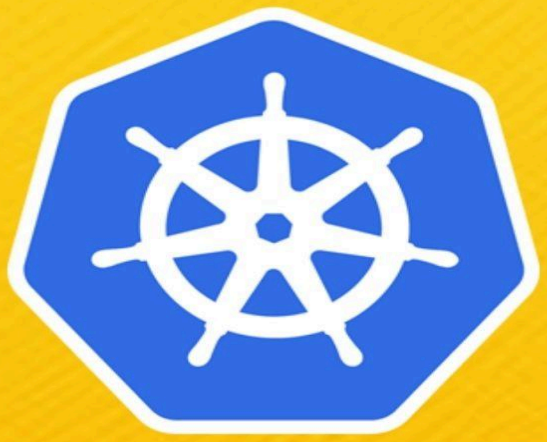
Kubernetes Scaling Troubleshooting



50 ERRORS

PART 2

1. Kubernetes Pod Not Scaling as Expected

Problem Statement

You have configured Horizontal Pod Autoscaler (HPA) to scale pods based on CPU or memory usage, but the number of pods is not increasing as expected under high load.

What Needs to Be Analyzed

- Check if the HPA is correctly configured and attached to the deployment.
- Verify if the CPU and memory requests/limits are set properly for the pods.
- Inspect `metrics-server` to ensure it is running and collecting resource usage data.
- Examine the HPA event logs for possible errors.

How to Resolve Step by Step

1. **Check HPA Configuration:**

```
Unset
kubectl get hpa
```

   - Ensure HPA is configured for the right deployment and resource type (CPU/memory).
2. **Inspect Metrics Server:**

```
Unset
kubectl get deployment -n kube-system | grep metrics-server
```

   - If missing, install it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

3. **Verify Resource Requests and Limits:**

```
Unset
kubectl get pod <pod-name> -o yaml | grep resources -A 5
```

    - Ensure CPU/memory requests are properly defined, as HPA relies on these values.
4. **Check HPA Logs:**

```
Unset
kubectl describe hpa <hpa-name>
```

    - Look for events like `unable to fetch metrics` or `scaling limited due to missing requests`.
5. **Manually Test Scaling:**

```
Unset
kubectl run test-pod --image=busybox -- sh -c "while true; do :; done"
```

    - This should create CPU load and trigger scaling.

Skills Required to Resolve This Issue

- Understanding of Kubernetes HPA and resource metrics.
- Familiarity with `metrics-server` and troubleshooting resource limits.
- Debugging `kubectl` commands and reading logs.

Conclusion

If pods are not scaling as expected, checking the HPA configuration, resource requests, and `metrics-server` setup is crucial. Properly defined resource requests and a running `metrics-server` ensure effective autoscaling.

---

2. Kubernetes Cluster Over-Scaling Issues

Problem Statement

The Kubernetes cluster is scaling more pods than necessary, consuming excessive resources and increasing cloud costs.

What Needs to Be Analyzed

- Check the HPA configuration to ensure the correct min/max replicas are set.
- Inspect the actual resource usage versus configured thresholds.
- Verify if multiple HPA rules are conflicting.
- Analyze if there's a sudden surge in requests causing scaling.

How to Resolve Step by Step

1. **Check the HPA Settings:**

```
Unset
kubectl describe hpa <hpa-name>
```

   - Look for misconfigured thresholds that trigger unnecessary scaling.
2. **Analyze Pod Resource Usage:**

```
Unset
kubectl top pods
```

   - Ensure scaling is happening due to actual resource demand.
3. **Check for Conflicting HPA Rules:**

```
Unset
kubectl get hpa --all-namespaces
```

   - Ensure multiple HPAs aren't scaling the same deployment.
4. **Inspect Incoming Traffic:**

```
Unset
kubectl logs -l app=<app-name>
```

- Look for sudden traffic spikes causing autoscaling.

5. **Reduce Scaling Sensitivity (If Needed):**

```
Unset
kubectl edit hpa <hpa-name>
```

- Adjust target CPU/memory utilization and cooldown period.

Skills Required to Resolve This Issue

- Understanding HPA min/max replica settings.
- Ability to analyze Kubernetes logs and metrics.
- Troubleshooting scaling anomalies in real-time.

Conclusion

Over-scaling in Kubernetes can lead to resource wastage and cost overruns. Reviewing HPA configurations, monitoring actual resource usage, and adjusting thresholds can optimize autoscaling behavior.

---

3. Node Resource Exhaustion Causing Scaling Failure

Problem Statement

Kubernetes attempts to scale pods, but new pods fail to schedule due to node resource exhaustion.

What Needs to Be Analyzed

- Check available CPU and memory on nodes.
- Inspect pending pods to see why they are not scheduling.
- Look for resource fragmentation issues preventing efficient scaling.

How to Resolve Step by Step

1. **Check Node Resource Usage:**

```
Unset
kubectl top nodes
```

   - Identify if nodes are running out of CPU/memory.
2. **Inspect Pending Pods:**

```
Unset
kubectl get pods --field-selector=status.phase=Pending
kubectl describe pod <pending-pod-name>
```

   - Look for errors like `Insufficient CPU` or `Insufficient Memory`.
3. **Check Pod Distribution:**

```
Unset
kubectl get nodes -o wide
```

   - Ensure pods are evenly distributed across nodes.
4. **Scale Cluster Nodes (If Needed):**
   - If using a cloud provider with autoscaling:

```
Unset
kubectl scale node-pool --replicas=5
```

5. **Evict Unused Pods:**

```
Unset
kubectl delete pod <unused-pod-name>
```

   - Frees up resources for critical applications.

Skills Required to Resolve This Issue

- Ability to analyze Kubernetes node and pod resource metrics.
- Experience with cluster autoscaling mechanisms.
- Troubleshooting pending pod scheduling issues.

Conclusion

Node resource exhaustion can prevent Kubernetes from scaling. Monitoring node capacity, removing unused pods, and enabling cluster autoscaling help ensure efficient scaling.

---

4. Kubernetes Pods Stuck in CrashLoopBackOff Due to Scaling Issues

Problem Statement

Pods keep restarting due to a `CrashLoopBackOff` state when scaling up under high load.

What Needs to Be Analyzed

- Inspect the pod logs to identify failure reasons.
- Check if the application is failing due to resource limits.
- Verify if scaling up is causing database connection failures.

How to Resolve Step by Step

1. **Check Pod Logs for Errors:**

```
Unset
kubectl logs <crashing-pod-name>
```

   - Look for errors like out-of-memory (OOM) or connection timeouts.
2. **Inspect Resource Limits:**

```
Unset
kubectl describe pod <pod-name>
```

   - If the pod is running out of memory, increase its limits.
3. **Check Readiness/Liveness Probes:**

```
Unset
kubectl get deployment <deployment-name> -o yaml | grep -A 5
livenessProbe
```

   - Ensure probes are correctly configured to prevent false failures.
4. **Monitor Database Connections (If Needed):**
   - If scaling is causing too many connections, increase DB limits.
5. **Manually Restart the Pod (For Testing):**

```
Unset
kubectl delete pod <pod-name>
```

Skills Required to Resolve This Issue

- Understanding Kubernetes pod health checks.
- Debugging pod logs and application failures.
- Adjusting resource limits and database configurations.

Conclusion

`CrashLoopBackOff` errors during scaling often stem from resource limits, application failures, or database connection issues. Analyzing logs and adjusting configurations can resolve the problem.

5. Kubernetes HPA Not Scaling Down Pods After Load Decreases

Problem Statement

After a traffic spike, the Horizontal Pod Autoscaler (HPA) successfully scales up the pods, but when the load decreases, the number of pods does not scale down as expected, leading to unnecessary resource consumption.

What Needs to Be Analyzed

- Check if HPA has the correct `minReplicas` value.
- Analyze if the CPU/memory usage has indeed dropped below the threshold.
- Verify if the cooldown period (`stabilizationWindowSeconds`) is preventing immediate scaling down.
- Ensure there are no pending requests keeping the pods active.

How to Resolve Step by Step

1. **Check HPA Status and Min Replicas:**

```
Unset
kubectl describe hpa <hpa-name>
```

   - Ensure `minReplicas` is set appropriately (not too high).
2. **Check Current CPU/Memory Usage:**

```
Unset
kubectl top pods
```

   - If CPU/memory is still high, find out why (e.g., long-running requests).
3. **Analyze HPA Events for Issues:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp | grep HPA
```

   - Look for messages about why scaling down is not happening.
4. **Modify the Cooldown Period If Needed:**

```
Unset
kubectl edit hpa <hpa-name>
```

- Adjust `stabilizationWindowSeconds` to allow faster scale-down.

5. **Force a Scale Down (For Testing):**

```
Unset
kubectl scale deployment <deployment-name> --replicas=<desired-count>
```

- Check if HPA properly resizes the pods afterward.

Skills Required to Resolve This Issue

- Knowledge of HPA parameters like `minReplicas`, `cooldown period`, and scaling triggers.
- Experience in analyzing Kubernetes metrics (`kubectl top`).
- Debugging Kubernetes events related to HPA.

Conclusion

If HPA is not scaling down, reviewing CPU/memory usage, cooldown settings, and HPA logs can help diagnose and resolve the issue efficiently.

---

6. Cluster Autoscaler Not Scaling Nodes Even When Pods Are Pending

Problem Statement

The Cluster Autoscaler is enabled, but when Kubernetes tries to schedule new pods, they remain in a `Pending` state due to insufficient node resources, and new nodes are not added automatically.

What Needs to Be Analyzed

- Check if the Cluster Autoscaler is running and properly configured.
- Inspect the pod conditions to verify why they are pending.
- Analyze cloud provider scaling policies (if running on AWS, GCP, Azure, etc.).

How to Resolve Step by Step

1. **Check Cluster Autoscaler Status:**

```
Unset
kubectl get pods -n kube-system | grep cluster-autoscaler
```

   - If missing, install it for your cloud provider:

```
Unset
kubectl apply -f cluster-autoscaler.yaml
```

2. **Inspect Pending Pods for Scheduling Issues:**

```
Unset
kubectl get pods --field-selector=status.phase=Pending

kubectl describe pod <pending-pod-name>
```

   - Look for `Insufficient CPU` or `Insufficient Memory` messages.
3. **Verify Node Group Autoscaling Configuration:**

```
Unset
kubectl get nodes -o wide
```

- Ensure autoscaling is enabled in the cloud provider settings (AWS ASG, GCP Node Pool, Azure VMSS).

4. **Check Autoscaler Logs for Errors:**

```
Unset
kubectl logs -n kube-system -l app=cluster-autoscaler
```

- Look for messages like `failed to scale up node group`.

5. **Manually Increase Node Count (For Testing):**

```
Unset
kubectl scale node-pool --replicas=<desired-count>
```

- If manual scaling works, the issue is likely with the autoscaler settings.

Skills Required to Resolve This Issue

- Understanding Kubernetes Cluster Autoscaler and cloud provider scaling policies.
- Debugging pending pods and node resource availability.
- Analyzing logs for autoscaler errors.

Conclusion

If the Cluster Autoscaler is not scaling nodes, verifying its configuration, checking pending pod logs, and reviewing cloud provider settings are key troubleshooting steps.

---

7. Kubernetes Deployment Scaling But New Pods Failing to Start

Problem Statement

When scaling a deployment, new pods are created but fail to start due to image pull failures, incorrect configurations, or resource constraints.

What Needs to Be Analyzed

- Check pod events to identify the failure reason.
- Verify if the container image exists and is accessible.
- Inspect resource availability on nodes.

How to Resolve Step by Step

1. **Check Pod Events for Failure Reason:**

```
Unset
kubectl describe pod <failing-pod-name>
```

   - Look for messages like `ErrImagePull`, `CrashLoopBackOff`, or `CreateContainerConfigError`.

2. **Verify the Container Image is Available:**

```
Unset
kubectl get pod <pod-name> -o jsonpath="{.spec.containers[*].image}"
```

   - Try pulling it manually:

```
Unset
docker pull <image-name>
```

3. **Check Node Resource Availability:**

```
Unset
kubectl describe node <node-name>
```

```
kubectl top nodes
```

- Ensure enough CPU and memory are available.

4. **Fix Image Pull Secrets (If Needed):**

```
Unset
kubectl create secret docker-registry my-secret \
  --docker-server=<registry-url> \
  --docker-username=<username> \
  --docker-password=<password>
```

- Then link it to your deployment.

5. **Restart Failing Pods (For Testing):**

