes.io/flowschema` and `apf.kubernetes.io/prioritylevel`. This confirms how APF classified the request.

Note the `responseStatus.code`. Are there many 429s (Too Many Requests)?

6. Resolve APF Misconfiguration:

A) Create a Dedicated PriorityLevelConfiguration and FlowSchema: For critical controllers, it's often best to define a dedicated, higher-priority `PriorityLevelConfiguration` with sufficient `assuredConcurrencyShares` and a specific `FlowSchema` that precisely matches the controller's requests.

```
# Example PriorityLevelConfiguration
apiVersion: flowcontrol.apiserver.k8s.io/v1beta3 # or v1beta2, v1
kind: PriorityLevelConfiguration
metadata:
  name: resource-optimizer-controller
spec:
  type: Limited
  limited:
```

```
    assuredConcurrencyShares: 20 # Give it more shares
    limitResponse:
      queuing:
        queues: 64
        queueLengthLimit: 200
        handSize:

# Example FlowSchema
apiVersion: flowcontrol.apiserver.k8s.io/v1beta3
kind: FlowSchema
metadata:
  name: resource-optimizer-controller
spec:
  matchingPrecedence: 200 # Higher than general catch-alls
  priorityLevelConfiguration:
    name: resource-optimizer-controller # Point to dedicated PLC
  rules:
  - resourceRules:
    - apiGroups: ["stable.example.com"] # API group of OptimizedResource
      namespaces: [""]
      resources: ["optimizedresources"]
      verbs: [""]
    subjects:
    - kind: ServiceAccount
      serviceAccount:
        name: resource-optimizer-controller-sa
        namespace: kube-system # Or wherever the controller SA is
```

B) Adjust Existing PLC/FS: If a dedicated one is overkill, carefully adjust the `assuredConcurrencyShares` of the existing PLC or the `matchingPrecedence` and rules of `FlowSchemas` to ensure the controller gets fair treatment. This requires a holistic view of APF usage.

C) Optimize Controller API Usage: Ensure the controller uses informers with shared caches, efficient LIST/WATCH selectors, and avoids making unnecessary API calls.

7. **Monitor Controller and APF Metrics:**

After applying changes, monitor the controller's performance and APF metrics (Step 2) to confirm that request wait times for its PLC have decreased and rejections are minimal.

Outcome:

The API Priority and Fairness misconfiguration starving the `resource-optimizer-controller` (e.g., too few concurrency shares, incorrect FlowSchema matching) is identified.

APF configuration is adjusted (e.g., by creating a dedicated PriorityLevelConfiguration and FlowSchema for the controller or tuning existing ones).

The `resource-optimizer-controller` can make API calls efficiently, its reconciliation loops speed up, and it performs its duties reliably.

Scenario 147: Controller Manager / Scheduler Certificate Expiration Halting Leader Election

Scenario: After several months of stable operation, new pods are stuck in `Pending`, and Deployments are not rolling out updates. `kubectl get componentstatuses` shows `controller-manager` and `scheduler` as `Unhealthy` with messages like "failed to get leader: Get "https://<kube-controller-manager-svc-or-ip>:10257/healthz": x509: certificate has expired or is not yet valid". The API server itself is accessible and its certificate is fine.

Solution

Context: kubectl config use-context k8s-c38-controlplane

Steps:

1. **Confirm Component Status and Error Messages:**

Check component statuses:

```
kubectl get componentstatuses (cs)
```

Note the exact error messages for `controller-manager` and `scheduler`.

2. Inspect Controller Manager and Scheduler Logs:

Find the `kube-controller-manager` and `kube-scheduler` pods (usually in `kube-system`).

Examine their logs:

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

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

Look for repeated errors related to:

Leader election failures.

TLS handshake errors when communicating with the API server or other components for leader election lease objects.

"x509: certificate has expired" messages, potentially referencing specific certificate files or internal certs used for secure port communication.

3. Identify Which Certificates Have Expired:

The error might be due to:

Kubeconfig certs: The certificates within the kubeconfig files these components use to talk to the API server (`/etc/kubernetes/controller-manager.conf`, `/etc/kubernetes/scheduler.conf`).

Serving certs: If these components expose secure health/metrics ports (e.g., 10257 for controller-manager, 10259 for scheduler), the TLS certificates for these ports might have expired.

