CAREER BYTE CODE
REALTIME PROJECTS PLATFORM

91 COUNTRIES 241k Learners
+32 471 40 89 08
CAREERBYTECODE.SUBSTACK.COM

50

Kubernetes Issues Troubleshooting

Pocket Guide

careerbytecode.substack.com
+32 - 471408908

Error 1: CrashLoopBackOff Error in a Pod

1. Problem Statement

A pod is repeatedly crashing and restarting, showing the error status `CrashLoopBackOff` when checking with `kubectl get pods.`

2. What Needs to Be Analyzed

- Check pod logs: `kubectl logs <pod-name> -n <namespace>`
- Check events: `kubectl describe pod <pod-name> -n <namespace>`
- Inspect container exit codes and reasons: `kubectl get pod <pod-name> -o jsonpath='{.status.containerStatuses[0].state}'`

3. How to Resolve Step by Step

Check Logs for Clues

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

1. Look for application errors, missing dependencies, or misconfigurations.

Inspect Events and Status

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

2. Identify issues like OOMKilled, image pull errors, or missing environment variables.
3. **Verify Image and Entry Command**

Check if the container's command is incorrect:

```
kubectl get pod <pod-name> -o jsonpath='{.spec.containers[0].command}'
```

   - If the command is wrong, update the deployment YAML.

4. **Fix Misconfigurations**
   - If the issue is missing environment variables, check `kubectl get deploy <deployment-name> -o yaml` and fix the values.
   - If a dependency is missing, ensure all required services are running.
5. **Check Resource Limits**

If the error is `OOMKilled`, increase memory limits in the deployment:

```
resources:
```

```
  requests:
    memory: "256Mi"
  limits:
    memory: "512Mi"
```

Manually Restart the Pod

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

6. This forces Kubernetes to reschedule it.

Update or Roll Back the Deployment
If an incorrect version is deployed, roll back:

```
kubectl rollout undo deployment <deployment-name> -n <namespace>
```

4. Skills Required to Resolve This Issue

- Familiarity with `kubectl` commands
- Knowledge of containerized applications and logs
- Understanding of Kubernetes resource limits
- Experience with YAML configuration

5. Conclusion

The `CrashLoopBackOff` error usually indicates an issue inside the container, such as an application crash, misconfiguration, or insufficient resources. Analyzing logs, inspecting pod events, and adjusting configurations can help resolve it effectively.

Error 2: ImagePullBackOff Error

1. Problem Statement

When trying to deploy a pod, it fails to start due to an `ImagePullBackOff` error. This indicates that Kubernetes is unable to pull the container image from the specified registry.

2. What Needs to Be Analyzed

- Verify the image name and tag in the deployment YAML.
- Check for network connectivity or permissions issues with the container registry.
- Inspect the error message when running `kubectl describe pod <pod-name>`.
- Ensure the image is available in the registry.
- Check if authentication is required for private registries.

3. How to Resolve Step by Step

Verify Image Name and Tag
Ensure that the image name and tag are correct in the deployment YAML:

```
spec:
  containers:
    - name: my-app
      image: <registry>/<image>:<tag>
```

1. If the tag is incorrect, update it with the correct one.
2. **Check Image Availability in Registry**
   Log in to the container registry and verify if the image exists.
   Example:
   - For Docker Hub: `docker pull <image-name>:<tag>`

For Google Container Registry (GCR):

```
gcloud container images list-tags gcr.io/<project-id>/<image-name>
```

3. **Check for Network Issues**
   Run `kubectl describe pod <pod-name>` and inspect the events. Look for errors like `Failed to pull image`.
   - If there's a DNS issue, ensure that DNS is configured correctly.
   - If you are behind a proxy, configure the proxy settings for Kubernetes to pull the image.

Authenticate for Private Registries
If pulling from a private registry, ensure the correct secrets are set for pulling the image.
Create or configure a secret to allow Kubernetes to authenticate with the private registry:

```
kubectl create secret docker-registry my-registry-secret \
  --docker-server=<registry-server> \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email> \
  --namespace <namespace>
```

Reference this secret in your deployment YAML:

```
spec:
  imagePullSecrets:
    - name: my-registry-secret
```

Retry Pulling the Image
If the image is available and the issue persists, delete the pod to allow Kubernetes to retry pulling the image:

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

4. Skills Required to Resolve This Issue

- Knowledge of Kubernetes deployments and image configurations
- Familiarity with container registries (Docker Hub, GCR, ECR, etc.)
- Experience with Kubernetes secrets for private registry authentication
- Basic understanding of networking and DNS in Kubernetes

5. Conclusion

The `ImagePullBackOff` error occurs when Kubernetes fails to pull the container image due to reasons like incorrect image names, network issues, or authentication problems with private registries. Verifying the image and network configuration, along with proper authentication, should resolve the issue.

Error 3: ResourceQuota Exceeded

1. Problem Statement

A pod fails to schedule due to exceeding the specified `ResourceQuota`. This error typically occurs when the total resource usage exceeds the limits defined for a namespace.

2. What Needs to Be Analyzed

Check the `ResourceQuota` configuration in the namespace:

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

Check the resource requests and limits of the pod:

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

-
- Compare the pod's resource usage with the quota limits.
- Verify other pods running in the same namespace and their resource consumption.

3. How to Resolve Step by Step

View ResourceQuota Configuration
Check the current resource quotas in the namespace:

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

Inspect Pod Resource Requests and Limits
Review the pod's resource requests and limits in the deployment YAML. Ensure they are within the allocated `ResourceQuota`. Example:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "0.5"
  limits:
    memory: "512Mi"
```

```
    cpu: "1"
```

Check for Overuse by Other Pods
List the resources used by other pods in the same namespace:

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

1. **Modify Resource Requests or Limits**

If the pod is overusing resources, adjust the resource requests and limits to fit within the quota:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "0.25"
  limits:
    memory: "256Mi"
    cpu: "0.5"
```

-

Update or Increase the ResourceQuota
If your application requires more resources than the current quota, update the `ResourceQuota` to allow more resources:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-quota
  namespace: <namespace>
spec:
  hard:
```

```
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "6"
    limits.memory: "12Gi"
```

Remove Unnecessary Resources
Delete unused or unnecessary pods or resources to free up quota. For example:

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

Reapply the Deployment
Once resource adjustments are made, redeploy the pod or application:

```
kubectl apply -f <deployment.yaml>
```

4. Skills Required to Resolve This Issue

- Understanding of Kubernetes resource management (requests, limits, quotas)
- Familiarity with namespace resource restrictions
- Ability to analyze pod resource usage and adjust accordingly
- Experience with YAML configurations for resource allocation

5. Conclusion

The `ResourceQuota Exceeded` error happens when a pod tries to exceed the resource allocation for its namespace. By analyzing and adjusting the pod's resource requests/limits or increasing the `ResourceQuota`, the error can be resolved effectively.

Error 4: NoResourcesAvailable Error (Pod Scheduling)

1. Problem Statement

When deploying a pod, Kubernetes fails to schedule it with the error `NoResourcesAvailable`. This occurs when the cluster has insufficient resources (CPU, memory, etc.) to schedule the pod.

2. What Needs to Be Analyzed

- Check the resource requests and limits of the pod.

Inspect the available resources on the nodes:

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

Verify the current resource usage in the cluster:

```
kubectl top nodes
```

- Check if there are other pods consuming too many resources, preventing scheduling.

3. How to Resolve Step by Step

Check Pod Resource Requests
Review the pod's resource requests and limits in the YAML configuration:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "0.5"
  limits:
    memory: "512Mi"
    cpu: "1"
```

Check Node Resources
Run the following command to see resource availability on the nodes:

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

1. This will show the CPU and memory capacity, along with the resources that are already in use.

Check Cluster Resource Usage
Use the `kubectl top nodes` command to see if the cluster is running low on resources.

```
kubectl top nodes
```

2. If resources are tight, consider scaling the cluster by adding more nodes or reducing the resource usage of other pods.
3. **Scale the Cluster**
   If the nodes have insufficient resources, consider adding more nodes to the cluster or upgrading the existing nodes to handle additional pods.

Reduce Resource Requests of the Pod
If possible, adjust the resource requests in the pod definition to lower values that are more likely to fit the available resources:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "0.25"
```

Evict Unnecessary Pods
If there are pods consuming too many resources, consider evicting or scaling down unnecessary pods. For example:

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

4. **Check Affinity and Taints/Tolerations**
   Ensure that the pod has proper affinity or tolerations if it needs to be scheduled on specific nodes. Review the pod configuration to make sure there are no constraints preventing scheduling on available nodes.

Re-deploy the Pod
Once resources are adjusted, re-deploy the pod:

```
kubectl apply -f <deployment.yaml>
```

4. Skills Required to Resolve This Issue

- Understanding of Kubernetes resource management (CPU, memory, limits)
- Familiarity with Kubernetes scheduling and node resource management
- Ability to analyze cluster health and resource consumption
- Knowledge of scaling and node management in Kubernetes

5. Conclusion

The `NoResourcesAvailable` error occurs when there are insufficient resources in the cluster to schedule the pod. By analyzing the pod's resource requirements, checking the cluster's resource usage, and possibly scaling the cluster, this issue can be resolved.

Error 5: CrashLoopBackOff Error

1. Problem Statement

When trying to start a pod, it continuously crashes and enters a `CrashLoopBackOff` state. This error indicates that the pod is failing to start or is restarting repeatedly due to issues in the container or configuration.

2. What Needs to Be Analyzed

Check the pod logs for errors:
`kubectl logs <pod-name> -n <namespace>`

-
- Inspect the exit code of the container to determine the cause of failure.
- Check if the container's entry point or command is correct.
- Examine resource requests/limits for misconfigurations.
- Review health checks (liveness and readiness probes) for misconfiguration.

3. How to Resolve Step by Step

Check Pod Logs
First, view the pod's logs to get detailed information about why the container is failing:

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

1. If the logs show an error, fix the issue that's causing the container to crash.

Check Container Exit Code
Check the exit code of the container to understand why it's failing:

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

2. Common exit codes:
    - `Exit Code 1` often indicates a general error in the container.
    - `Exit Code 137` often means the container was killed due to resource limits.

Inspect Container Entry Point and Command
Verify that the container's entry point or command is set correctly in the YAML configuration.
Example:

```
spec:
  containers:
    - name: my-app
      image: <image-name>
```

```
    command: ["/bin/sh", "-c", "node server.js"]
```

3. If the command is incorrect, update it with the correct startup script or executable.

Check Resource Limits
Ensure the resource requests and limits are not too restrictive. If resources are exhausted, Kubernetes will terminate the container. Modify the resource limits if necessary:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "0.5"
  limits:
    memory: "512Mi"
    cpu: "1"
```

Review Liveness and Readiness Probes
Misconfigured health checks can cause the pod to fail. Verify the liveness and readiness probes in the pod definition:

```
spec:
  containers:
    - name: my-app
      image: <image-name>
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 3
        periodSeconds: 5
      readinessProbe:
```

```
    httpGet:
      path: /readiness
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 5
```

4. Ensure that the probes are configured correctly and that the application is ready to respond.
5. **Check for Dependency Failures**
   If the application depends on other services or databases, verify that those dependencies are up and running. If there are connection issues, the application might keep crashing.

Manually Restart the Pod
After making adjustments, restart the pod to see if it resolves the issue:

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

Increase RestartPolicy (Optional)
If necessary, you can adjust the restart policy in the YAML configuration to provide more time for the container to stabilize:

```
spec:
  restartPolicy: Always
```

4. Skills Required to Resolve This Issue

- Knowledge of Kubernetes pod management and troubleshooting
- Familiarity with container logs and exit codes
- Understanding of health checks and resource management
- Ability to modify YAML configurations for deployment and health checks

5. Conclusion

The `CrashLoopBackOff` error happens when a pod continuously fails to start due to issues with the container's configuration, resources, health checks, or dependencies. Analyzing logs, adjusting configurations, and ensuring proper resource allocation should resolve this issue.

Error 6: ImagePullBackOff Error

1. Problem Statement

The `ImagePullBackOff` error occurs when Kubernetes is unable to pull the specified container image from the container registry. This may be due to incorrect image name, authentication issues, or network problems.

2. What Needs to Be Analyzed

- Verify the image name and tag in the pod specification.
- Check if the image exists in the container registry.

Inspect Kubernetes events for more details about the image pull failure:

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

- Confirm that proper authentication is set up if using a private registry.
- Check network connectivity to the container registry.