```
Unset
kubectl delete pod <failing-pod-name>
```

Skills Required to Resolve This Issue

- Troubleshooting Kubernetes pod failures using logs and events.
- Debugging container image issues and registry authentication.
- Understanding Kubernetes resource constraints.

Conclusion

If new pods fail to start during scaling, analyzing pod events, checking image availability, and verifying node resources help identify and fix the root cause.

---

8. Kubernetes Pods Not Scaling Beyond a Certain Limit Despite HPA Settings

Problem Statement

Even though the Horizontal Pod Autoscaler (HPA) is configured to scale pods up to a high limit, the number of running pods does not increase beyond a specific number, even under high CPU or memory load.

What Needs to Be Analyzed

- Verify if `maxReplicas` in the HPA is set correctly.
- Check if there are resource quota limits at the namespace level.
- Ensure there are enough available nodes in the cluster to accommodate more pods.
- Analyze HPA events to see if scaling failures are being logged.

How to Resolve Step by Step

1. **Check HPA Settings for maxReplicas:**

```
Unset
kubectl describe hpa <hpa-name>
```

   - If `maxReplicas` is too low, increase it in the deployment manifest.
2. **Check Namespace Resource Quotas:**

```
Unset
kubectl get resourcequotas -n <namespace>
```

   - If limits are preventing scaling, increase the allowed CPU/memory for the namespace.
3. **Verify Node Availability for Additional Pods:**

```
Unset
kubectl get nodes

kubectl describe nodes

kubectl top nodes
```

   - If nodes are fully utilized, consider enabling Cluster Autoscaler to provision new nodes.

4. **Analyze HPA Events for Scaling Errors:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp | grep HPA
```

   - Look for messages like `not enough resources available to schedule pods`.
5. **Manually Scale Deployment for Testing:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=<desired-count>
```

   - If this fails, investigate cluster limits or workload constraints.

Skills Required to Resolve This Issue

- Understanding Kubernetes HPA scaling behavior.
- Debugging namespace resource quotas and node resource availability.
- Analyzing Kubernetes events for scaling issues.

Conclusion

If HPA does not scale beyond a certain limit, checking `maxReplicas`, namespace quotas, and node availability helps identify and fix the bottleneck.

---

9. Kubernetes Horizontal Pod Autoscaler (HPA) Scaling Too Aggressively

Problem Statement

The HPA is scaling up pods too quickly, creating excessive resource consumption, even when the traffic or CPU/memory load does not justify such rapid scaling.

What Needs to Be Analyzed

- Check the scaling threshold values in the HPA.
- Analyze whether the metric calculation is inaccurate.
- Ensure the cooldown period (`stabilizationWindowSeconds`) is properly set.
- Verify if spikes in CPU/memory usage are triggering unnecessary scaling.

How to Resolve Step by Step

1. **Check HPA Thresholds:**

```
Unset
kubectl describe hpa <hpa-name>
```

   - Ensure CPU/memory threshold values are set appropriately (not too low).
2. **Analyze Metric Trends Using Metrics Server:**

```
Unset
kubectl top pods
```

   - If momentary CPU/memory spikes are causing scaling, increase the cooldown period.
3. **Modify the HPA Cooldown Period to Avoid Frequent Scaling:**

```
Unset
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

   - Apply the updated HPA configuration.
4. **Use Average Utilization Instead of Instantaneous Metrics:**

```
Unset
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: AverageUtilization
        averageUtilization: 70
```

- This ensures scaling is based on sustained load, not temporary spikes.

5. **Monitor Scaling Events:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp | grep HPA
```

- Look for logs indicating frequent unnecessary scaling.

Skills Required to Resolve This Issue

- Understanding Kubernetes HPA configuration and behavior.
- Debugging CPU/memory usage trends in Kubernetes.
- Modifying HPA settings to stabilize scaling decisions.

Conclusion

If HPA scales too aggressively, adjusting threshold values, enabling cooldown settings, and using average utilization instead of instant metrics can improve stability.

---

10. Kubernetes StatefulSet Not Scaling Properly

Problem Statement

When attempting to scale a StatefulSet, new pods are not being created, or existing pods are stuck in a `Pending` or `CrashLoopBackOff` state.

What Needs to Be Analyzed

- Check if persistent volumes (PVs) are available for the new replicas.
- Verify if the headless service for the StatefulSet is correctly defined.
- Ensure there are no affinity rules preventing scheduling.
- Analyze pod logs to identify startup issues.

How to Resolve Step by Step

1. **Check StatefulSet Scaling and Current Pods:**

```
Unset
kubectl get statefulset <statefulset-name>

kubectl get pods -l app=<statefulset-app>
```

   - Ensure the desired number of replicas matches the StatefulSet definition.
2. **Verify Persistent Volume Availability:**

```
Unset
kubectl get pvc -n <namespace>
```

   - If required PVs are not bound, check storage class settings.
3. **Inspect StatefulSet Events for Issues:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp | grep
StatefulSet
```

   - Look for errors like `volume claim pending` or `node affinity constraints`.
4. **Ensure Headless Service Exists for StatefulSet:**

```
Unset
kubectl get svc -n <namespace>
```

- If missing, recreate it:

```
Unset
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  clusterIP: None
  selector:
    app: my-statefulset
```

5. **Manually Create a Test StatefulSet Pod:**

```
Unset
kubectl run test-pod --image=<image> --restart=Never
```

- If it fails, check node and storage availability.

Skills Required to Resolve This Issue

- Understanding StatefulSet behavior, including persistent volumes.
- Debugging storage and scheduling issues.
- Working with Kubernetes networking and headless services.

Conclusion

If a StatefulSet is not scaling, checking persistent volume claims, headless service configuration, and affinity rules can help identify and resolve the issue.

11. Kubernetes Cluster Autoscaler Not Adding New Nodes

Problem Statement

The Cluster Autoscaler is enabled, but when the workload increases and more pods are scheduled, new nodes are not being provisioned.

What Needs to Be Analyzed

- Check if the Cluster Autoscaler is running and properly configured.
- Verify if there are node group limits preventing new nodes from being created.
- Analyze whether pod scheduling constraints (taints, affinity, resource requests) are blocking scaling.
- Inspect the autoscaler logs for errors.

How to Resolve Step by Step

1. **Check Cluster Autoscaler Deployment:**

```
Unset
kubectl get pods -n kube-system | grep cluster-autoscaler
```

   - If it is not running, restart it:

```
Unset
kubectl rollout restart deployment cluster-autoscaler -n kube-system
```

2. **Analyze Cluster Autoscaler Logs:**

```
Unset
kubectl logs -n kube-system -l
app.kubernetes.io/name=cluster-autoscaler
```

   - Look for errors such as `max node limit reached` or `scaling disabled`.
3. **Check Node Group Scaling Limits:**

```
Unset
kubectl describe nodes | grep Allocatable
```

- If `maxSize` is reached, increase the node group size in the cloud provider settings.

4. **Inspect Pending Pods for Scheduling Issues:**

```
Unset
kubectl get pods --all-namespaces --field-selector=status.phase=Pending

kubectl describe pod <pod-name>
```

- Look for messages like `nodeSelector mismatch` or `taint toleration issues`.

5. **Manually Trigger a Scale-Up Test:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=50
```

- Check if new nodes appear with `kubectl get nodes`.

Skills Required to Resolve This Issue

- Kubernetes Cluster Autoscaler troubleshooting.
- Cloud provider-specific scaling configuration.
- Understanding Kubernetes scheduling constraints.

Conclusion

If Cluster Autoscaler is not adding nodes, checking logs, node group limits, and pending pod scheduling constraints will help diagnose the issue.

---

12. Kubernetes Pod Disruption Budget (PDB) Preventing Scaling Down

Problem Statement

When trying to scale down a Deployment or StatefulSet, some pods are not terminating because of a configured **Pod Disruption Budget (PDB)**.

What Needs to Be Analyzed

- Verify if a PDB is defined for the application.
- Check the minimum available pod count specified in the PDB.
- Inspect whether ongoing voluntary evictions are being blocked.

How to Resolve Step by Step

1. **Check Existing PDBs:**

```
Unset
kubectl get pdb -n <namespace>
```

   - Identify if a budget is applied to the affected deployment.
2. **Describe the PDB to Analyze Limits:**

```
Unset
kubectl describe pdb <pdb-name> -n <namespace>
```

   - Look for `minAvailable` or `maxUnavailable` settings.
3. **Temporarily Patch the PDB to Allow Scaling:**

```
Unset
kubectl patch pdb <pdb-name> -n <namespace> --type='json' -p='[{"op":
"replace", "path": "/spec/minAvailable", "value": "0"}]'
```

   - This temporarily allows pods to terminate.
4. **Manually Delete the Stuck Pod to Validate:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

- If deletion is blocked, adjust PDB settings accordingly.

5. **Modify the PDB to a More Flexible Value:**

```
Unset
apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

  name: my-app-pdb

spec:

  minAvailable: 50%
```

- Apply the changes to reduce disruption constraints.

Skills Required to Resolve This Issue

- Understanding Kubernetes PDB behavior.
- Troubleshooting pod scaling and eviction.
- Managing PDBs dynamically for scaling scenarios.

Conclusion

If a PDB is preventing scaling down, adjusting `minAvailable` or `maxUnavailable` settings helps maintain availability while allowing scale-down operations.

---

13. Kubernetes Deployment Scaling Causes Container Restarts

Problem Statement

When increasing the number of replicas in a Deployment, some pods keep restarting, leading to instability.

What Needs to Be Analyzed

- Check pod logs for crash-related errors.
- Verify if resource requests and limits are causing pod evictions.
- Ensure there are no readiness/liveness probe failures.
- Analyze node resource availability for accommodating additional pods.

How to Resolve Step by Step

1. **Check Pod Logs for Errors:**

```
Unset
kubectl logs <pod-name> -n <namespace>
```

   - Look for errors such as `OutOfMemory` or `CrashLoopBackOff`.
2. **Analyze Resource Limits in the Deployment:**

```
Unset
kubectl describe deployment <deployment-name> -n <namespace>
```

   - If limits are too low, increase CPU/memory allocation.
3. **Check Readiness and Liveness Probe Failures:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - If probes are failing, adjust probe settings in the deployment YAML:

```
Unset
livenessProbe:
```

```
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

4. **Monitor Node Resource Availability:**

```
Unset
kubectl top nodes
```

   - If nodes are low on resources, consider enabling Cluster Autoscaler.
5. **Manually Delete a Failing Pod to Validate Recovery:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

   - If the new pod also restarts, resource constraints need to be adjusted.

Skills Required to Resolve This Issue

- Kubernetes pod troubleshooting and log analysis.
- Resource allocation tuning in Deployments.
- Configuring health probes properly.

Conclusion

If scaling a Deployment causes pod restarts, adjusting resource requests/limits, fixing health probes, and ensuring node availability can stabilize scaling.

---

14. Horizontal Pod Autoscaler (HPA) Not Scaling as Expected

Problem Statement

The Horizontal Pod Autoscaler (HPA) is enabled for a Deployment, but it does not scale the pods up or down despite increased or decreased load.

What Needs to Be Analyzed

- Check if HPA is configured correctly and is monitoring the right metrics.
- Verify if CPU or memory usage is crossing the defined thresholds.
- Inspect if the metric server is running and collecting data.
- Ensure that resource requests are defined in the pod specification.

How to Resolve Step by Step

1. **Check HPA Configuration:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Ensure it is targeting the correct deployment and has defined CPU/memory thresholds.
2. **Describe HPA for Detailed Analysis:**

```
Unset
kubectl describe hpa <hpa-name> -n <namespace>
```

   - Look for errors like `unable to fetch metrics` or `desired replica count does not change`.
3. **Verify Metrics Server is Running:**

```
Unset
kubectl get pods -n kube-system | grep metrics-server
```

   - If missing, deploy it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

4. **Check Resource Utilization Metrics:**

```
Unset
kubectl top pods -n <namespace>
```

   - If CPU/memory usage is low, the HPA may not trigger scaling.

5. **Ensure Resource Requests Are Defined in Deployment:**

```
Unset
resources:
  requests:
    cpu: "200m"
    memory: "256Mi"
```

   - If missing, HPA won't function correctly.

6. **Manually Increase Load to Trigger Scaling:**

```
Unset
kubectl run load-generator --image=busybox -- /bin/sh -c "while true;
do wget -q -O- http://<service-name>; done"
```

   - Check if HPA scales up the pods accordingly.

Skills Required to Resolve This Issue

- Kubernetes HPA and metrics troubleshooting.
- Monitoring resource utilization and scaling logic.
- Debugging issues related to the Metrics Server.

Conclusion

If HPA is not scaling as expected, verifying metric availability, resource requests, and ensuring the Metrics Server is functional will help resolve the issue.