On a control plane node, check the expiration of these certificate files:

```
# For kubeconfig client certs (paths might vary slightly)

sudo openssl x509 -in /etc/kubernetes/controller-manager.crt -noout -dates # If separate cert/key in kubeconfig

sudo openssl x509 -in /etc/kubernetes/scheduler.crt -noout -dates

# Or, if kubeconfig has embedded certs, extract and check:

kubectl config view --kubeconfig=/etc/kubernetes/controller-manager.conf --raw -o jsonpath='{.users[0].user.client-certificate-data}' | base64 -d | openssl x509 -noout -dates

kubectl config view --kubeconfig=/etc/kubernetes/scheduler.conf --raw -o jsonpath='{.users[0].user.client-certificate-data}' | base64 -d | openssl x509 -noout -dates

# For serving certs (if applicable, paths are usually in /etc/kubernetes/pki or passed as flags)
```

```
# e.g., /etc/kubernetes/pki/controller-manager-server.crt (path depends on setup)
# Check component manifests for --tls-cert-file and --tls-private-key-file flags
```

4. **Renew the Expired Certificates:**

If `kubeadm` managed: `kubeadm` can renew most control plane certificates.

```
# First, back up /etc/kubernetes/
sudo cp -r /etc/kubernetes /etc/kubernetes.backup.$(date +%F-%T)
# Check expiration status with kubeadm
sudo kubeadm certs check-expiration
# Renew specific certs (or 'all')
sudo kubeadm certs renew controller-manager.conf # For the kubeconfig
sudo kubeadm certs renew scheduler.conf         # For the kubeconfig
# If serving certs are also managed by kubeadm and expired, renew them too (e.g., 'controller-manager-server', 'scheduler-server' if such certs exist)
# Or, more broadly:
# sudo kubeadm certs renew all
```

If manually managed / non-kubeadm: You'll need to re-issue these certificates using your PKI tools (e.g., `openssl`, `cfssl`) based on your CA. Update the respective `.crt`, `.key`, and `.conf` files.

5. **Restart Control Plane Components:**

After renewing certificates, the components need to be restarted to pick up the new ones. Since they are usually run as static pods:

1. Move their manifests out of `/etc/kubernetes/manifests/` temporarily. Kubelet will stop the pods.

```
sudo mv /etc/kubernetes/manifests/kube-controller-manager.yaml /tmp/
sudo mv /etc/kubernetes/manifests/kube-scheduler.yaml /tmp/
```

2. Wait for pods to stop (`kubectl get pods -n kube-system`).
3. Move the manifests back. Kubelet will restart them.

```
sudo mv /tmp/kube-controller-manager.yaml /etc/kubernetes/manifests/
sudo mv /tmp/kube-scheduler.yaml /etc/kubernetes/manifests/
```

Alternatively, for some components or setups, a simple `docker restart <container_id>` (if using Docker runtime for static pods) or `sudo systemctl restart kubelet` might trigger a reload, but moving manifests is often more reliable for static pods.

6. **Verify Component Health and Functionality:**

   Check component statuses again:

   ```
   kubectl get cs
   # They should become Healthy
   ```

   Monitor their logs for successful leader election and normal operation.

   Verify new pods are now being scheduled and Deployments are progressing.

Outcome:

Expired certificates for `kube-controller-manager` and/or `kube-scheduler` (either their client certs for API server communication or their serving certs for health checks) are identified as the cause of leader election failure.

The affected certificates are renewed using `kubeadm` or manual PKI processes.

The `kube-controller-manager` and `kube-scheduler` components are restarted and successfully perform leader election.

Pod scheduling and controller operations resume, restoring normal cluster functionality.

Scenario 148: Resource Contention Between Critical System Pods and User Workloads on Control Plane Nodes

Scenario: Control plane nodes, which are not tainted to prevent user workloads by default in some setups (or taints were removed), are experiencing performance issues. `etcd` members on these nodes show high disk commit latencies, and the API server is occasionally sluggish. Investigation reveals that some user-deployed DaemonSets or high-resource pods have landed on these control plane nodes and are consuming significant CPU, memory, or disk I/O, starving core components.

Solution:

Context:

```
kubectl config use-context k8s-c39-mixedcp
```

Steps:

1. **Identify Resource Usage on Control Plane Nodes:**

   List nodes with control plane labels (e.g., `node-role.kubernetes.io/control-plane` or `node-role.kubernetes.io/master`):

   ```
   kubectl get nodes -l 'node-role.kubernetes.io/control-plane'
    # Or check taints to see if they are schedulable by user pods:
    kubectl describe node <control-plane-node-name> | grep Taints
   ```

   For each control plane node, check resource utilization:

   ```
    kubectl top node <control-plane-node-name>
    # SSH to the node and use 'top', 'htop', 'iostat', 'vmstat'
   ```

2. **Identify Non-System Pods on Control Plane Nodes:**

   List all pods running on a specific control plane node, excluding those in `kube-system`:

   ```
    kubectl get pods -A -o wide --field-selector spec.nodeName=<control-plane-node-name> | grep -v kube-system
   ```

   Identify any unexpected user pods or resource-intensive DaemonSets.