3. How to Resolve Step by Step

Check the Image Name and Tag
Verify that the image name and tag specified in the pod definition are correct. Example:

```
spec:
  containers:
    - name: my-app
      image: <registry>/<image-name>:<tag>
```

1. Ensure that both the image name and tag match the image available in the registry.

Confirm Image Availability
Verify that the image exists in the specified container registry. You can search for it in the registry's UI or use the following command to check:

```
docker pull <registry>/<image-name>:<tag>
```

Inspect Kubernetes Events
Check the events associated with the pod to get more details about the `ImagePullBackOff` error:

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

2. Look for specific error messages related to image pulling.

Check Registry Authentication

If you're pulling from a private registry, ensure Kubernetes is set up to authenticate with it. You can do this by creating a `Secret` for Docker registry authentication and referencing it in the pod spec:

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

Then, specify this secret in your pod definition:

```
spec:
  imagePullSecrets:
    - name: my-registry-secret
```

3. **Check Network Connectivity**
   Ensure that the cluster can reach the container registry. If there are network restrictions (e.g., firewalls or private networks), make sure that your nodes have internet access or can connect to the registry.

Retry the Image Pull
After resolving the issue, delete the pod and let Kubernetes retry pulling the image:

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

Set a Retry Policy
If you want Kubernetes to attempt to pull the image multiple times, you can adjust the `imagePullPolicy`:

```
spec:
  containers:
    - name: my-app
      imagePullPolicy: Always
```

4. Skills Required to Resolve This Issue

- Understanding of Kubernetes image pull mechanism
- Familiarity with Docker and container registries
- Knowledge of Kubernetes secrets for handling private registry authentication
- Basic network troubleshooting skills

5. Conclusion

The `ImagePullBackOff` error occurs when Kubernetes cannot pull the specified container image. Ensuring the correct image name, verifying authentication for private registries, and checking network connectivity are key to resolving this issue.

Error 7: FailedScheduling Error (Pod Pending)

1. Problem Statement

The `FailedScheduling` error occurs when a pod is in the `Pending` state and fails to be scheduled due to lack of resources, affinity constraints, or other scheduling issues. Kubernetes is unable to find a suitable node for the pod.

2. What Needs to Be Analyzed

- Examine the pod's resource requests and limits.
- Check the node's available resources to ensure there is sufficient capacity.
- Review any node selectors, affinity rules, or taints/tolerations in the pod specification.
- Investigate the cluster's overall resource utilization.

Inspect any pod's current state using:

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

3. How to Resolve Step by Step

Check Pod Resource Requests
Examine the pod's resource requests to ensure they are within the cluster's available resources. If the pod requests more resources than available, it may not be scheduled:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "0.5"
```

Inspect Node Resources
Use the following command to check if any node has enough resources to schedule the pod:

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

1. Ensure that CPU, memory, and storage are not fully consumed by other pods.

Check Node Affinity and Taints/Tolerations
Review the pod's affinity settings and any node taints. If the pod has affinity rules that limit which nodes it can be scheduled on, make sure those nodes are available:

```
spec:
  affinity:
```

```
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: "kubernetes.io/hostname"
              operator: In
              values:
                - <node-name>
```

Also, check for any taints on nodes that might prevent scheduling:

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

Check Cluster Resource Usage
If resources are heavily utilized in the cluster, consider scaling the cluster by adding more nodes. Use the following command to view the overall resource usage:

```
kubectl top nodes
```

2. **Scale the Cluster or Node Pools**
   If no suitable nodes are available to accommodate the pod, consider adding more nodes or scaling the existing node pools (if using cloud-based Kubernetes like GKE, AKS, or EKS).

Adjust Resource Requests or Limits
If the pod has over-requested resources, consider reducing its requests for resources. Lowering the memory or CPU requests in the pod specification might help it to be scheduled:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "0.25"
```

Evict Unnecessary Pods

If there are pods consuming unnecessary resources, you can evict them to free up resources for the pending pod:

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

Check for Disk Pressure
If nodes are under disk pressure, pods may not be scheduled. Inspect node conditions to verify if disk pressure is an issue:

```
kubectl describe node <node-name> | grep "Conditions"
```

4. Skills Required to Resolve This Issue

- Knowledge of Kubernetes scheduling and resource management
- Understanding of pod resource requests, limits, and node availability
- Familiarity with Kubernetes node affinity, taints, and tolerations
- Ability to scale Kubernetes clusters
- Experience with cloud-based Kubernetes (optional, for managed services)

5. Conclusion

The `FailedScheduling` error happens when Kubernetes cannot find a suitable node for the pod to be scheduled. Resolving the issue involves checking node resources, pod resource requests, and affinity constraints, along with scaling the cluster if necessary.

Error 8: Pod Disrupted Due to Resource Limit (OOMKilled)

1. Problem Statement

The `OOMKilled` error indicates that a pod was terminated because it exceeded its memory limit. This often results in the pod being killed and restarted by Kubernetes. This issue is caused when the container inside the pod consumes more memory than its defined limit.

2. What Needs to Be Analyzed

Check the pod logs to understand the memory consumption pattern:

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

- Review the memory limits and requests set in the pod specification.

Inspect the Kubernetes events to confirm that the pod was terminated due to memory overuse:

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

3. How to Resolve Step by Step

1. **Check Pod Logs**
   Analyze the pod logs to determine what triggered excessive memory usage. If there are memory leaks or resource-intensive operations, those should be identified and fixed.

Review Memory Requests and Limits
Check if the memory request and limit for the container are configured correctly. If the memory limit is too low, increase it based on the application's needs. For example:

```
spec:
  containers:
    - name: my-app
      image: <image-name>
      resources:
        requests:
          memory: "256Mi"
```

```
    limits:
      memory: "512Mi"
```

Analyze Resource Usage with `kubectl top`
Use `kubectl top` to monitor the node's and pod's resource usage:

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

kubectl top node <node-name>
```

Inspect Kubernetes Events
Look for memory-related events that might provide more information on the OOMKill, especially if it is due to insufficient node resources:

```
kubectl describe pod <pod-name> -n <namespace> | grep -i "oom"
```

2. **Increase Memory Limits**
   If memory usage patterns indicate the pod is consuming more memory than anticipated, increase the memory limits in the pod definition. Adjust the limits based on the logs and resource usage patterns.
3. **Enable Resource Limits on All Containers**
   Ensure that all containers in the pod have appropriate memory requests and limits defined to avoid over-consumption of resources.
4. **Optimize Memory Consumption**
   If your application is using excessive memory, profile and optimize it to reduce memory consumption. Memory leaks or inefficient resource usage should be fixed at the application level.
5. **Use Resource Requests and Limits Wisely**
   Setting proper memory requests and limits ensures Kubernetes schedules pods on nodes with adequate resources and prevents pods from consuming excessive memory. Define both `requests` (amount of memory Kubernetes should reserve) and `limits` (maximum memory the container can use).

4. Skills Required to Resolve This Issue

- Knowledge of Kubernetes resource management (CPU, memory)
- Ability to analyze pod logs and system resource usage
- Familiarity with application performance optimization techniques
- Proficiency in managing resource requests and limits in Kubernetes YAML configurations

5. Conclusion

The `OOMKilled` error occurs when a pod exceeds its memory limit, leading to its termination. To resolve this, ensure proper memory resource allocation, review application memory usage, and

optimize it when necessary. Adjusting memory requests and limits for containers will help prevent this issue in the future.

Error 9: CrashLoopBackOff Error

1. Problem Statement

The `CrashLoopBackOff` error occurs when a pod repeatedly fails to start or crashes after starting, entering a loop of crashes. This often happens when there is an issue with the application inside the pod, configuration errors, or insufficient resources.

2. What Needs to Be Analyzed

Check the pod logs to understand the application failure:

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

Inspect the pod events for any relevant error messages:

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

-
- Analyze resource limits, readiness, and liveness probes.
- Verify the container's entry point and environment variables to ensure they are set correctly.

3. How to Resolve Step by Step

Check Pod Logs for Errors
Review the logs to identify the specific error or failure reason. Common issues could include missing dependencies, incorrect configurations, or application crashes. For example:

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

Inspect Kubernetes Events
Review the events for additional details about the pod failure. This might give insights into why the pod is failing to start:

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

Verify Resource Limits and Requests
Ensure the pod has adequate resources to start successfully. If the resources are too low, the pod might fail to start. Increase the resource limits and requests if necessary:

```
spec:
  containers:
    - name: my-app
      image: <image-name>
      resources:
```

```
      requests:
        memory: "256Mi"
        cpu: "0.5"
      limits:
        memory: "512Mi"
        cpu: "1"
```

Examine Liveness and Readiness Probes
Misconfigured liveness and readiness probes can lead to the pod being restarted repeatedly. Make sure the probes are set correctly and have enough time to succeed:

```
spec:
  containers:
    - name: my-app
      image: <image-name>
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 30
        periodSeconds: 10
      readinessProbe:
        httpGet:
          path: /readiness
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 5
```

1. **Check for Incorrect Configuration or Dependencies**

Verify the configuration files, environment variables, or command-line arguments passed to the container. Incorrect configurations can cause the container to crash. Ensure that the environment variables are correct and that the necessary configuration files are available.

2. **Examine the Container's Entry Point**
   If the container's entry point is incorrectly defined, the application might fail to start. Ensure the `ENTRYPOINT` or `CMD` in the Dockerfile is set correctly.

Debugging with `kubectl exec`
If the pod is starting but crashing, use `kubectl exec` to get a shell into the container and try running the application manually. This can help identify any missing dependencies or environment issues:

```
kubectl exec -it <pod-name> -n <namespace> -- /bin/bash
```

Limit the Restart Policy
If the pod is crashing due to temporary issues, you can modify the restart policy to limit the restarts or disable it completely during debugging. Example:

```
spec:

  restartPolicy: OnFailure
```

4. Skills Required to Resolve This Issue

- Understanding of Kubernetes pod lifecycle and container management
- Experience with log analysis and debugging application crashes
- Knowledge of Kubernetes liveness/readiness probes
- Familiarity with resource management in Kubernetes
- Ability to troubleshoot containerized applications and their configuration

5. Conclusion

The `CrashLoopBackOff` error happens when a pod fails to start and keeps restarting. To resolve this, check the logs for specific error messages, verify resource allocations, and ensure that application configuration, probes, and entry points are correct.

Error 10: ImagePullBackOff Error

1. Problem Statement

The `ImagePullBackOff` error occurs when Kubernetes is unable to pull the container image from the specified container registry. This can happen due to incorrect image names, authentication issues, network problems, or if the image is not available in the registry.

2. What Needs to Be Analyzed

Check pod events for detailed error messages:

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

-
- Verify the image name and tag in the pod configuration.
- Check if the container registry is accessible.
- Ensure that the necessary authentication credentials are provided if the image is private.

3. How to Resolve Step by Step

Check Pod Events for Image Pull Issues
Use the `kubectl describe pod` command to get details on why the image pull is failing:

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

Verify the Image Name and Tag
Ensure that the image name and tag specified in the pod definition are correct. Check if the correct registry is being used.
Example of a valid configuration:

```
spec:
  containers:
    - name: my-app
      image: myregistry.com/my-image:latest
```

Test Pulling the Image Manually
Try pulling the image manually on a local machine or on a Kubernetes node to check if it's available:

```
docker pull myregistry.com/my-image:latest
```

Check Network Connectivity to the Registry

If the image is hosted on an external registry, test network connectivity from the Kubernetes node:

```
curl -v https://myregistry.com
```

Ensure Proper Authentication for Private Registries
If the image is from a private registry, create a Kubernetes secret and attach it to the pod configuration:

```
kubectl create secret docker-registry my-secret \
  --docker-server=myregistry.com \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email>
```

Then, update the pod configuration to use this secret:

```
spec:
  imagePullSecrets:
    - name: my-secret
```

Use `kubectl get secrets` to Verify Secret Exists
Ensure that the secret is created properly:

```
kubectl get secrets
```

Check Node-Level Issues
If the image pull is failing on all pods, check if the Kubernetes worker nodes are running out of disk space:

```
df -h
```

Restart the Kubelet Service
If there are persistent image pull issues, restarting the kubelet service on the affected node might help:

```
systemctl restart kubelet
```

4. Skills Required to Resolve This Issue

- Familiarity with Kubernetes pod scheduling and image management
- Experience with container registries (Docker Hub, ECR, GCR, etc.)
- Ability to debug network connectivity issues
- Understanding of Kubernetes secrets for authentication

5. Conclusion

The `ImagePullBackOff` error occurs when Kubernetes cannot pull a container image due to incorrect image names, authentication issues, or network problems. By checking pod events, verifying image credentials, and testing connectivity, you can resolve this issue efficiently.

Error 11: ErrImagePull Error

1. Problem Statement

The `ErrImagePull` error occurs when Kubernetes fails to download a container image from the specified container registry. This is similar to `ImagePullBackOff` but happens during the first attempt before Kubernetes retries. It can be caused by incorrect image names, authentication issues, network failures, or an unavailable image.

2. What Needs to Be Analyzed

Check the pod events for error details:

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

-
- Verify the image name, tag, and registry URL.
- Ensure the registry is accessible and the image exists.
- Check authentication credentials if pulling from a private registry.

3. How to Resolve Step by Step

Check Pod Events for Image Pull Failures
Run the following command to inspect the error message:

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

Verify the Image Name and Tag
Ensure that the image name and tag in the pod specification are correct:

```
spec:
  containers:
    - name: my-app
      image: myregistry.com/my-image:v1.0
```

Try Pulling the Image Manually
If the image is public, attempt to pull it on your local system:

```
docker pull myregistry.com/my-image:v1.0
```

1. If the pull fails, check if the image exists in the registry.

Check Network Connectivity to the Registry
Verify if the Kubernetes nodes can access the image registry:

```
curl -I https://myregistry.com
```

Ensure Proper Authentication for Private Registries
If using a private registry, create and configure an image pull secret:

```
kubectl create secret docker-registry my-secret \
  --docker-server=myregistry.com \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email>
```

Then update the pod configuration:

```
spec:
  imagePullSecrets:
    - name: my-secret
```

Check if Kubernetes Nodes Have Sufficient Storage
If the node is out of disk space, images may fail to pull. Check disk usage:

```
df -h
```

Inspect Docker Daemon Logs on the Node
If using containerd or Docker, check logs for further troubleshooting:

```
journalctl -u docker.service --no-pager | tail -n 50
```

or

```
journalctl -u containerd.service --no-pager | tail -n 50
```

Restart the Kubelet Service
If all else fails, restart the kubelet service:

```
systemctl restart kubelet
```

4. Skills Required to Resolve This Issue

- Knowledge of Kubernetes pod lifecycle and image pulling
- Familiarity with container registries (Docker Hub, ECR, GCR, etc.)
- Ability to debug network and authentication issues
- Understanding of Kubernetes secrets and disk storage management

5. Conclusion

The `ErrImagePull` error occurs when Kubernetes cannot retrieve a container image due to incorrect configurations, authentication failures, or network issues. By checking logs, verifying credentials, and testing connectivity, this issue can be resolved effectively.

Error 12: CrashLoopBackOff Error

1. Problem Statement

The `CrashLoopBackOff` error occurs when a pod repeatedly crashes and Kubernetes keeps restarting it. This can be caused by misconfigurations, application errors, missing dependencies, insufficient resources, or permission issues.

2. What Needs to Be Analyzed

Check pod status and events for error messages:

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

View pod logs to identify application errors:

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

- Verify resource limits in the deployment configuration.
- Check for missing environment variables or config maps.

3. How to Resolve Step by Step

Check Pod Events and Restart Patterns
Run the following command to inspect error messages:

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

Check Container Logs for Errors
View logs to understand why the container is crashing:

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

If the pod has multiple containers, specify the container name:

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

Run the Pod Manually for Debugging
If the logs are unclear, try running the pod in an interactive mode:

```
kubectl run debug-pod --image=<image-name> --rm -it -- bash
```

Verify Environment Variables and ConfigMaps
Check if the pod is missing required environment variables:

```
kubectl get configmap -n <namespace>
```

If required, update the pod spec to provide missing variables:

```
spec:
  containers:
    - name: my-app
      image: myregistry.com/my-image:v1.0
      env:
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: database_url
```

Check Resource Limits and Requests
If the pod is crashing due to memory exhaustion, increase the resource limits:

```
spec:
  containers:
    - name: my-app
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
        limits:
          memory: "1Gi"
          cpu: "500m"
```

Verify File Permissions and Mounts
If the application fails due to missing file permissions, check volume mounts:

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

Check for Readiness and Liveness Probe Failures
If probes are misconfigured, the pod might restart continuously. Check probe configurations:

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

Manually Delete and Restart the Pod
If the pod is stuck in a crash loop, restart it:

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

1. Then, redeploy it.

Check for Node-Level Issues
If the issue persists across multiple pods, check node health:

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

2. **Debug Using an Alternative Image**
   If you suspect an issue with the container image, try running a different base image for debugging:

```
kubectl run debug --image=alpine --rm -it -- sh
```

4. Skills Required to Resolve This Issue

- Familiarity with Kubernetes pod lifecycle
- Debugging application logs and container runtime
- Understanding of resource management in Kubernetes
- Knowledge of readiness and liveness probes

5. Conclusion

The `CrashLoopBackOff` error occurs when a pod repeatedly crashes due to misconfigurations, missing dependencies, or insufficient resources. By analyzing logs, checking resource limits, and debugging the container manually, the issue can be resolved.

Error 13: Node Not Ready

1. Problem Statement

The `Node Not Ready` error occurs when a Kubernetes node enters an unhealthy state and is marked as `NotReady`. This prevents the scheduler from assigning new pods to the node and can lead to service disruptions. Common causes include networking issues, kubelet failures, disk pressure, or resource exhaustion.

2. What Needs to Be Analyzed

Check the status of all nodes:

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

Describe the problematic node for more details:

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

Check kubelet logs for errors:

```
journalctl -u kubelet -n 50 --no-pager
```

- Verify network connectivity and resource utilization.

3. How to Resolve Step by Step

Check Node Status
Run the following command to list all nodes and check their status:

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

1. If a node is marked as `NotReady`, proceed with further analysis.

Describe the Node for More Details
Get additional information about the node's condition:

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

2. Look for conditions like `MemoryPressure`, `DiskPressure`, or `NetworkUnavailable`.

Check Kubelet Logs
If the issue is kubelet-related, inspect logs:

```
journalctl -u kubelet -n 50 --no-pager
```

Verify Resource Usage on the Node

Check CPU, memory, and disk utilization:

```
top
```

```
df -h
```

```
free -m
```

Restart Kubelet on the Node
If kubelet is unresponsive, restart it:

```
systemctl restart kubelet
```

Then, verify its status:

```
systemctl status kubelet
```

Check and Restart Networking Services
If networking issues are detected, restart network services:

```
systemctl restart networking
```

Verify and Restart Docker or Container Runtime
If the node uses Docker, restart it:

```
systemctl restart docker
```

If using containerd:

```
systemctl restart containerd
```

Check if the Node is Out of Disk Space
If disk pressure is the issue, free up space:

```
du -sh /var/lib/docker
```

```
rm -rf /var/lib/docker/tmp/*
```

Drain and Rejoin the Node
If the issue persists, remove the node from the cluster and rejoin:

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

```
kubectl delete node <node-name>
```

3. Then, re-add the node.

Reboot the Node as a Last Resort
If all else fails, restart the node:

```
reboot
```

4. Skills Required to Resolve This Issue

- Kubernetes node management and troubleshooting
- Linux system administration
- Network debugging
- Understanding kubelet and container runtime

5. Conclusion

The `Node Not Ready` error can be caused by resource exhaustion, networking failures, or kubelet issues. By inspecting logs, verifying system health, and restarting critical services, the node can be restored to a healthy state.

Error 14: ImagePullBackOff Error

1. Problem Statement

The `ImagePullBackOff` error occurs when a Kubernetes pod fails to pull a container image from the specified registry. This can be due to incorrect image names, authentication failures, network issues, or the image not being available in the registry.

2. What Needs to Be Analyzed

Check the status of the pod:

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

Describe the pod to get detailed error messages:

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

- Verify that the image name and tag are correct.
- Ensure that the container registry is accessible.
- Check if authentication credentials are required for private images.

3. How to Resolve Step by Step

Check the Pod Events and Errors
Run the following command to see error details:

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

1. Look for messages like `ErrImagePull` or `ImagePullBackOff`.

Verify Image Name and Tag
Ensure the image exists in the registry and the tag is correct:

```
docker pull <image-name>:<tag>
```

2. If the pull fails, check for typos or missing images.

Test Image Pulling on a Node
SSH into a Kubernetes node and manually pull the image:

```
docker pull <image-name>:<tag>
```

3. If this fails, verify network access and authentication.

Check Private Registry Authentication
If the image is in a private registry, ensure Kubernetes has the correct secret:

```
kubectl get secrets -n <namespace>
```

If missing, create a secret:

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

Then, reference it in the deployment:

```
spec:
  imagePullSecrets:
    - name: my-registry-secret
```

Check Network Connectivity
If the registry is external, test connectivity:

```
curl -v <registry-url>
```

4. Ensure firewall rules allow access.

Force Image Pull and Restart Pod
If the issue is a stale image cache, force Kubernetes to pull the latest image:

```
kubectl delete pod <pod-name> -n <namespace>
kubectl apply -f <deployment-file>.yaml
```

Check Node Disk Space
If the node is out of space, remove unused images:

```
docker system prune -a
```

Restart Kubelet if Necessary
If kubelet is unresponsive, restart it:

```
systemctl restart kubelet
```

4. Skills Required to Resolve This Issue

- Kubernetes container image management
- Docker registry authentication and secrets

- Network debugging
- Kubernetes deployment troubleshooting

5. Conclusion

The `ImagePullBackOff` error is typically caused by incorrect image names, authentication failures, or network issues. Verifying image details, configuring registry secrets, and ensuring network connectivity can resolve this error.

Error 15: CrashLoopBackOff Error

1. Problem Statement

The `CrashLoopBackOff` error occurs when a pod repeatedly crashes and Kubernetes restarts it in a loop. This can be caused by application errors, missing dependencies, incorrect environment variables, or resource constraints.

2. What Needs to Be Analyzed

Check the pod status:

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

Describe the pod for more details:

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

View container logs to identify errors:

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

- Check resource usage and limits.

3. How to Resolve Step by Step

Check Pod Events and Error Messages
Run:

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

1. Look for reasons like `OOMKilled`, `Exit Code 1`, or `Segmentation Fault`.

Analyze Container Logs
View logs for debugging:

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

2. **Ensure Required Configurations Are Set**

Verify environment variables:

```
kubectl get configmap -n <namespace>
```

Check secrets if required:

```
kubectl get secrets -n <namespace>
```

3. **Check Application-Level Issues**
    - If the container exits due to a missing dependency, update the container image.

If a script is failing on startup, modify the entrypoint:

```
command: ["sleep", "3600"]
```

Adjust Resource Limits if Needed
If the container is OOMKilled, increase memory:

```
resources:
  limits:
    memory: "512Mi"
  requests:
    memory: "256Mi"
```

Manually Run the Container for Debugging
Start the container interactively:

```
kubectl run test-container --image=<image-name> -it -- /bin/sh
```

Delete and Redeploy the Pod

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

kubectl apply -f <deployment.yaml>
```

4. Skills Required to Resolve This Issue

- Kubernetes pod troubleshooting
- Application debugging
- Resource management
- Log analysis

5. Conclusion

The `CrashLoopBackOff` error is often due to application failures, missing dependencies, or resource constraints. Checking logs, adjusting configurations, and debugging interactively can resolve the issue.

Error 16: OOMKilled (Out of Memory Killed) Error

1. Problem Statement

The `OOMKilled` error occurs when a pod exceeds its allocated memory limit, causing the Kubernetes scheduler to terminate it. This happens due to insufficient memory allocation, memory leaks, or unoptimized applications.

2. What Needs to Be Analyzed

Check pod status:

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

Describe the pod to check termination reasons:

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

View logs for memory-related errors:

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

- Check resource requests and limits in the pod spec.

3. How to Resolve Step by Step

Check Pod Events and Exit Codes
Run:
```
kubectl describe pod <pod-name> -n <namespace>
```

1. Look for `Reason: OOMKilled` under `State: Terminated`.

Analyze Application Logs
View logs for memory-related errors:

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

2. **Check Node Memory Availability**

Check node memory usage:

```
kubectl top node
```

    - If the node is out of memory, consider adding more nodes.