15. Kubernetes Node Resource Pressure Preventing Scaling

Problem Statement

New pods are not being scheduled even though the Deployment is scaled up, and some existing pods may be evicted due to `NodePressure` conditions.

What Needs to Be Analyzed

- Check node resource availability (CPU, memory, disk).
- Verify if the node is under `DiskPressure`, `MemoryPressure`, or `PIDPressure`.
- Inspect if any pods have been evicted due to insufficient resources.

How to Resolve Step by Step

1. **Check Node Conditions:**

```
Unset
kubectl describe node <node-name>
```

   - Look for conditions like `MemoryPressure` or `DiskPressure`.
2. **Check Resource Utilization on Nodes:**

```
Unset
kubectl top nodes
```

   - If CPU/memory usage is high, autoscaling may be required.
3. **Check if Pods Are Being Evicted:**

```
Unset
kubectl get pods --all-namespaces | grep Evicted
```

   - If many pods are evicted, resource constraints need adjustment.
4. **Free Up Resources on Overloaded Nodes:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

- Consider moving non-essential workloads to other nodes.

5. **Increase Node Pool Size if Scaling is Blocked:**

```
Unset
gcloud container clusters resize <cluster-name> --node-pool
<node-pool-name> --num-nodes=5
```

- Adjust based on your cloud provider.

Skills Required to Resolve This Issue

- Kubernetes node resource management.
- Understanding node pressure conditions.
- Cloud provider node scaling configurations.

Conclusion

If node resource pressure prevents scaling, freeing up resources, adjusting node limits, or enabling autoscaling will help resolve the issue.

---

16. Cluster Autoscaler Not Adding New Nodes

Problem Statement

The Cluster Autoscaler is enabled, but when the workload increases, new nodes are not being added, leading to pod scheduling failures due to resource constraints.

What Needs to Be Analyzed

- Check if the Cluster Autoscaler is enabled and running.
- Verify if the node pool has reached its maximum limit.
- Inspect if pending pods exist due to insufficient resources.
- Ensure the IAM permissions and cloud provider configurations allow scaling.

How to Resolve Step by Step

1. **Check if Cluster Autoscaler is Enabled:**

```
Unset
kubectl get deployment cluster-autoscaler -n kube-system
```

   - If it is not running, enable it via your cloud provider's CLI.
2. **Verify Node Pool Scaling Limits:**

```
Unset
gcloud container node-pools describe <node-pool-name> --cluster
<cluster-name>
```

   - Ensure `max-nodes` is greater than the current count.
3. **Check for Pending Pods Due to Insufficient Resources:**

```
Unset
kubectl get pods --all-namespaces | grep Pending
```

   - If pods are pending, check the events:

```
Unset
kubectl describe pod <pod-name>
```

- Look for errors like `Insufficient CPU` or `Insufficient Memory`.

4. **Check Cluster Autoscaler Logs for Issues:**

```
Unset
kubectl logs -f deployment/cluster-autoscaler -n kube-system
```

- Look for messages like `max node limit reached` or `autoscaler failed to scale`.

5. **Manually Trigger Scaling for Testing:**
   - Increase the replicas to check if autoscaler responds:

```
Unset
kubectl scale deployment <deployment-name> --replicas=50
```

6. **Ensure IAM Permissions Allow Node Scaling (Cloud-Specific):**
   - Example for GKE:

```
Unset
gcloud projects add-iam-policy-binding <project-id> \
  --member serviceAccount:<service-account> \
  --role roles/container.admin
```

- AWS EKS or Azure AKS users should check their respective IAM policies.

Skills Required to Resolve This Issue

- Kubernetes Cluster Autoscaler troubleshooting.
- Cloud provider scaling policies and IAM configurations.
- Debugging pod scheduling and resource constraints.

Conclusion

If Cluster Autoscaler is not adding nodes, verifying scaling limits, IAM permissions, and pending pod statuses will help in identifying and resolving the issue.

17. Kubernetes Deployment Not Scaling Even After HPA Increases Replicas

Problem Statement

The Horizontal Pod Autoscaler (HPA) increases the replica count, but the actual number of running pods does not change.

What Needs to Be Analyzed

- Check if the Deployment or ReplicaSet is stuck at a lower count.
- Verify if new pods are failing due to resource constraints.
- Ensure there are no quota restrictions preventing scaling.
- Check if there are scheduling constraints such as node affinity or taints.

How to Resolve Step by Step

1. **Check the Deployment and ReplicaSet Status:**

```
Unset
kubectl get deployment <deployment-name> -n <namespace>

kubectl get replicaset -n <namespace>
```

   - If the desired replicas count is increased but actual pods remain unchanged, there may be an issue with scheduling.
2. **Inspect Pod Events for Failures:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for errors like `FailedScheduling` or `Quota exceeded`.
3. **Check Resource Quotas in the Namespace:**

```
Unset
kubectl get resourcequotas -n <namespace>
```

   - If quotas are too restrictive, increase the limits:

```
Unset
kubectl edit resourcequota <quota-name> -n <namespace>
```

4. **Verify Node Affinity and Taints:**

```
Unset
kubectl describe nodes | grep -A10 "Taints"
```

   - If taints exist, ensure the pods have tolerations:

```
Unset
tolerations:
- key: "example-taint"
  operator: "Exists"
  effect: "NoSchedule"
```

5. **Manually Scale Deployment for Testing:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=10
kubectl get pods -n <namespace>
```

   - If pods are not increasing, check for scheduling issues.

Skills Required to Resolve This Issue

- Kubernetes Deployment and ReplicaSet debugging.
- Understanding quota restrictions and node scheduling constraints.
- Resource allocation and scaling troubleshooting.

Conclusion

If HPA scales up replicas but pods do not increase, resolving quota restrictions, node taints, and scheduling constraints will help fix the issue.

18. Kubernetes Pods Not Scaling Due to Resource Requests and Limits

Problem Statement

Even after scaling a Deployment manually or via HPA, some pods remain in a `Pending` state because they request more resources than available.

What Needs to Be Analyzed

- Check if resource requests exceed node capacity.
- Verify if nodes have enough CPU/memory to accommodate additional pods.
- Ensure there are no hard limits restricting pod scheduling.

How to Resolve Step by Step

1. **Check Pending Pods and Their Events:**

```
Unset
kubectl get pods -n <namespace> | grep Pending

kubectl describe pod <pod-name> -n <namespace>
```

   - Look for messages like `Insufficient CPU` or `Insufficient Memory`.
2. **Inspect Resource Requests and Limits in Deployment:**

```
Unset
resources:
  requests:
    cpu: "500m"
    memory: "512Mi"
  limits:
    cpu: "1000m"
    memory: "1024Mi"
```

   - If requests are too high, reduce them and redeploy.
3. **Check Node Capacity:**

```
Unset
kubectl top nodes
```

- If nodes are running low on resources, consider scaling the cluster.

4. **Manually Reduce Resource Requests and Reapply:**

```
Unset
kubectl edit deployment <deployment-name> -n <namespace>
```

- Adjust CPU/memory requests to a lower value and save.

5. **If Necessary, Add More Nodes to the Cluster:**

```
Unset
gcloud container clusters resize <cluster-name> --node-pool
<node-pool-name> --num-nodes=5
```

- Adjust this command for AWS EKS or Azure AKS.

Skills Required to Resolve This Issue

- Kubernetes pod scheduling and resource allocation.
- Monitoring node CPU/memory usage.
- Adjusting resource requests and limits for optimal scaling.

Conclusion

If pods fail to scale due to high resource requests, adjusting CPU/memory values and ensuring sufficient node capacity will resolve the problem.

---

19. Kubernetes Horizontal Pod Autoscaler (HPA) Not Triggering Scaling

Problem Statement

The Horizontal Pod Autoscaler (HPA) is configured, but it does not increase or decrease pod replicas even when CPU or memory utilization crosses the defined threshold.

What Needs to Be Analyzed

- Check if the HPA is correctly attached to the deployment.
- Verify if metrics-server is running and collecting resource metrics.
- Inspect if resource requests are correctly defined in the deployment.
- Ensure CPU or memory usage is genuinely exceeding the threshold.

How to Resolve Step by Step

1. **Check HPA Configuration and Status:**

```
Unset
kubectl get hpa -n <namespace>

kubectl describe hpa <hpa-name> -n <namespace>
```

   - Ensure the `TARGETS` column shows actual CPU/memory usage.
2. **Verify Resource Requests and Limits in Deployment:**

```
Unset
kubectl get deployment <deployment-name> -o yaml | grep -A5 resources
```

   - Ensure the `requests` section is set, as HPA depends on it.
3. **Check If Metrics Server is Running:**

```
Unset
kubectl get deployment metrics-server -n kube-system
```

   - If missing, deploy it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

4. **Test Metrics Collection:**

```
Unset
kubectl top pod -n <namespace>
```

   - If the output is empty, metrics-server is not collecting data correctly.

5. **Manually Increase Load to Test Scaling:**
   - Run a CPU-intensive workload inside a pod:

```
Unset
kubectl run -it --rm load-generator --image=busybox -- /bin/sh

while true; do wget -q -O- http://<service-name>; done
```

   - Monitor if HPA scales pods using:

```
Unset
kubectl get hpa -n <namespace> --watch
```

6. **Check for HPA Errors in Logs:**

```
Unset
kubectl logs -n kube-system deployment/metrics-server
```

   - Look for permission issues or missing metrics.

Skills Required to Resolve This Issue

- Kubernetes HPA debugging and troubleshooting.
- Working with metrics-server and resource utilization monitoring.
- Kubernetes logging and debugging techniques.

Conclusion

If HPA is not scaling, verifying metrics-server availability, ensuring correct resource requests, and testing with artificial load can help diagnose and fix the issue.

20. Kubernetes Cluster Autoscaler Scaling Down Incorrectly

Problem Statement

Cluster Autoscaler is removing nodes even when workloads are running, leading to evicted pods and application downtime.

What Needs to Be Analyzed

- Check the autoscaler logs for unexpected scale-down decisions.
- Inspect if node utilization is below the configured threshold.
- Verify pod disruption budgets to prevent unnecessary evictions.
- Ensure no misconfigurations in autoscaler parameters.

How to Resolve Step by Step

1. **Check the Autoscaler Logs for Scale-Down Decisions:**

```
Unset
kubectl logs -f deployment/cluster-autoscaler -n kube-system
```

   - Look for logs mentioning `removing node` or `scale down`.
2. **Verify Node Utilization Before Scale-Down:**

```
Unset
kubectl top nodes
```

   - If nodes are underutilized, autoscaler might remove them.
3. **Check Pod Disruption Budgets to Prevent Evictions:**

```
Unset
kubectl get pdb -n <namespace>

kubectl describe pdb <pdb-name> -n <namespace>
```

   - If `minAvailable` is too low, increase it to prevent pod evictions.
4. **Modify Cluster Autoscaler Scaling Settings:**

```
Unset
gcloud container clusters update <cluster-name> \
  --enable-autoscaling --min-nodes=2 --max-nodes=10
```

    - Ensure min-nodes is not set too low.
5. **Taint Critical Nodes to Prevent Scale-Down:**

```
Unset
kubectl taint nodes <node-name> key=value:NoSchedule
```

Skills Required to Resolve This Issue

- Kubernetes Cluster Autoscaler debugging.
- Understanding pod disruption budgets and taints.
- Cloud provider autoscaling configuration.

Conclusion

If the autoscaler removes active nodes, adjusting scaling policies, setting pod disruption budgets, and verifying node utilization can help fix the problem.

---

21. Kubernetes Pods Stuck in "Pending" State Due to Insufficient Resources

Problem Statement

Pods are stuck in a `Pending` state because the cluster lacks enough CPU or memory to schedule them.

What Needs to Be Analyzed

- Check if the cluster has enough resources available.
- Identify nodes that have insufficient capacity.
- Verify if resource requests/limits are too high for scheduling.
- Ensure cluster autoscaler is correctly configured (if enabled).

How to Resolve Step by Step

1. **Check Pod Status and Describe It:**

```
Unset
kubectl get pods -n <namespace>

kubectl describe pod <pod-name> -n <namespace>
```

   - Look for `FailedScheduling` events in the description.
2. **Check Cluster Node Resource Availability:**

```
Unset
kubectl top nodes
```

   - If nodes are fully utilized, new pods won't be scheduled.
3. **Verify Requested Resources in Deployment:**

```
Unset
kubectl get deployment <deployment-name> -o yaml | grep -A5 resources
```

   - Ensure the requested resources fit within node capacity.
4. **Manually Scale the Cluster (If Autoscaler is Disabled):**

   - On GKE:

```
Unset
gcloud container clusters resize <cluster-name> --node-pool <pool-name>
--num-nodes=<new-node-count>
```

- On AWS EKS:

```
Unset
eksctl scale nodegroup --name=<nodegroup-name> --cluster=<cluster-name>
--nodes=<count>
```

5. **Check and Enable Cluster Autoscaler:**

```
Unset
kubectl get deployment cluster-autoscaler -n kube-system
```

- If missing, install it based on the cloud provider's instructions.

6. **Lower Resource Requests to Fit Node Capacity:**

```
Unset
resources:
  requests:
    cpu: "200m"
    memory: "256Mi"
```

- Reduce requests to fit existing nodes.

Skills Required to Resolve This Issue

- Kubernetes scheduling and resource management.
- Cluster Autoscaler troubleshooting.
- Cloud provider-specific cluster scaling commands.

Conclusion

When pods are stuck in `Pending`, checking node capacity, scaling the cluster, and adjusting resource requests can help resolve the issue.

---

22. Kubernetes Scale-Up Fails Due to Pod Anti-Affinity Rules

Problem Statement

New pods are not getting scheduled due to strict pod anti-affinity rules, even when enough resources are available.

What Needs to Be Analyzed

- Check if pod anti-affinity rules are preventing scheduling.
- Verify if nodes meet the required affinity constraints.
- Inspect logs for scheduling failures related to affinity.

How to Resolve Step by Step

1. **Describe the Pod to Identify Scheduling Issues:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for errors mentioning `no nodes available` or `affinity rules not met`.
2. **Check Anti-Affinity Rules in the Deployment:**

```
Unset
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - my-app
        topologyKey: "kubernetes.io/hostname"
```

- If `requiredDuringScheduling` is too strict, change it to `preferredDuringScheduling`.

3. **Modify Anti-Affinity Rules to Allow More Flexibility:**

```
Unset
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - my-app
          topologyKey: "kubernetes.io/hostname"
```

4. **Manually Scale Nodes to Meet Affinity Requirements (If Needed):**

   - On GKE:

```
Unset
gcloud container clusters resize <cluster-name> --node-pool <pool-name>
--num-nodes=<new-count>
```

   - On AWS EKS:

```
Unset
eksctl scale nodegroup --name=<nodegroup-name> --cluster=<cluster-name>
--nodes=<count>
```

Skills Required to Resolve This Issue

- Kubernetes pod scheduling and affinity rules.
- YAML configuration and Kubernetes deployment updates.
- Cloud-based cluster scaling.

Conclusion

If strict anti-affinity rules prevent scaling, adjusting them to be more flexible or manually scaling nodes can resolve the issue.

23. Kubernetes Horizontal Pod Autoscaler (HPA) Not Scaling Pods

Problem Statement

The Horizontal Pod Autoscaler (HPA) is not increasing or decreasing the number of pods as expected, even under high load.

What Needs to Be Analyzed

- Check if HPA is correctly defined and enabled.
- Verify that the metrics server is running and providing accurate data.
- Confirm that CPU/memory thresholds are being crossed.
- Check if resource requests/limits are correctly set.

How to Resolve Step by Step

1. **Check the HPA Configuration:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Ensure the target deployment or stateful set is linked correctly.
2. **Describe the HPA for Debugging:**

```
Unset
kubectl describe hpa <hpa-name> -n <namespace>
```

   - Look for errors such as `unable to get metrics`.
3. **Ensure Metrics Server is Running:**

```
Unset
kubectl get pods -n kube-system | grep metrics-server
```

   - If missing, install it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

4. **Check Pod Metrics for CPU Usage:**

```
Unset
kubectl top pods -n <namespace>
```

   - If CPU usage is low, HPA won't trigger scaling.

5. **Ensure Correct Resource Requests/Limits in Deployment:**

```
Unset
resources:
  requests:
    cpu: "100m"
    memory: "256Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

   - If missing, add reasonable values.

6. **Manually Trigger Scaling to Validate HPA:**

```
Unset
kubectl scale --replicas=5 deployment/<deployment-name> -n <namespace>
```

7. **Check Event Logs for Errors:**

```
Unset
kubectl logs -n kube-system -l k8s-app=metrics-server
```

   - Look for errors such as `failed to scrape` or `metrics not available`.

Skills Required to Resolve This Issue

- Kubernetes HPA and scaling configurations.
- Kubernetes metrics server troubleshooting.
- Resource management in Kubernetes.

Conclusion

HPA scaling issues are usually caused by missing metrics servers, incorrect resource requests, or insufficient CPU load. Ensuring proper configurations and checking logs can resolve this issue.

---

24. Kubernetes Cluster Autoscaler Not Scaling Nodes

Problem Statement

The Kubernetes Cluster Autoscaler is not adding or removing nodes even when there is high resource demand.

What Needs to Be Analyzed

- Check if Cluster Autoscaler is installed and enabled.
- Verify logs to check if autoscaler detects the need for scaling.
- Ensure node group settings allow automatic scaling.
- Check for taints that might prevent pod scheduling.

How to Resolve Step by Step

1. **Check the Cluster Autoscaler Status:**

```
Unset
kubectl get deployment cluster-autoscaler -n kube-system
```

   - If missing, install it based on cloud provider instructions.
2. **Check Logs for Autoscaler Activity:**

```
Unset
kubectl logs -n kube-system deployment/cluster-autoscaler | grep
"scale"
```

   - Look for messages about scaling decisions.
3. **Verify Node Pool Scaling Limits:**
   - GKE:

```
Unset
gcloud container node-pools describe <pool-name> --cluster
<cluster-name>
```

   - AWS EKS:

```
Unset
eksctl get nodegroup --cluster=<cluster-name>
```

- Ensure `min` and `max` values allow scaling.

4. **Manually Scale the Node Pool to Validate Autoscaler Behavior:**

```
Unset
kubectl scale --replicas=5 deployment/<deployment-name> -n <namespace>
```

5. **Check for Node Taints That Prevent Scaling:**

```
Unset
kubectl describe node <node-name>
```

- If nodes have `NoSchedule` taints, remove them:

```
Unset
kubectl taint nodes <node-name> key=value:NoSchedule-
```

Skills Required to Resolve This Issue

- Kubernetes Cluster Autoscaler configurations.
- Cloud provider-specific node pool management.
- Debugging Kubernetes logs.

Conclusion

Cluster autoscaler issues often stem from misconfiguration, taints, or limits on node pools. Ensuring proper settings and reviewing logs can help resolve this issue.

---

25. Kubernetes Pods Not Scaling Due to Resource Quotas

Problem Statement

Pods are not scaling beyond a certain limit due to namespace resource quotas restricting CPU, memory, or pod count.

What Needs to Be Analyzed

- Check if a namespace resource quota exists.
- Verify if the requested resources exceed quota limits.
- Look for events related to quota violations.

How to Resolve Step by Step

1. **Check If a Resource Quota Exists in the Namespace:**

```
Unset
kubectl get resourcequota -n <namespace>
```

   - Example output:

```
Unset
NAME           CPU REQUEST   MEMORY REQUEST   PODS

quota-limit    2             4Gi              10
```

2. **Describe the Resource Quota for More Details:**

```
Unset
kubectl describe resourcequota quota-limit -n <namespace>
```

   - Look for exceeded limits.
3. **Check Pod Resource Requests and Limits:**

```
Unset
resources:
```

```
  requests:
    cpu: "500m"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "512Mi"
```

4. **Modify Resource Quotas If Necessary:**

```
Unset
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-limit
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    pods: "20"
```

5. Apply the updated quota:

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

6. **If Quotas Cannot Be Increased, Optimize Pod Resources:**
    - Lower resource requests to fit within the allowed limits.

Skills Required to Resolve This Issue

- Kubernetes resource quotas and namespace management.
- YAML configurations for resource limits.
- Troubleshooting Kubernetes pod scheduling.

Conclusion

Scaling issues due to resource quotas can be resolved by adjusting quota limits or optimizing resource requests within the defined constraints.

---

26. Kubernetes Pods Not Scaling Due to Insufficient Node Resources

Problem Statement

Pods are not scaling because there are not enough available resources (CPU, memory) on the existing nodes, and no new nodes are being added.

What Needs to Be Analyzed

- Check if nodes have sufficient CPU and memory.
- Verify if the cluster autoscaler is enabled and working.
- Analyze pod events to see if scheduling is failing due to resource constraints.

How to Resolve Step by Step

1. **Check Available Node Resources:**

```
Unset
kubectl describe nodes

kubectl top nodes
```

   - If nodes have high CPU/memory usage, they might not accommodate new pods.
2. **Check Events for Scheduling Failures:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp
```

   - Look for errors like `FailedScheduling` due to insufficient resources.
3. **Verify Node Limits in Cluster Autoscaler:**
   - GKE:

```
Unset
gcloud container clusters describe <cluster-name>
```

   - AWS EKS:

```
Unset
eksctl get nodegroup --cluster=<cluster-name>
```

- Ensure node pools allow auto-scaling beyond current limits.

4. **Manually Scale the Node Pool to Add More Nodes:**
   - GKE:

```
Unset
gcloud container clusters resize <cluster-name> --node-pool <pool-name>
--num-nodes=5
```

   - AWS EKS:

```
Unset
eksctl scale nodegroup --cluster=<cluster-name> --name=<nodegroup-name>
--nodes=5
```

5. **Verify Resource Requests in Deployment Configurations:**

```
Unset
resources:
  requests:
    cpu: "250m"
    memory: "512Mi"
  limits:
    cpu: "500m"
    memory: "1Gi"
```

   - Adjust resource requests to fit within available resources.

6. **Evict Unnecessary Pods to Free Up Resources:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes resource allocation and node scaling.
- Cloud provider-specific autoscaler settings.
- Analyzing Kubernetes events and logs.

Conclusion

Scaling issues due to insufficient node resources can be resolved by checking node availability, ensuring the autoscaler is configured correctly, and adjusting resource requests.

27. Kubernetes Pods Not Scaling Due to Pod Disruption Budgets (PDBs)

Problem Statement

Pods are not scaling down (reducing count) due to Pod Disruption Budgets (PDBs) preventing eviction.

What Needs to Be Analyzed

- Check if a Pod Disruption Budget (PDB) exists.
- Verify if the minimum available pod count is restricting scale-down.
- Ensure that voluntary disruptions allow termination.

How to Resolve Step by Step

1. **Check if a PDB Exists for the Affected Pods:**

```
Unset
kubectl get pdb -n <namespace>
```

   - Example output:

```
Unset
NAME              MIN AVAILABLE   MAX UNAVAILABLE

my-app-pdb        2               N/A
```

2. **Describe the PDB for More Details:**

```
Unset
kubectl describe pdb my-app-pdb -n <namespace>
```

   - Look for restrictions preventing scale-down.
3. **Modify the PDB to Allow Scaling:**

```
Unset
apiVersion: policy/v1
```

```
kind: PodDisruptionBudget
metadata:
  name: my-app-pdb
  namespace: my-namespace
spec:
  minAvailable: 1  # Reduce if necessary
```

4. Apply the updated PDB:

```
Unset
kubectl apply -f pdb.yaml
```

5. **Manually Evict a Pod to Test Scaling:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

6. **If PDB Is Not Needed, Delete It:**

```
Unset
kubectl delete pdb my-app-pdb -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes Pod Disruption Budgets and scheduling.
- YAML configurations for PDBs.
- Understanding workload availability requirements.

Conclusion

Scaling issues due to PDBs can be fixed by adjusting `minAvailable` settings or removing unnecessary PDB constraints.

---

28. Kubernetes Pods Not Scaling Due to Affinity and Node Selectors

Problem Statement

Pods are not scaling because node affinity rules or node selectors are restricting where new pods can be scheduled.

What Needs to Be Analyzed

- Check if node affinity rules exist.
- Verify if there are enough available nodes matching the criteria.
- Ensure node labels are correctly assigned.

How to Resolve Step by Step

1. **Check if the Deployment Has Node Selectors:**

```
Unset
spec:
  nodeSelector:
    disktype: ssd
```

   - If too restrictive, modify the selector.
2. **Check Pod Scheduling Failures in Events:**

```
Unset
kubectl get events --sort-by=.metadata.creationTimestamp
```

3. **Check Node Affinity Rules in Deployment Configuration:**

```
Unset
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
```

```
      nodeSelectorTerms:
      - matchExpressions:
        - key: disktype
          operator: In
          values:
          - ssd
```

- Remove or modify `requiredDuringSchedulingIgnoredDuringExecution` if it's too restrictive.