3. **Assess Resource Consumption by Offending Pods:**

   For each suspicious non-system pod found on a control plane node, check its resource usage:

   ```
    kubectl top pod <pod-name> -n <namespace> --containers
   ```

   If disk I/O is suspected, try to correlate pod activity with `iostat` on the node.

4. **Review Node Taints and Tolerations:**

   Ensure control plane nodes have appropriate taints to prevent general user workloads from scheduling on them. The standard taint is `node-role.kubernetes.io/control-plane:NoSchedule` or `node-role.kubernetes.io/master:NoSchedule`.

   If these taints are missing or were removed, user pods without specific tolerations can be scheduled there.

Check if the offending user pods/DaemonSets have tolerations for control plane taints.

5. Examine System Pod Resource Requests/Limits:

Check the manifests for `etcd`, `kube-apiserver`, `kube-controller-manager`, `kube-scheduler` (usually in `/etc/kubernetes/manifests/`):

Do they have appropriate CPU/memory requests and limits set?

If requests are too low, they might not be guaranteed enough resources when contention occurs.

6. Implement Isolation and Resource Protection:

A) Taint Control Plane Nodes: If not already done, apply `NoSchedule` (or `NoExecute` for stronger eviction) taints to control plane nodes.

kubectl taint nodes <control-plane-node-name> node-role.kubernetes.io/control-plane=:NoSchedule

# Or node-role.kubernetes.io/master=:NoSchedule

This will prevent new general pods from scheduling. Existing pods will need to be evicted or deleted.

B) Evict/Delete Offending User Pods:

If the offending pods are part of Deployments/StatefulSets, scale them down or add `nodeSelector`/`affinity` rules to move them to worker nodes.

For DaemonSets, use `tolerations` and `affinity` in their spec to ensure they only run on appropriate worker nodes, or if they must run on control plane nodes (e.g., a security agent), ensure they have strict resource limits.

Manually delete problematic pods if they are not managed by a controller or after adjusting their controller.

C) Set/Adjust Resource Requests/Limits for System Pods: Ensure core components have adequate `requests` to guarantee their resource share and reasonable `limits`.

D) Utilize Node Allocatable Resources: Configure Kubelet's `--kube-reserved` and/or `--system-reserved` flags on control plane nodes to reserve resources specifically for Kubernetes system daemons and OS components, protecting them from pod resource pressure.

E) PriorityClasses for System Pods: Assign higher `PriorityClass` names to critical system pods (like `etcd`, `kube-apiserver`) to ensure they are less likely to be preempted by lower-priority user pods if scheduling pressure exists. `system-cluster-critical` and `system-node-critical` are built-in high priorities.

7. **Monitor Control Plane Health and Resource Usage:**

   After implementing changes, monitor:

   `etcd` latencies and leader stability.

   API server responsiveness.

   Resource usage on control plane nodes (should be dominated by system components).

Outcome:

User workloads causing resource contention on control plane nodes are identified.

Control plane nodes are properly tainted to prevent general workload scheduling, or problematic workloads are reconfigured/moved.

System components (`etcd`, API server) have their necessary resources protected, either through explicit requests/limits, Node Allocatable, or by removing contending workloads.

Control plane stability and performance are restored.

Scenario 149: GPU Device Plugin Registration Failure or Unhealthy Devices

Scenario: After installing an NVIDIA GPU device plugin DaemonSet, nodes with GPUs are not reporting `nvidia.com/gpu: <count>` as an allocatable resource. `kubectl describe node <gpu-node-name>` shows no GPU capacity or allocatable GPUs. Pods requesting GPUs are stuck in `Pending`. Alternatively, GPUs are reported, but pods fail to use them, with errors in pod logs like "CUDA driver version is insufficient for CUDA runtime version" or device access errors.

Solution - Context:

kubectl config use-context k8s-c40-ml

Steps:

1. **Verify GPU Device Plugin Pod Status:**

   Check the status of the GPU device plugin pods (e.g., `nvidia-device-plugin-daemonset`) on the GPU-enabled nodes:

```
kubectl get pods -n <device-plugin-namespace> -l app=nvidia-device-plugin-ds -o wide
```

If pods are crashing or not running, describe them and check logs:

```
kubectl describe pod <nvidia-device-plugin-pod-on-gpu-node> -n <device-plugin-namespace>
```

```
kubectl logs <nvidia-device-plugin-pod-on-gpu-node> -n <device-plugin-namespace>
```

Common device plugin pod errors:

Failure to find NVIDIA drivers or libraries.