Adjust Memory Requests and Limits

If the pod exceeds limits, update the resource allocation in the deployment YAML:
yaml

```
resources:
  requests:
    memory: "512Mi"
  limits:
    memory: "1Gi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

Enable Memory Limits for the Container
If memory leaks occur, enforce stricter limits:
yaml

```
command: ["java", "-Xmx512m", "-jar", "app.jar"]
```

3. **Optimize Application Memory Usage**
    - Profile memory usage in the application.
    - Use garbage collection techniques if applicable.
    - Optimize large data structures.

Restart the Pod

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

4. Skills Required to Resolve This Issue

- Kubernetes resource management
- Application performance tuning
- Memory profiling

5. Conclusion

The `OOMKilled` error is caused by excessive memory consumption. Setting appropriate resource limits, optimizing memory usage, and monitoring node resources can prevent it.

Error 17: ErrImageNeverPull Error

1. Problem Statement

The `ErrImageNeverPull` error occurs when Kubernetes is unable to pull the specified container image due to the `imagePullPolicy` setting or incorrect image availability.

2. What Needs to Be Analyzed

Check the pod status:

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

Describe the pod for error details:

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

- Check if the `imagePullPolicy` is set to `Never`.
- Verify if the image exists on the local node.

3. How to Resolve Step by Step

Check the Pod Events
Run:

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

1. Look for `ErrImageNeverPull`.

Verify `imagePullPolicy` Setting
Open the deployment YAML and ensure `imagePullPolicy` is correct:

```
spec:
  containers:
    - name: my-container
      image: my-registry/my-image:latest
      imagePullPolicy: Always
```

Ensure the Image Exists on the Node
If `imagePullPolicy: Never` is set, the image must be available locally:

```
docker images | grep my-image
```

If the image is missing, pull it manually:

```
docker pull my-registry/my-image:latest
```

Change Image Pull Policy
Update the YAML file:

```
imagePullPolicy: IfNotPresent
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

Use a Private Registry Secret if Required
If using a private registry, create a secret:

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

Reference it in the deployment:

```
spec:
  imagePullSecrets:
    - name: my-secret
```

Restart the Pod

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

4. Skills Required to Resolve This Issue

- Kubernetes container image management
- Docker registry authentication
- YAML configuration

5. Conclusion

The `ErrImageNeverPull` error occurs due to incorrect `imagePullPolicy` settings or missing local images. Ensuring the correct policy and availability of the image resolves this issue.

Error 18: ImagePullBackOff Error

1. Problem Statement

The `ImagePullBackOff` error occurs when Kubernetes is unable to pull the specified container image from the container registry. This can be caused by incorrect image names, authentication issues, or network problems.

2. What Needs to Be Analyzed

Check pod status:

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

-

Describe the pod for detailed error messages:

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

-
- Check the container registry credentials if using a private registry.
- Verify network connectivity to the registry.
- Ensure the image tag exists in the repository.

3. How to Resolve Step by Step

Check the Pod Events
Run:

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

1. Look for reasons like `ErrImagePull` or `Back-off pulling image`.

Verify the Image Name and Tag
Ensure the image exists in the registry:

```
docker pull my-registry/my-image:latest
```

If the image is missing, push it:

```
docker tag my-image my-registry/my-image:latest
```

```
docker push my-registry/my-image:latest
```

Check ImagePullPolicy
Update the YAML file:

```
imagePullPolicy: IfNotPresent
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

2.

Authenticate with the Private Registry
If using a private registry, create a secret:

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

Reference it in the deployment:

```
spec:
  imagePullSecrets:
    - name: my-secret
```

Check Network Connectivity to the Registry
Run:

```
curl -v https://<registry-url>
```

3. If the registry is unreachable, check firewall or DNS settings.

Restart the Pod

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

4. Skills Required to Resolve This Issue

- Kubernetes image management
- Docker registry troubleshooting
- Network diagnostics

5. Conclusion

The `ImagePullBackOff` error is caused by incorrect image names, authentication failures, or network issues. Ensuring the correct image, authentication, and registry connectivity resolves this issue.

Error 19: CrashLoopBackOff Error

1. Problem Statement

The `CrashLoopBackOff` error occurs when a pod continuously crashes and Kubernetes restarts it in a loop. This can happen due to application errors, incorrect configurations, missing dependencies, or insufficient resources.

2. What Needs to Be Analyzed

Check pod status:

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

Describe the pod for failure reasons:

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

Check container logs for application errors:

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

- Verify readiness and liveness probes.
- Check resource limits and node capacity.

3. How to Resolve Step by Step

Check Pod Events and Logs
Run:

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

1. Look for `CrashLoopBackOff` under `State: Terminated`.

View Application Logs
Check logs to identify errors:

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

2. **Check for Missing Dependencies**

Ensure all required environment variables are set:

```
kubectl get pod <pod-name> -o yaml | grep env
```

Validate required services are running:

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

Verify Readiness and Liveness Probes
If probes are incorrectly configured, update them in the YAML file:

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

Apply changes:

```
kubectl apply -f <deployment.yaml>
```

Adjust Resource Limits
If the pod is failing due to insufficient resources, increase limits:

```
resources:
  requests:
    cpu: "250m"
    memory: "256Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

Apply changes:

```
kubectl apply -f <deployment.yaml>
```

Manually Restart the Pod

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

4. Skills Required to Resolve This Issue

- Kubernetes pod management
- Application debugging
- Resource allocation tuning

5. Conclusion

The `CrashLoopBackOff` error is caused by repeated container crashes due to misconfigurations, missing dependencies, or resource issues. Debugging logs, adjusting resource limits, and fixing readiness probes help resolve this issue.

Error 20: FailedScheduling Error

1. Problem Statement

The `FailedScheduling` error occurs when Kubernetes cannot schedule a pod to any node due to resource constraints, unsatisfied affinities, or insufficient node capacity.

2. What Needs to Be Analyzed

Check pod status:

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

Describe the pod for scheduling errors:

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

Check node resources:

```
kubectl top nodes
```

- Review pod affinity, anti-affinity, and taints settings.

3. How to Resolve Step by Step

Check Pod Events
Run:

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

1. Look for the `FailedScheduling` event, which indicates why the pod cannot be scheduled.
2. **Analyze Node Resources**

Check CPU and memory usage:

```
kubectl top nodes
```

    - If nodes are running out of resources, consider adding more nodes or increasing node capacity.