4. **List Nodes and Check Their Labels:**

```
Unset
kubectl get nodes --show-labels
```

- If labels are missing, add them:

```
Unset
kubectl label nodes <node-name> disktype=ssd
```

5. **Remove Node Affinity If Not Needed:**

```
Unset
affinity: {}
```

6. Apply the updated deployment:

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

Skills Required to Resolve This Issue

- Kubernetes affinity and anti-affinity rules.
- Kubernetes scheduling and node labeling.
- YAML configurations for affinity settings.

Conclusion

Scaling issues caused by affinity or node selectors can be fixed by ensuring that node labels match the required criteria or by relaxing affinity rules.

29. Kubernetes Pods Not Scaling Due to Overprovisioned Resources

Problem Statement

Pods are not scaling because existing pods are using more resources than expected, causing the cluster to run out of capacity.

What Needs to Be Analyzed

- Check actual resource usage vs. requested resources.
- Identify which pods are consuming excessive resources.
- Optimize pod resource requests and limits.

How to Resolve Step by Step

1. **Check Pod Resource Usage:**

```
Unset
kubectl top pods -n <namespace>
```

   - Identify pods consuming excessive CPU or memory.
2. **Check Pod Resource Requests and Limits:**

```
Unset
resources:
  requests:
    cpu: "500m"
    memory: "512Mi"
  limits:
    cpu: "1"
    memory: "1Gi"
```

   - Reduce resource limits if they are too high.
3. **Check Node Resource Utilization:**

```
Unset
kubectl top nodes
```

    - If nodes are at full capacity, consider adding more nodes.
4. **Evict High-Usage Pods to Free Resources:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

5. **Manually Scale the Deployment to Verify Scaling Works:**

```
Unset
kubectl scale --replicas=5 deployment/<deployment-name> -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes resource management.
- Performance monitoring with `kubectl top`.
- Kubernetes pod scheduling and eviction.

Conclusion

Scaling issues due to overprovisioned resources can be fixed by optimizing resource requests and limits, monitoring pod usage, and ensuring nodes have enough capacity.

---

30. Kubernetes Pods Not Scaling Due to Insufficient IP Addresses in CNI

Problem Statement

Pods are failing to scale because the cluster has run out of available IP addresses for new pods. This is commonly seen in Kubernetes clusters using Container Network Interface (CNI) plugins like Calico, Cilium, or Amazon VPC CNI.

What Needs to Be Analyzed

- Check if the network plugin is running out of IP addresses.
- Verify the current IP allocation for nodes and pods.
- Ensure that IP range allocations in the CNI are properly configured.

How to Resolve Step by Step

1. **Check IP Usage in the Cluster:**

```
Unset
kubectl get pods -A -o wide
```

   - Look for signs of pending pods due to lack of IPs.
2. **Check Node IP Allocation Status:**
   - For AWS EKS (Amazon VPC CNI):

```
Unset
kubectl get nodes -o json | jq '.items[].status.allocatable'
```

   - For Calico CNI:

```
Unset
calicoctl ipam check
```

3. **Verify the CNI Configuration for IP Address Limits:**

```
Unset
kubectl get cm aws-node -n kube-system -o yaml
```

- For AWS EKS, increase `WARM_ENI_TARGET` or `WARM_IP_TARGET`.

4. **Increase IP Address Allocation (Based on CNI Type):**
   - **Amazon VPC CNI:** Increase ENI limits using:

```
Unset
aws ec2 modify-instance-metadata-options --instance-id
<node-instance-id> --http-endpoint enabled
```

   - **Calico CNI:** Expand CIDR range:

```
Unset
apiVersion: operator.tigera.io/v1

kind: Installation

metadata:

  name: default

spec:

  calicoNetwork:

    ipPools:

      - cidr: 192.168.0.0/16  # Increase this range if needed
```

5. **Manually Restart CNI DaemonSet to Apply Changes:**

```
Unset
kubectl delete pod -n kube-system -l k8s-app=aws-node
```

Skills Required to Resolve This Issue

- Kubernetes networking and CNI plugins.
- Understanding IP allocation for pods and nodes.
- Cloud provider-specific networking (AWS, GCP, Azure).

Conclusion

Scaling issues due to insufficient IP addresses can be resolved by increasing the available IP pool in the CNI plugin and ensuring nodes have enough IP allocations for new pods.

31. Kubernetes Horizontal Pod Autoscaler (HPA) Not Scaling Pods

Problem Statement

The Horizontal Pod Autoscaler (HPA) is enabled, but the number of replicas is not increasing even when CPU or memory utilization is high.

What Needs to Be Analyzed

- Check if the HPA is correctly applied to the deployment.
- Verify that metrics are being collected properly.
- Ensure the resource requests/limits allow scaling.

How to Resolve Step by Step

1. **Check HPA Status:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Example output:

```
Unset
NAME       REFERENCE              TARGETS   MINPODS   MAXPODS
REPLICAS

my-app-hpa  Deployment/my-app      50%/80%  2         10        2
```

   - If `TARGETS` is not increasing, HPA is not detecting load.
2. **Check HPA Events for Errors:**

```
Unset
kubectl describe hpa <hpa-name> -n <namespace>
```

   - Look for errors like `unable to fetch metrics`.
3. **Verify Metrics Server Is Running:**

```
Unset
kubectl get pods -n kube-system | grep metrics-server
```

- If missing, install it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

4. **Check Pod Resource Requests to Ensure They Are Defined:**

```
Unset
resources:
  requests:
    cpu: "200m"
    memory: "256Mi"
```

- HPA does not work without `requests`.

5. **Check if the Right Metric Type is Used:**
   - HPA supports CPU, memory, and custom metrics. Check with:

```
Unset
kubectl get hpa -o yaml
```

6. **Manually Scale the Deployment to See if Scaling Works:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

- If manual scaling works but HPA does not, the issue is with metrics collection.

Skills Required to Resolve This Issue

- Kubernetes Horizontal Pod Autoscaler (HPA).
- Metrics collection with Metrics Server or Prometheus.
- Kubernetes resource management.

Conclusion

HPA issues are often caused by missing resource requests, an inactive metrics server, or incorrect metric thresholds. Ensuring proper configurations and monitoring tools will resolve scaling failures.

---

32. Kubernetes Cluster Autoscaler Not Scaling Nodes

Problem Statement

The Cluster Autoscaler is enabled, but it is not adding new nodes when pods fail to schedule due to resource shortages.

What Needs to Be Analyzed

- Check if the Cluster Autoscaler is installed and running.
- Verify the current autoscaler limits and configurations.
- Analyze why new nodes are not being added.

How to Resolve Step by Step

1. **Check If Cluster Autoscaler Is Running:**

```
Unset
kubectl get pods -n kube-system | grep cluster-autoscaler
```

   - If missing, install it for your cloud provider.
2. **Check Autoscaler Logs for Errors:**

```
Unset
kubectl logs -n kube-system -l app=cluster-autoscaler
```

   - Look for messages like `no eligible nodes found for scale-up`.
3. **Check If Node Pool Has Scaling Enabled:**
   - GKE:

```
Unset
gcloud container clusters describe <cluster-name>
```

   - AWS EKS:

```
Unset
eksctl get nodegroup --cluster=<cluster-name>
```

4. **Increase Node Pool Limits If Needed:**
    - GKE:

```
Unset
gcloud container clusters update <cluster-name> --enable-autoscaling
--min-nodes=1 --max-nodes=10
```

    - AWS EKS:

```
Unset
eksctl scale nodegroup --cluster=<cluster-name> --name=<nodegroup-name>
--nodes=5
```

5. **Check if Pod Disruption Budgets (PDBs) Are Preventing Scale-Down:**

```
Unset
kubectl get pdb -A
```

    - If too restrictive, adjust `minAvailable`.
6. **Manually Increase Node Count to Verify Scaling Works:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes Cluster Autoscaler.
- Cloud provider-specific autoscaler configurations.
- Resource management in Kubernetes clusters.

Conclusion

Cluster Autoscaler issues are usually caused by misconfigured limits, node pool restrictions, or PDB constraints. Ensuring proper configuration will allow nodes to scale dynamically.

33. Kubernetes Pods Not Scaling Due to Resource Quotas

Problem Statement

When attempting to scale pods, new replicas are stuck in `Pending` state due to resource quota limits set at the namespace level.

What Needs to Be Analyzed

- Check if there are resource quotas (`cpu`, `memory`, or `pods`) restricting pod scaling.
- Verify namespace-level quota usage and availability.
- Ensure requests and limits in pod specifications align with quota settings.

How to Resolve Step by Step

1. **Check If a Resource Quota Is Applied to the Namespace:**

```
Unset
kubectl get resourcequota -n <namespace>
```

   - Example output:

```
Unset
NAME                 CPU REQUESTS   CPU LIMITS   MEMORY REQUESTS
MEMORY LIMITS   PODS

namespace-quota      2              4            4Gi                8Gi
10
```

   - If the `PODS` limit is reached, no more pods can be scheduled.
2. **Check Current Quota Usage:**

```
Unset
kubectl describe resourcequota <quota-name> -n <namespace>
```

   - Look for values like `Used: 10` under `Pods`, meaning the namespace has hit its pod limit.
3. **Check If the New Pod's Resource Requests Exceed Quota:**

```
Unset
kubectl get pods -n <namespace> -o
jsonpath='{.items[*].spec.containers[*].resources.requests}'
```

- Ensure that the sum of all pod requests does not exceed the allowed quota.

4. **Increase the Resource Quota If Needed:**
   - Edit and update the quota:

```
Unset
apiVersion: v1

kind: ResourceQuota

metadata:

  name: namespace-quota

  namespace: <namespace>

spec:

  hard:

    pods: "20"  # Increase this value

    requests.cpu: "4"

    requests.memory: "8Gi"
```

   - Apply the changes:

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

5. **Delete Unused Pods to Free Up Resources:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

6. **Verify If Pods Can Now Scale:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes resource quotas and limit ranges.
- Pod scheduling and resource management.
- Kubernetes YAML configuration and API objects.

Conclusion

Resource quotas are essential for controlling resource consumption but can block scaling if limits are too low. Adjusting quota settings appropriately ensures smooth pod scaling.

---

34. Kubernetes Pods Not Scaling Due to Node Affinity and Taints

Problem Statement

Pods are not scaling because they are restricted to specific nodes due to affinity rules or node taints that prevent scheduling.

What Needs to Be Analyzed

- Check if node affinity rules are restricting pod placement.
- Verify if node taints are preventing new pods from being scheduled.
- Ensure that pod tolerations align with node taints.

How to Resolve Step by Step

1. **Check Node Affinity Rules in the Pod Specification:**

```
Unset
kubectl get pod <pod-name> -o yaml | grep -A5 affinity
```

   - Example output:

```
Unset
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: "node-role.kubernetes.io/worker"
              operator: In
              values:
                - "true"
```

   - Ensure the node labels match this affinity requirement.
2. **Check Node Labels to See If They Match the Affinity Requirements:**

```
Unset
kubectl get nodes --show-labels
```

- If necessary, add missing labels:

```
Unset
kubectl label nodes <node-name> node-role.kubernetes.io/worker=true
```

3. **Check for Node Taints That Might Be Preventing Scheduling:**

```
Unset
kubectl describe node <node-name> | grep Taints
```

- Example output:

```
Unset
Taints: node-role.kubernetes.io/master:NoSchedule
```

- This means the node does not allow scheduling of regular pods.

4. **Allow Pods to Be Scheduled by Adding a Toleration:**

- Update the deployment YAML:

```
Unset
spec:
  tolerations:
  - key: "node-role.kubernetes.io/master"
    operator: "Exists"
    effect: "NoSchedule"
```

- Apply the changes:

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

5. **Check if Pods Can Now Scale:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes affinity and anti-affinity rules.
- Node taints and tolerations.
- Pod scheduling and cluster management.

Conclusion

Affinity rules and node taints are useful for controlling workload placement but can block scaling if not properly configured. Adjusting affinity rules or adding tolerations can resolve these issues.

35. Kubernetes Pods Not Scaling Due to Insufficient Disk Space on Nodes

Problem Statement

Pods fail to scale because nodes do not have enough disk space, causing scheduling failures.

What Needs to Be Analyzed

- Check if nodes are running out of disk space.
- Verify `Evicted` or `Failed` pod status due to `disk pressure`.
- Ensure node storage is sufficient for pod scaling.

How to Resolve Step by Step

1. **Check Node Disk Usage:**

```
Unset
kubectl describe node <node-name> | grep -i "disk"
```

   - Look for messages like `NodeHasDiskPressure`.
2. **Check If Any Pods Are Evicted Due to Disk Pressure:**

```
Unset
kubectl get pods -A | grep Evicted
```

3. **Clear Unused Docker/Kubelet Storage:**