Permission issues accessing `/dev/nvidia` devices or `/var/lib/kubelet/device-plugins/`.

Incompatibility with Kubelet version.

2. Check Kubelet Logs for Device Plugin Interaction:

On the GPU node, examine Kubelet logs:

```
sudo journalctl -u kubelet -f | grep -i 'device_plugin\|gpu'
```

Look for messages about:

Registering the `nvidia.com/gpu` device plugin.

Allocation of GPU resources to pods.

Errors communicating with the device plugin socket (`/var/lib/kubelet/device-plugins/kubelet.sock` or plugin-specific socket).

3. Verify NVIDIA Driver Installation and Health on the Node:

SSH into the GPU node.

Run `nvidia-smi`. This command should:

Execute without errors.

List all installed GPUs.

Show their health, temperature, utilization, and driver version.

If `nvidia-smi` fails or shows unhealthy GPUs, troubleshoot the NVIDIA driver installation, kernel module, or hardware itself.

Ensure the driver version is compatible with the CUDA toolkit version used by your applications and the device plugin.

4. Check Device Plugin Configuration:

Inspect the DaemonSet manifest or any ConfigMap used by the NVIDIA device plugin.

Look for configuration options like:

`MIG_STRATEGY` (if using Multi-Instance GPUs).

Path to `nvidia-container-cli`.

Environment variables that might influence device discovery or reporting (e.g., `NVIDIA_VISIBLE_DEVICES`).

5. Ensure `nvidia-container-toolkit` (or `nvidia-docker2`) is Installed:

For containers to use NVIDIA GPUs, the `nvidia-container-toolkit` (and its dependencies like `libnvidia-container`) must be installed on the node and the container runtime (Docker, containerd) must be configured to use `nvidia-container-runtime` as a runtime.

Check container runtime configuration (e.g., `/etc/docker/daemon.json` or `/etc/containerd/config.toml`).

6. For "CUDA driver version insufficient" errors in Pods:

This is a mismatch between the NVIDIA driver version installed on the host node and the CUDA toolkit version compiled into the application container.

Use `nvidia-smi` on the node to get the host driver version.

Check the Dockerfile or image details of the application container to find its CUDA toolkit version.

Consult the NVIDIA CUDA compatibility matrix. You may need to:

Upgrade the host NVIDIA driver.

Rebuild the application container with an older/compatible CUDA toolkit.

Use a base container image (e.g., from NVIDIA NGC) that matches the host driver's capabilities.

7. Resolve Issues:

Device Plugin Not Registering:

Fix driver issues on the node (reinstall, reboot).

Correct permissions for the device plugin pod if it can't access `/dev` or Kubelet sockets.

Ensure compatibility between device plugin version, Kubelet version, and driver version.

Restart the device plugin pod.

Unhealthy Devices:

Address hardware issues or driver corruption identified by `nvidia-smi`.

Application CUDA/Driver Mismatch:

Align host driver and container CUDA toolkit versions.

8. Verify GPU Allocation and Pod Execution:

After fixes, check `kubectl describe node <gpu-node-name>` for `nvidia.com/gpu` in allocatable resources.

Deploy a test pod requesting GPUs:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-test
spec:
  containers:
  - name: cuda-vector-add
    image: "nvidia/cuda:11.8.0-base-ubuntu22.04" # Use an image compatible with your setup
    command: ["/bin/sh", "-c"]
    args: ["nvidia-smi && sleep 3600"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

The pod should run, and `nvidia-smi` inside the pod should show the allocated GPU.

Outcome:

The NVIDIA device plugin successfully registers with Kubelet, and GPU resources are reported as allocatable on GPU nodes.

NVIDIA drivers on the host are healthy and compatible with the CUDA toolkit versions used in application containers.

Pods requesting `nvidia.com/gpu` are successfully scheduled and can utilize the GPUs within their containers.

Scenario 150: Custom Scheduler Plugin Causing Inconsistent Pod Placement or Scheduling Deadlocks

Scenario: A custom scheduler plugin, `region-affinity-scheduler`, designed to place pods in specific node regions based on a pod annotation (`location.example.com/region`), is behaving erratically. Sometimes pods are placed correctly, other times they are scheduled to unexpected regions, or, in rare cases, pods get stuck in `Pending` indefinitely even if suitable nodes seem available. There are no obvious errors in the main kube-scheduler logs.

Solution - Context:

kubectl config use-context k8s-c41-customsched

Steps:

1. Observe and Document Inconsistent Behavior:

   Deploy multiple test pods with different `location.example.com/region` annotations.

   Note which pods land correctly, which land incorrectly, and which get stuck.

```
# test-pod-region-a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-region-a-pod
  annotations:
    location.example.com/region: "us-east-1"
spec:
  containers:
  - name: test
```

```
    image: pause
```

Check pod events for any clues (though often custom plugin issues don't emit clear event errors unless the plugin itself crashes).

2. Enable Verbose Logging for the Custom Plugin:

The `kube-scheduler` component and its plugins often support verbosity levels. The method depends on how the plugin is integrated (compiled-in or as a scheduler extender).

If compiled-in, increasing kube-scheduler's overall verbosity (`--v=5` or higher in its manifest) might reveal more detailed logs from the custom plugin's Filter, Score, or Bind phases.

If the custom plugin has its own logging mechanism, configure it for higher verbosity.

3. Review Custom Plugin Code Logic:

Thoroughly audit the code for `region-affinity-scheduler`:

Filter Plugin:

Does it correctly parse the `location.example.com/region` annotation from the pod?

Does it correctly read region labels from nodes (e.g., `topology.kubernetes.io/region`)?

Are comparisons case-sensitive when they shouldn't be (or vice-versa)?

Does it handle pods without the annotation gracefully (e.g., by allowing them on any node or filtering them out as intended)?

Error handling: What happens if a node label is missing?

Score Plugin (if applicable):

How does it score nodes? Is the logic sound?

Could it be producing non-deterministic scores or scores that conflict with other default scorer plugins?

Statefulness/Caching: Does the plugin maintain any internal state or cache? Could this state become stale or corrupted, leading to inconsistent decisions?

Concurrency Issues: If the plugin accesses shared data structures, are there proper locks to prevent race conditions?

4. Inspect Kube-scheduler Metrics for Plugin Performance:

If Prometheus is scraping scheduler metrics, look at:

scheduler_framework_extension_point_duration_seconds_bucket{extension_point="Filter"}` and filter by plugin name if possible (newer Kubernetes versions might label these by plugin).

Similar metrics for `Score`, `PreFilter`, etc.

Extremely high latencies or error counts for your custom plugin are red flags.

5. Test with Simplified Scenarios:

Reduce complexity:

Test with only one pod and one node of the target region.

Test with a pod that should not match any region.

Disable other custom scheduler plugins temporarily if multiple are active.

This helps isolate whether the issue is with the core logic or interactions.

6. For Scheduling Deadlocks (Stuck Pending Pods):

This is often the hardest to debug. Possibilities:

Filter Mismatch: Your custom Filter plugin might be incorrectly rejecting all nodes, even suitable ones, due to a bug.

Conflicting Plugins: Your plugin might conflict with default plugins. For example, if your plugin only allows nodes in "region-A", but default predicates (NodeAffinity, TaintToleration) then filter out all nodes in "region-A", no node will be viable.

Scoring producing all zeros: If your Score plugin (and others) return a score of 0 for all nodes that passed filtering, the pod might not be scheduled.

Plugin Panic/Crash: A severe bug could cause the plugin to panic during a specific pod's evaluation, potentially disrupting the scheduling cycle for that pod. Kube-scheduler logs should show this if verbosity is high.

7. Implement Fixes in Plugin Code:

Based on code review and verbose logs:

Correct logic errors in annotation parsing, label matching, or scoring.

Add robust error handling and defensive programming (e.g., for missing annotations/labels).

Address any caching or concurrency bugs.

Ensure the plugin interacts correctly with the scheduler framework's expected behavior (e.g., returning appropriate status codes from Filter).

8. Redeploy Scheduler with Fixed Plugin:

Recompile `kube-scheduler` with the updated plugin (if compiled-in) or redeploy the scheduler extender.

This usually involves updating the `kube-scheduler` static pod manifest and letting Kubelet restart it.

9. Rigorous Retesting:

Re-run all test scenarios (Step 1 and Step 5) to confirm consistent and correct behavior.

Monitor scheduler logs and metrics.

Outcome

The logical flaw, bug, or misinteraction within the custom scheduler plugin `region-affinity-scheduler` is identified.

The plugin's code is corrected and thoroughly tested.

The custom scheduler plugin now reliably and consistently places pods according to the defined regional affinity policy, and no longer causes scheduling deadlocks.

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.