```
Unset
docker system prune -a

sudo journalctl --vacuum-time=2d
```

4. **Increase Node Disk Size (Cloud-Specific Steps):**

   - **AWS EKS:**

```
Unset
aws ec2 modify-volume --volume-id <volume-id> --size 100
```

- **GKE:**

```
Unset
gcloud compute disks resize <disk-name> --size=100GB
```

5. **Drain and Restart the Node if Disk Pressure Persists:**

```
Unset
kubectl drain <node-name> --ignore-daemonsets

kubectl uncordon <node-name>
```

6. **Check If Pods Can Now Scale:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes node management.
- Cloud provider storage management.
- Linux disk management and cleanup.

Conclusion

Scaling issues due to disk pressure can be resolved by clearing unused storage, increasing node disk size, or adding more nodes with sufficient disk space.

---

36. Kubernetes Pods Not Scaling Due to API Server Rate Limits

Problem Statement

Pods fail to scale because the Kubernetes API server is throttling requests due to exceeding rate limits, causing delays in auto-scaling actions.

What Needs to Be Analyzed

- Check API server logs for throttling messages.
- Monitor the number of requests hitting the API server.
- Verify Horizontal Pod Autoscaler (HPA) and cluster autoscaler logs for scaling failures.

How to Resolve Step by Step

1. **Check API Server Logs for Throttling Messages:**

```
Unset
kubectl logs -n kube-system kube-apiserver-<node-name> | grep -i
throttle
```

   - Example output:

```
Unset
I1017 12:45:10.123456    1 request.go:123] Throttling request due to
client-side rate limiting
```

2. **Monitor API Server Request Rate:**

```
Unset
kubectl get --raw /metrics | grep request
```

   - Look for high request rates from controllers, applications, or kubelets.
3. **Check HPA Logs for Scaling Failures:**

```
Unset
kubectl describe hpa <hpa-name>
```

- Look for messages like:

```
Unset
Scaling delayed due to API server throttling.
```

4. **Increase API Server Rate Limits (For Self-Managed Clusters):**
   - Modify the `kube-apiserver` startup configuration:

```
Unset
--max-requests-inflight=1000

--max-mutating-requests-inflight=500
```

   - Restart the API server:

```
Unset
systemctl restart kube-apiserver
```

5. **Reduce API Server Load by Optimizing Requests:**
   - Use **client-side caching** instead of frequent API requests.
   - Optimize **controller reconciliation loops**.
   - Limit **unnecessary polling** in applications.
6. **Check If Pods Can Now Scale:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=10
```

Skills Required to Resolve This Issue

- Kubernetes API server monitoring and tuning.
- HPA and cluster autoscaler troubleshooting.
- Application and controller optimization.

Conclusion

API server rate limits can block scaling when request rates exceed thresholds. Optimizing API requests and adjusting rate limits can help restore normal autoscaling behavior.

37. Kubernetes Pods Not Scaling Due to Misconfigured Horizontal Pod Autoscaler (HPA)

Problem Statement

HPA is not scaling pods up or down as expected due to incorrect configurations or missing metrics.

What Needs to Be Analyzed

- Verify HPA configuration settings.
- Check if CPU or memory utilization metrics are available.
- Ensure HPA has the correct API version and settings.

How to Resolve Step by Step

1. **Check the Current HPA Configuration:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Example output:

```
Unset
NAME          REFERENCE            TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-hpa        Deployment/my-app    0%/50%    2         10        2          10m
```

2. **Describe HPA for Detailed Status:**

```
Unset
kubectl describe hpa <hpa-name>
```

   - Look for messages like:

```
Unset
Metrics not available
```

3. **Check If Metrics Server Is Running:**

```
Unset
kubectl top pods
```

- If it fails with:

```
Unset
Error from server (NotFound): No metrics available
```

- Restart the metrics server:

```
Unset
kubectl delete -n kube-system deployment metrics-server

kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

4. **Update HPA to Use Correct Metrics:**

```
Unset
apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

  name: my-hpa

spec:

  scaleTargetRef:

    apiVersion: apps/v1
```

```
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- Apply changes:

```
Unset
kubectl apply -f hpa.yaml
```

5. **Check If Pods Can Now Scale:**

```
Unset
kubectl get hpa -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes HPA troubleshooting.
- Metrics server debugging.
- YAML configurations for autoscaling.

Conclusion

Misconfigured HPAs can prevent pods from scaling. Ensuring the correct API version, resource metrics, and proper configurations fixes the issue.

---

38. Kubernetes Pods Not Scaling Due to Insufficient Network Bandwidth

Problem Statement

Pods fail to scale because network congestion or bandwidth limits are restricting new pod communication.

What Needs to Be Analyzed

- Check if nodes are experiencing high network utilization.
- Verify pod-to-pod and pod-to-service communication.
- Ensure CNI plugins and network policies are not causing bottlenecks.

How to Resolve Step by Step

1. **Check Network Bandwidth Usage on Nodes:**

```
Unset
kubectl top nodes
```

   - Look for high network usage on any node.
2. **Check Pod-to-Pod Connectivity:**

```
Unset
kubectl exec -it <pod-name> -- ping <target-pod-ip>
```

   - If packets are lost, investigate network policies.
3. **Check If Network Policies Are Restricting Traffic:**

```
Unset
kubectl get networkpolicy -n <namespace>
```

4. **Increase Node Network Bandwidth (Cloud-Specific Steps):**
   - **AWS EKS:**

```
Unset
aws ec2 modify-instance-attribute --instance-id <instance-id>
--network-interface <new-bandwidth>
```

- **GKE:**

```
Unset
gcloud compute instances set-machine-type <instance-name>
--machine-type=n1-standard-8
```

5. **Optimize CNI Plugin Configuration:**
   - If using Calico, increase pod networking limits:

```
Unset
spec:
  ipPool:
    cidr: 192.168.0.0/16
    blockSize: 26
```

   - Apply changes:

```
Unset
kubectl apply -f calico-config.yaml
```

6. **Restart Affected Pods:**

```
Unset
kubectl rollout restart deployment <deployment-name>
```

Skills Required to Resolve This Issue

- Kubernetes networking and CNI plugins.
- Network policy debugging.
- Cloud provider network management.

Conclusion

Network bandwidth issues can prevent pods from scaling due to congestion or policy restrictions. Monitoring network traffic and adjusting configurations ensures proper scaling.

39. Kubernetes Pods Not Scaling Due to Insufficient Persistent Volume (PV) Storage

Problem Statement

Pods are failing to scale because they are unable to attach a Persistent Volume (PV) due to storage exhaustion or misconfigurations.

What Needs to Be Analyzed

- Check if PVs are exhausted or in `Pending` state.
- Verify storage class configurations.
- Check logs for volume attachment failures.

How to Resolve Step by Step

1. **Check PV Status:**

```
Unset
kubectl get pv
```

   - Example output:

```
Unset
NAME           CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM

pvc-12345      10Gi       RWO            Retain           Bound
default/data-app

pvc-67890      10Gi       RWO            Retain           Pending
default/data-app2
```

   - If the volume is `Pending`, it may not have enough storage or the correct storage class.
2. **Check PVC (Persistent Volume Claim) Status:**

```
Unset
kubectl get pvc -n <namespace>
```

   - Look for `Pending` claims, which indicate the PV is not properly bound.
3. **Check Logs for Volume Attachment Errors:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

- Look for messages like:

```
Unset
Unable to attach or mount volumes: timed out waiting for the condition
```

4. **Verify Storage Class Configuration:**

```
Unset
kubectl get sc
```

- If a custom storage class is used, check if it supports dynamic provisioning:

```
Unset
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: my-storage
provisioner: kubernetes.io/aws-ebs
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Retain
```

- Apply the updated storage class if needed:

```
Unset
kubectl apply -f storage-class.yaml
```

5. **Expand Storage If Required (For Dynamic Storage Providers):**
    - Edit the PVC and increase the requested storage:

```
Unset
spec:
  resources:
    requests:
      storage: 20Gi
```

    - Apply the changes:

```
Unset
kubectl apply -f pvc.yaml
```

6. **Restart the Affected Pods:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

    - The pod should now start with the correct volume.

Skills Required to Resolve This Issue

- Kubernetes storage management.
- Persistent Volume and Persistent Volume Claim debugging.
- Understanding of storage classes in Kubernetes.

Conclusion

Scaling failures due to PV issues often result from storage exhaustion or incorrect configurations. Ensuring proper binding, storage class configuration, and dynamic provisioning allows seamless pod scaling.

40. Kubernetes Pods Not Scaling Due to Node Taints and Tolerations Misconfiguration

Problem Statement

Pods are not scaling because they cannot be scheduled on nodes due to taints preventing their placement.

What Needs to Be Analyzed

- Check if nodes have taints that block scheduling.
- Verify if pods have the necessary tolerations.
- Check if the scheduler is attempting to place pods on the tainted nodes.

How to Resolve Step by Step

1. **Check for Node Taints:**

```
Unset
kubectl get nodes -o
custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
```

   - Example output:

```
Unset
NAME        TAINTS

node-1      key=value:NoSchedule

node-2      <none>
```

2. **Check If Pods Have Tolerations:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for tolerations under the `spec` section.
3. **Add a Toleration to Allow Pods to Run on Tainted Nodes:**
   - Edit the pod definition:

```
Unset
spec:
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

- Apply the updated configuration:

```
Unset
kubectl apply -f pod.yaml
```

4. **Remove the Taint from a Node (If Not Needed):**

```
Unset
kubectl taint nodes node-1 key=value:NoSchedule-
```

5. **Restart the Affected Pods to Trigger Rescheduling:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

6. **Check If Pods Are Now Scaling Properly:**

```
Unset
kubectl get pods -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes node taints and tolerations.
- Pod scheduling and debugging.
- Cluster-wide node management.

Conclusion

Pods may fail to scale if nodes are tainted without proper tolerations. Ensuring that pods have the correct tolerations or removing unnecessary taints allows pods to scale correctly.

41. Kubernetes Pods Not Scaling Due to CPU/Memory Resource Limits

Problem Statement

Pods fail to scale because resource requests and limits are too restrictive, preventing new pods from being scheduled.

What Needs to Be Analyzed

- Check if CPU or memory limits are preventing new pod scheduling.
- Analyze resource allocation in the cluster.
- Verify if Horizontal Pod Autoscaler (HPA) is constrained by resource limits.

How to Resolve Step by Step

1. **Check Resource Requests and Limits in the Deployment:**

```
Unset
kubectl get deployment <deployment-name> -n <namespace> -o yaml | grep -A 10 "resources:"
```

   - Example output:

```
Unset
resources:
  requests:
    cpu: "500m"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "512Mi"
```

2. **Check Node Resource Utilization:**

```
Unset
kubectl top nodes
```

- Example output:

```
Unset
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
node-1        1900m        95%    3800Mi          90%
node-2        800m         40%    2000Mi          50%
```

- If CPU or memory is above 90%, new pods may not be scheduled.

3. **Check HPA (If Used) to See if Scaling is Limited by Resource Usage:**

```
Unset
kubectl get hpa -n <namespace>
```

- Example output:

```
Unset
NAME            REFERENCE                  TARGETS    MINPODS
MAXPODS   REPLICAS   AGE
my-app-hpa      Deployment/my-app          80%/75%    2         5
2         10m
```

- If the `TARGETS` value is below the threshold, HPA will not scale.

4. **Increase Resource Limits (If Needed):**

- Edit the deployment configuration:

```
Unset
spec:
  containers:
```

```
  - name: my-app
    resources:
      requests:
        cpu: "250m"
        memory: "512Mi"
      limits:
        cpu: "2"
        memory: "1Gi"
```

- Apply the changes:

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

5. **Manually Scale Up the Deployment (If Immediate Scaling is Needed):**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

6. **Restart the Pods to Apply Changes:**

```
Unset
kubectl delete pod -l app=<app-name> -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes resource management.
- Horizontal Pod Autoscaler (HPA) troubleshooting.

- Cluster resource analysis.

Conclusion

Restrictive CPU and memory limits can prevent pods from scaling. Adjusting requests and limits or adding more resources to the cluster ensures proper autoscaling.

42. Kubernetes Pods Not Scaling Due to Insufficient Cluster Nodes

Problem Statement

Pods fail to scale because the cluster does not have enough nodes to accommodate new replicas.

What Needs to Be Analyzed

- Check node availability and resource capacity.
- Verify if the cluster autoscaler is enabled (if applicable).
- Check scheduler logs for pod placement failures.

How to Resolve Step by Step

1. **Check Node Count and Availability:**

```
Unset
kubectl get nodes
```

   - Example output:

```
Unset
NAME          STATUS   ROLES    AGE    VERSION
node-1        Ready    worker   30d    v1.26.0
node-2        Ready    worker   30d    v1.26.0
node-3        NotReady worker   30d    v1.26.0
```

   - If some nodes are `NotReady`, they may need debugging.
2. **Check Node Resource Utilization:**

```
Unset
kubectl top nodes
```

   - If CPU or memory usage is too high, new pods cannot be scheduled.
3. **Check Pending Pods for Scheduling Issues:**

```
Unset
kubectl get pods -n <namespace> | grep Pending
```

4. **Describe a Pending Pod to See Scheduling Failures:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for messages like:

```
Unset
0/3 nodes are available: 3 Insufficient CPU.
```

5. **Enable Cluster Autoscaler (If Using a Managed Kubernetes Service):**
   - For **Google Kubernetes Engine (GKE):**

```
Unset
gcloud container clusters update my-cluster --enable-autoscaling
--min-nodes=2 --max-nodes=10 --zone=us-central1-a
```

   - For **Amazon EKS:**

```
Unset
eksctl create nodegroup --cluster my-cluster --name autoscale-nodes
--nodes-min 2 --nodes-max 10
```

6. **Manually Add More Nodes (For Self-Managed Clusters):**
   - Provision additional worker nodes with `kubeadm` or cloud provider tools.
7. **Restart the Affected Pods to Trigger Scheduling:**

```
Unset
kubectl delete pod <pod-name> -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes cluster scaling.
- Node resource management.
- Kubernetes Cluster Autoscaler configuration.

Conclusion

When cluster resources are exhausted, pods cannot scale. Enabling autoscaling or manually adding nodes ensures pods are scheduled efficiently.

43. Kubernetes Pods Not Scaling Due to Pod Disruption Budget (PDB) Constraints

Problem Statement

Pods are not scaling up or down because a **Pod Disruption Budget (PDB)** is restricting the number of pods that can be evicted or rescheduled.

What Needs to Be Analyzed

- Check if a **Pod Disruption Budget (PDB)** is set for the deployment.
- Verify the `minAvailable` or `maxUnavailable` values in the PDB.
- Analyze if the PDB is preventing scaling operations.

How to Resolve Step by Step

1. **Check Existing Pod Disruption Budgets:**

```
Unset
kubectl get pdb -n <namespace>
```

   - Example output:

```
Unset
NAME          MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS
AGE

my-app-pdb    2               N/A               0
10m
```

   - If `ALLOWED DISRUPTIONS` is `0`, Kubernetes will not allow the pod to be terminated or rescheduled.
2. **Describe the PDB for More Details:**

```
Unset
kubectl describe pdb <pdb-name> -n <namespace>
```

   - Look for messages like:

```
Unset
No disruptions allowed because minAvailable is set to 2 and only 2 pods exist.
```

3. **Modify or Remove the PDB (If Necessary):**
   - Edit the PDB configuration:

```
Unset
apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

  name: my-app-pdb

spec:

  minAvailable: 50%   # Adjust based on scaling needs
```

   - Apply the updated configuration:

```
Unset
kubectl apply -f pdb.yaml
```

4. **Manually Scale Up the Deployment:**

```
Unset
kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>
```

5. **Verify Scaling is Successful:**

```
Unset
kubectl get pods -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes Pod Disruption Budgets (PDB).
- YAML configuration and resource management.
- Scaling strategies in Kubernetes.

Conclusion

A restrictive **Pod Disruption Budget (PDB)** can prevent pods from scaling. Adjusting `minAvailable` or `maxUnavailable` ensures that pods can scale without violating availability constraints.

---

44. Kubernetes Pods Not Scaling Due to Insufficient IP Addresses

Problem Statement

Pods fail to scale because there are not enough **IP addresses** available in the cluster's networking configuration.

What Needs to Be Analyzed

- Check if the cluster is running out of IP addresses.
- Verify the **CIDR range** assigned to the cluster and node pools.
- Inspect the **CNI plugin** (Calico, Cilium, Flannel, etc.) for IP exhaustion.

How to Resolve Step by Step

1. **Check the Cluster's IP Allocation Status:**

```
Unset
kubectl get nodes -o wide
```

   - If many nodes are marked `NotReady`, it may be due to networking issues.
2. **Check Available IPs in the Cluster's CIDR Range (For GKE):**

```
Unset
gcloud container clusters describe my-cluster --zone=us-central1-a |
grep clusterIpv4Cidr
```

   - Example output:

```
Unset
clusterIpv4Cidr: 10.96.0.0/14
```

   - If all IPs are used, new pods cannot be assigned an IP.
3. **Check IP Usage on Nodes (For AWS EKS):**

```
Unset
aws eks describe-cluster --name my-cluster | grep -i "cidr"
```

- Look for messages like `Insufficient IPs for new pods`.

4. **Increase the IP Range for the Cluster (If Possible):**
   - **For GKE:**

```
Unset
gcloud container clusters update my-cluster --enable-ip-alias
```

   - **For AWS EKS:**

```
Unset
aws ec2 modify-subnet-attribute --subnet-id <subnet-id>
--map-public-ip-on-launch
```

5. **Restart the Affected Pods:**

```
Unset
kubectl delete pod -l app=<app-name> -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes networking concepts.
- Cloud provider networking (GCP, AWS, Azure).
- Cluster networking plugins (Calico, Flannel, etc.).

Conclusion

If the cluster runs out of IP addresses, new pods cannot be created. Expanding the CIDR range or enabling **IP aliasing** allows more pods to be scheduled.

---

45. Kubernetes Autoscaler Not Scaling Pods Due to Insufficient Node Resources

Problem Statement

The **Horizontal Pod Autoscaler (HPA)** or **Cluster Autoscaler** is not able to scale pods because there are **not enough node resources (CPU, memory, or ephemeral storage)** available in the cluster.

What Needs to Be Analyzed

- Check if the cluster has sufficient node capacity to handle new pods.
- Verify if the **Cluster Autoscaler** is enabled and functioning correctly.
- Inspect CPU and memory requests/limits set on the pods.
- Identify if node taints or scheduling constraints are preventing new pods from being scheduled.

How to Resolve Step by Step

1. **Check Current Node Resource Availability:**

```
Unset
kubectl describe nodes | grep -A 10 "Allocatable"
```

    - Look for available **CPU, memory, and ephemeral storage** in the output.
    - If a node is running out of resources, new pods cannot be scheduled.
2. **Check if Pods Are in a Pending State Due to Resource Constraints:**

```
Unset
kubectl get pods --all-namespaces | grep Pending
```

    - If pods are pending for a long time, they might be waiting for resources.
3. **Describe a Pending Pod to Identify Resource Issues:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

    - Look for messages like:

```
Unset
0/5 nodes are available: 5 Insufficient cpu.
```

4. **Verify the HPA and Cluster Autoscaler Logs:**
   - Check the **HPA status:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Describe the HPA for more details:

```
Unset
kubectl describe hpa <hpa-name> -n <namespace>
```

   - Check if Cluster Autoscaler is running:

```
Unset
kubectl get pods -n kube-system | grep cluster-autoscaler
```

5. **Manually Scale the Cluster (If Needed):**
   - **For GKE:**

```
Unset
gcloud container clusters resize my-cluster --num-nodes=5
--zone=us-central1-a
```

   - **For AWS EKS:**

```
Unset
eksctl scale nodegroup --cluster=my-cluster --nodes=5
```

6. **Adjust CPU and Memory Requests (If Necessary):**
    - Open the deployment YAML file:

```
Unset
resources:
  requests:
    cpu: "200m"
    memory: "256Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

    - Apply the updated configuration:

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

7. **Verify Scaling Works After Fixing Issues:**

```
Unset
kubectl get pods -o wide -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes Autoscaling (HPA, Cluster Autoscaler).
- Resource management in Kubernetes.
- Cloud provider CLI tools for scaling nodes.

Conclusion

If the cluster lacks sufficient CPU, memory, or other resources, **Cluster Autoscaler** might fail to add new nodes, and **HPA** will not scale pods as expected. Ensuring adequate node capacity and correctly configuring resource requests/limits helps in efficient scaling.

46. Kubernetes Not Scaling Due to Pending Persistent Volume Claims (PVCs)

Problem Statement

Pods are not scaling because the required **Persistent Volume Claims (PVCs)** are stuck in a `Pending` state, preventing new pods from being created.

What Needs to Be Analyzed

- Check if **Persistent Volumes (PVs)** are available to fulfill the PVC requests.
- Verify if the **StorageClass** is correctly set and supports dynamic provisioning.
- Inspect if there are any cloud provider limitations on storage allocation.

How to Resolve Step by Step

1. **Check PVC Status:**

```
Unset
kubectl get pvc -n <namespace>
```

   - Example output:

```
Unset
NAME           STATUS    VOLUME   CAPACITY   ACCESS MODES
STORAGECLASS   AGE

my-app-pvc     Pending   -        -          -             standard
10m
```

   - If the PVC is `Pending`, no storage is available.
2. **Describe the PVC to Identify Issues:**

```
Unset
kubectl describe pvc <pvc-name> -n <namespace>
```

   - Look for errors like:

```
Unset
Waiting for a volume to be created, either manually or by the storage class.
```

3. **Check Available Persistent Volumes (PVs):**

```
Unset
kubectl get pv
```

   - Ensure there is a PV with the correct **StorageClass** and enough capacity.

4. **Verify StorageClass Supports Dynamic Provisioning:**

```
Unset
kubectl get storageclass
```

   - Look for a **provisioner** (e.g., `kubernetes.io/aws-ebs` for AWS).
   - If missing, create a default StorageClass:

```
Unset
apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: standard

provisioner: kubernetes.io/aws-ebs

volumeBindingMode: WaitForFirstConsumer
```

   - Apply the StorageClass:

```
Unset
kubectl apply -f storageclass.yaml
```

5. **Manually Create a Persistent Volume (If Needed):**
    - If dynamic provisioning is not enabled, create a **PV** manually:

```
Unset
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data"
```

    - Apply the PV:

```
Unset
kubectl apply -f pv.yaml
```

6. **Delete and Restart Stuck Pods:**

```
Unset
kubectl delete pod -l app=<app-name> -n <namespace>

kubectl apply -f deployment.yaml
```

7. **Verify PVC is Now Bound to a PV:**

```
Unset
kubectl get pvc -n <namespace>
```

   - The `STATUS` should change from `Pending` to `Bound`.

Skills Required to Resolve This Issue

- Kubernetes Persistent Volume (PV) and Persistent Volume Claim (PVC).
- Storage provisioning and StorageClasses.
- Cloud provider storage configuration (EBS, Azure Disk, GCE Persistent Disks).

Conclusion

If a pod's PVC is stuck in `Pending`, it prevents the pod from starting. Ensuring the **StorageClass** supports dynamic provisioning or manually binding a PV can resolve scaling issues.

---

47. Kubernetes Not Scaling Due to Resource Quotas or Limit Ranges

Problem Statement

Pods are not scaling because the **namespace has a resource quota or limit range**, restricting CPU, memory, or the number of pods that can be created.

What Needs to Be Analyzed

- Check if a **ResourceQuota** is set on the namespace.
- Check if a **LimitRange** is restricting pod resources.
- Identify if the deployment is exceeding the allowed quota.

How to Resolve Step by Step

1. **Check Resource Quotas in the Namespace:**

```
Unset
kubectl get resourcequota -n <namespace>
```

   - Example output:

```
Unset
NAME             CPU(REQUEST)   MEMORY(REQUEST)   PODS   AGE
namespace-quota 2              4Gi               10     15d
```

   - If the quota for **pods** is already used up, new pods won't be scheduled.
2. **Describe the ResourceQuota for More Details:**

```
Unset
kubectl describe resourcequota namespace-quota -n <namespace>
```

   - Look for errors like:

```
Unset
Exceeded quota: requested 1, but 10 pods are already running.
```

3. **Check LimitRange in the Namespace:**

```
Unset
kubectl get limitrange -n <namespace>
```

   - Example output:

```
Unset
NAME              TYPE           CPU REQUEST   MEMORY REQUEST   AGE
namespace-limits Container     500m          512Mi            10d
```

   - If a pod requests more resources than allowed, it won't be scheduled.

4. **Describe the LimitRange for More Details:**

```
Unset
kubectl describe limitrange namespace-limits -n <namespace>
```

   - Look for errors like:

```
Unset
Pod exceeds CPU limit of 500m.
```

5. **Increase or Adjust the Resource Quota (If Necessary):**

   - Edit the quota to allow more pods or resources:

```
Unset
apiVersion: v1
kind: ResourceQuota
metadata:
```

```
  name: namespace-quota
  namespace: <namespace>
spec:
  hard:
    pods: "20"
    requests.cpu: "4"
    requests.memory: 8Gi
```

- Apply the new quota:

```
Unset
kubectl apply -f resourcequota.yaml
```

6. **Modify the LimitRange to Allow More Resources (If Necessary):**
   - Edit the limit range:

```
Unset
apiVersion: v1
kind: LimitRange
metadata:
  name: namespace-limits
  namespace: <namespace>
spec:
  limits:
    - type: Container
```

```
    max:
      cpu: "2"
      memory: 2Gi
    min:
      cpu: "250m"
      memory: 256Mi
```

- Apply the updated configuration:

```
Unset
kubectl apply -f limitrange.yaml
```

7. **Verify Scaling Works After Fixing the Limits:**

```
Unset
kubectl get pods -o wide -n <namespace>
```

Skills Required to Resolve This Issue

- Kubernetes ResourceQuota and LimitRange concepts.
- YAML configuration for quota adjustments.
- Understanding CPU and memory requests and limits in Kubernetes.

Conclusion

If a namespace has strict **ResourceQuotas** or **LimitRanges**, it can prevent autoscaling from working properly. Adjusting these settings appropriately allows scaling to function as expected.

---

48. Kubernetes Not Scaling Due to Failed Liveness or Readiness Probes

Problem Statement

Pods are not scaling because **liveness or readiness probes are failing**, causing new instances to be marked as unhealthy and preventing Kubernetes from scheduling more pods.

What Needs to Be Analyzed

- Check if the liveness/readiness probes are correctly configured.
- Inspect the pod logs to see why the probe is failing.
- Verify if the application takes longer to start and needs a **startupProbe** instead.

How to Resolve Step by Step

1. **Check Pod Status for Failing Probes:**

```
Unset
kubectl get pods -n <namespace>
```

   - Example output:

```
Unset
NAME           READY   STATUS             RESTARTS   AGE
my-app-abc     0/1     CrashLoopBackOff   5          10m
```

2. **Describe the Pod to Find Probe Failures:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for events like:

```
Unset
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with
status code 500
```

3. **Check the Application Logs for More Details:**

```
Unset
kubectl logs <pod-name> -n <namespace>
```

   - If the application takes too long to start, a **startupProbe** might be needed.

4. **Verify and Fix the Probe Configuration in Deployment YAML:**

```
Unset
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
  failureThreshold: 3


readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 3
  failureThreshold: 3
```

   - If the app takes longer to start, use a **startupProbe**:

```
Unset
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 5
  failureThreshold: 10
```

5. **Apply the Updated Deployment:**

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

6. **Restart the Affected Pods:**

```
Unset
kubectl rollout restart deployment <deployment-name> -n <namespace>
```

7. **Monitor the Pod Status to Confirm Fix:**

```
Unset
kubectl get pods -n <namespace>
```

   - The `STATUS` should now be `Running`.

Skills Required to Resolve This Issue

- Kubernetes health checks (liveness, readiness, startup probes).
- Application debugging and log analysis.
- YAML configuration for Kubernetes probes.

Conclusion

Failing **liveness** or **readiness probes** can cause Kubernetes to restart or fail to scale pods. Properly configuring the **startupProbe**, **livenessProbe**, and **readinessProbe** ensures that the application runs smoothly and scales correctly.

49. Kubernetes Not Scaling Due to Insufficient Cluster Resources

Problem Statement

Pods are not scaling because the Kubernetes cluster does not have enough **CPU, memory, or node capacity** to schedule new pods.

What Needs to Be Analyzed

- Check **cluster resource availability** (CPU, memory, disk).
- Check if the **nodes are under high utilization** and cannot accommodate new pods.
- Verify **pod resource requests and limits** to ensure they fit within available node resources.
- Check **autoscaler logs** if cluster autoscaling is enabled.

How to Resolve Step by Step

1. **Check Node Status to Identify Resource Shortages:**

```
Unset
kubectl get nodes
```

   - Example output:

```
Unset
NAME          STATUS   ROLES    AGE    VERSION   CPU   MEMORY
node-1        Ready    worker   120d   v1.25     2     8Gi
node-2        Ready    worker   120d   v1.25     2     8Gi
node-3        NotReady worker   100d   v1.25     4     16Gi
```

   - A node with `NotReady` status may indicate resource pressure.
2. **Check Resource Requests and Limits for the Scaling Pods:**

```
Unset
kubectl describe pod <pod-name> -n <namespace>
```

   - Look for high resource requests like:

```
Unset
Requests:
  cpu: 2
  memory: 4Gi
Limits:
  cpu: 4
  memory: 8Gi
```

- If the requested resources exceed available node capacity, the pod won't be scheduled.

3. **Check Events for Scheduling Failures:**

```
Unset
kubectl get events -n <namespace> --sort-by=.metadata.creationTimestamp
```

- Look for errors like:

```
Unset
FailedScheduling: No nodes are available that match resource requests.
```

4. **Check Cluster-Wide Resource Utilization:**

```
Unset
kubectl top nodes
```

- Example output:

```
Unset
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
```

```
node-1       1900m        95%    7.5Gi          95%
node-2       1700m        85%    6.8Gi          85%
node-3       4000m        100%   16Gi           100%
```

- If CPU or memory is at **high usage (above 90%)**, new pods cannot be scheduled.

5. **Scale the Cluster by Adding More Nodes (If Using Cluster Autoscaler):**

```
Unset
kubectl get configmap cluster-autoscaler-status -n kube-system -o yaml
```

- If autoscaler is enabled but nodes are not being added, check logs:

```
Unset
kubectl logs -f deployment/cluster-autoscaler -n kube-system
```

- If needed, **increase the max node limit**:

```
Unset
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler
  namespace: kube-system
data:
  max-nodes: "10"
```

- Apply the changes:

```
Unset
kubectl apply -f cluster-autoscaler.yaml
```

6. **Manually Add More Nodes (For Static Clusters):**
   - **For cloud-based clusters (EKS, AKS, GKE):**

```
Unset
gcloud container clusters resize my-cluster --node-pool default-pool
--num-nodes=5
```

   -

```
Unset
az aks scale --name my-aks-cluster --resource-group my-rg --node-count
5
```

   - **For on-prem clusters:** Add a new worker node manually and join it using `kubeadm join`.
7. **Optimize Resource Requests to Fit Available Resources:**
   - Edit the deployment YAML to reduce resource requests:

```
Unset
resources:
  requests:
    cpu: "500m"
    memory: "512Mi"
  limits:
    cpu: "1"
    memory: "1Gi"
```

   - Apply the changes:

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

8. **Restart the Deployment to Trigger Scaling:**

```
Unset
kubectl rollout restart deployment <deployment-name> -n <namespace>
```

9. **Verify Pods Are Now Scaling Correctly:**

```
Unset
kubectl get pods -n <namespace>
```

   - The `STATUS` should now show `Running`.

Skills Required to Resolve This Issue

- Kubernetes resource management (CPU, memory requests/limits).
- Cluster scaling (manual and autoscaler-based).
- Cloud provider commands for cluster scaling.
- Kubernetes node troubleshooting.

Conclusion

A Kubernetes cluster may fail to scale if there are **insufficient resources** to accommodate new pods. Monitoring node utilization, adjusting resource requests, and enabling cluster autoscaling can help resolve these issues effectively.

---

50. Kubernetes HPA Not Scaling Pods as Expected

Problem Statement

The Horizontal Pod Autoscaler (HPA) is configured, but the application is not scaling as expected. Either:

- Pods are **not increasing** under high load.
- Pods are **not decreasing** when the load reduces.

What Needs to Be Analyzed

- Check if the **HPA is properly configured** and monitoring the correct metric.
- Verify if the **CPU/memory utilization targets** are set correctly.
- Ensure that the **metrics server is running and reporting values**.
- Analyze **HPA event logs** for scaling failures.

How to Resolve Step by Step

1. **Check if HPA is Active and Watching the Correct Deployment:**

```
Unset
kubectl get hpa -n <namespace>
```

   - Example output:

```
Unset
NAME         REFERENCE              TARGETS    MINPODS   MAXPODS
REPLICAS   AGE

my-app-hpa  Deployment/my-app     0% / 80%   2         10        2
5m
```

   - If **TARGETS** is `0%`, the CPU utilization metric is not being reported.
2. **Check if Metrics Server is Running:**

```
Unset
kubectl get deployment metrics-server -n kube-system
```

   - If it is missing, install it:

```
Unset
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/downl
oad/components.yaml
```

3. **Verify CPU/Memory Metrics Are Being Reported:**

```
Unset
kubectl top pods -n <namespace>
```

   - If no output is returned, HPA cannot get pod metrics.

4. **Check HPA Events for Scaling Failures:**

```
Unset
kubectl describe hpa my-app-hpa -n <namespace>
```

   - Look for messages like:

```
Unset
unable to get metrics for resource cpu: no metrics returned
```

5. **Manually Increase Load to Trigger Scaling:**

```
Unset
kubectl run load-generator --image=busybox -- /bin/sh -c "while true;
do wget -q -O- http://my-app-service; done"
```

   - If scaling does not happen, HPA may not be monitoring the right metric.

6. **Ensure Correct Resource Requests Are Set in the Deployment:**

   - Edit the deployment and set CPU requests and limits:

```
Unset
resources:
  requests:
    cpu: "100m"
    memory: "256Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

- Apply the changes:

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

7. **Check the HPA Configuration and Adjust CPU Thresholds if Needed:**

```
Unset
kubectl get hpa my-app-hpa -o yaml -n <namespace>
```

- If CPU target is too high, reduce it:

```
Unset
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
  namespace: default
```

```
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50  # Lower from 80% to 50% if scaling is
slow
```

- Apply the changes:

```
Unset
kubectl apply -f hpa.yaml
```

8. **Manually Scale Up to Test HPA Reaction:**

```
Unset
kubectl scale deployment my-app --replicas=5 -n <namespace>
```

- If HPA does not reduce replicas later, check for scaling limits.

9. **Check for Conflicting Autoscalers (e.g., VPA and HPA Conflicts):**
    - If Vertical Pod Autoscaler (VPA) is also enabled, it can conflict with HPA.

```
Unset
kubectl get vpa -n <namespace>
```

    - If found, **disable VPA for the deployment** to allow HPA to scale correctly.
10. **Restart the Metrics Server if Needed:**

```
Unset
kubectl delete pod -n kube-system -l k8s-app=metrics-server
```

Skills Required to Resolve This Issue

- Kubernetes Horizontal Pod Autoscaler (HPA).
- Kubernetes Metrics Server and resource monitoring.
- YAML configuration for HPA tuning.
- Load testing with Kubernetes.

Conclusion

HPA scaling failures can result from missing CPU/memory metrics, misconfigurations, or conflicts with other autoscalers. Ensuring a **healthy metrics server, correct resource settings, and proper HPA targets** will help resolve scaling issues.

Hope You Learned Many Important Kubernetes Scaling errors in this series.

Feel free to connect with 

https://www.linkedin.com/in/careerbytecode/

Respect our sleepless efforts! 🤝
Before you hit download, show some love reshare this post and tag your friends! 🚀💙

CareerByteCode

Learning Made simple

ALL IN ONE PLATFORM

https://careerbytecode.substack.com

241K Happy learners from 91 Countries

Learning
Training
Usecases
Solutions
Consulting

RealTime Handson Usecases Platform to Launch Your IT Tech Career!

CAREER BYTE CODE
REALTIME PROJECTS PLATFORM

91 COUNTRIES 241k Learners
+32 471 40 89 08
CAREERBYTECODE.SUBSTACK.COM

→ TRAININGS

WE ARE DIFFERENT

At CareerByteCode, we redefine training by focusing on real-world, hands-on experience. Unlike traditional learning methods, we provide step-by-step implementation guides, 500+ real-time use cases, and industry-relevant projects across cutting-edge technologies like AWS, Azure, GCP, DevOps, AI, FullStack Development and more.

Our approach goes beyond theoretical knowledge—we offer expert mentorship, helping learners understand how to study effectively, close career gaps, and gain the practical skills that employers value.

16+
Years of operations

91+
Countries worldwide

241 K Happy clients

Our Usecases Platform
https://careerbytecode.substack.com

Our WebShop
https://careerbytecode.shop

CareerByteCode
Learning Made simple

CareerByteCode
All in One Platform

STAY IN TOUCH WITH US!

Website

Our WebShop https://careerbytecode.shop
Our Usecases Platform https://careerbytecode.substack.com

E-mail
careerbytec@gmail.com

Social Media
@careerbytecode

Phone
+32 471 40 8908

HQ address
Belgium, Europe