Check Resource Requests and Limits
Ensure resource requests are not higher than available resources:

```
resources:
  requests:
    cpu: "500m"
```

```
    memory: "1Gi"
  limits:
    cpu: "1"
    memory: "2Gi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

Review Affinity and Taints
Ensure pod affinity, anti-affinity, and taints are set correctly in the YAML:

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

Apply changes:

```
kubectl apply -f <deployment.yaml>
```

Check Node Selectors
If you're using node selectors, ensure they are properly set to match available nodes:

```
nodeSelector:
  disktype: ssd
```

Restart the Pod
If the configuration is correct and the issue persists, delete the pod and Kubernetes will attempt to

reschedule it:

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

4. Skills Required to Resolve This Issue

- Kubernetes scheduler management
- Node resource monitoring
- Pod affinity and anti-affinity configuration

5. Conclusion

The `FailedScheduling` error occurs when Kubernetes cannot find a suitable node for a pod due to resource constraints or configuration issues. Ensuring correct resource allocation and affinity settings resolves this issue.

Error 21: ResourceQuotaExceeded Error

1. Problem Statement

The `ResourceQuotaExceeded` error occurs when a Kubernetes namespace exceeds the resource quotas (e.g., CPU, memory, or number of pods) defined for it.

2. What Needs to Be Analyzed

Check the pod status:

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

Describe the pod for resource quota violations:

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

Review the resource quota defined in the namespace:

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

Check the resource usage within the namespace:

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

3. How to Resolve Step by Step

Check the Pod Events and Error Details
Run:

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

1. Look for errors related to exceeding resource limits.

Inspect Resource Quotas in the Namespace
Verify the resource quota:

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

2. Check if the namespace has exceeded any of the quota limits.
3. **Examine the Resource Usage**

Run the following command to check current resource usage:

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

- If the resources are exceeding the quota, consider scaling back on resource usage.

Modify Resource Requests and Limits
Adjust the resource requests and limits in your YAML to stay within the allocated quota:

```
resources:
  requests:
    cpu: "100m"
    memory: "128Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

Increase the Resource Quota (if necessary)
If needed, you can increase the resource quota in the namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    requests.cpu: "10"
    requests.memory: "10Gi"
    pods: "50"
```

Apply the changes:

```
kubectl apply -f <resourcequota.yaml>
```

Clean Up Unnecessary Resources
Delete unused resources (pods, deployments, etc.) to free up quota:

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

Restart the Pod
If everything is in order, restart the pod:

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

4. Skills Required to Resolve This Issue

- Kubernetes resource management
- Resource quota configuration
- YAML configuration

5. Conclusion

The `ResourceQuotaExceeded` error happens when a namespace exceeds the defined resource limits. Resolving the issue involves either reducing resource consumption or increasing the quota, depending on the requirements.

Error 22: ImagePullBackOff Error

1. Problem Statement

The `ImagePullBackOff` error occurs when Kubernetes is unable to pull the required container image for a pod, typically due to issues with image availability, registry authentication, or incorrect image tags.

2. What Needs to Be Analyzed

Check the pod status:

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

Describe the pod to find the error details:

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

- Check the image registry and authentication credentials.
- Verify the image tag and registry URL in the pod specification.

3. How to Resolve Step by Step

Check the Pod Events for Error Details
Run:

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

1. Look for `ImagePullBackOff` under the `State: Waiting` section. This will provide more information on the issue.

Verify the Image Tag and Registry
Ensure that the image tag and registry URL are correct. For example, check if the image in the YAML is correctly specified:

```
containers:
  - name: my-app
    image: my-registry/my-app:latest
```

2. **Check the Image Availability**
   Ensure the image exists in the specified registry. If using a private registry, verify that the image is pushed and accessible.

Check for Authentication Issues
If pulling from a private registry, ensure Kubernetes has the necessary credentials. Create a Kubernetes secret to store the credentials:

```
kubectl create secret docker-registry <secret-name> \
  --docker-server=<registry-url> \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email>
```

Reference the secret in the pod YAML:

```
imagePullSecrets:
  - name: <secret-name>
```

Test the Image Pull Command
Try pulling the image manually to ensure it's accessible:

```
docker pull my-registry/my-app:latest
```

3. **Ensure Network Connectivity**
   Make sure there are no network issues preventing access to the registry. Check your node's internet connectivity.

Reapply the Changes
After resolving the issue, apply the updated configuration:

```
kubectl apply -f <deployment.yaml>
```

Restart the Pod
If the issue persists, delete the pod and let Kubernetes restart it:

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

4. Skills Required to Resolve This Issue

- Docker and Kubernetes image management
- Registry configuration and authentication
- YAML configuration

5. Conclusion

The `ImagePullBackOff` error occurs when Kubernetes is unable to pull a container image, often due to incorrect image tags, registry issues, or authentication problems. Ensuring the image is available and credentials are correctly configured will resolve the issue.

Error 23: CrashLoopBackOff Error

1. Problem Statement

The `CrashLoopBackOff` error occurs when a container repeatedly crashes after being started by Kubernetes, usually due to issues within the container or its configuration.

2. What Needs to Be Analyzed

Check the pod status:

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

Describe the pod for detailed error information:

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

Look at the container logs to determine the cause of the crashes:

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

3. How to Resolve Step by Step

Check the Pod Events and Logs
Run:

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

1. Look for events related to container crashes, such as OOM (Out of Memory) or segmentation faults.

Analyze the Container Logs
Retrieve logs to determine the root cause of the crash:

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

2. Look for errors in the logs that may indicate configuration issues, missing files, or other problems inside the container.

Review Resource Requests and Limits
Ensure that the container has sufficient CPU and memory resources. Adjust the `resources` section in your deployment YAML:

```
resources:
  requests:
```

```
    cpu: "500m"
    memory: "1Gi"
  limits:
    cpu: "1"
    memory: "2Gi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

3.
4. **Check for Application Configuration Issues**
   If the application inside the container is misconfigured (e.g., incorrect environment variables or missing dependencies), fix the configuration. Ensure that all necessary environment variables and files are set correctly.

Verify the Liveness and Readiness Probes
If your pod has liveness or readiness probes configured, ensure they are set up correctly to prevent Kubernetes from prematurely restarting the container. For example:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 3
  periodSeconds: 5
```

Apply changes if necessary:

```
kubectl apply -f <deployment.yaml>
```

5. **Check Image Compatibility**
   Ensure the image you're using is compatible with the architecture of the cluster nodes. If using a custom image, test it locally to confirm that it runs as expected.
6. **Increase Restart Policy Delay**
   If the application takes time to initialize, consider increasing the delay between restarts by configuring the `restartPolicy` or using `initialDelaySeconds` in the probes.

Manually Restart the Pod
If you've fixed the issue, restart the pod to ensure it starts cleanly:

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

4. Skills Required to Resolve This Issue

- Container troubleshooting
- Resource management and optimization
- Application configuration and debugging
- Kubernetes pod and container management

5. Conclusion

The `CrashLoopBackOff` error happens when a container keeps crashing due to resource limits, application bugs, or misconfigurations. Checking the logs, adjusting resources, and resolving configuration issues can resolve this error.

Error 24: NoNodesAvailable Error

1. Problem Statement

The `NoNodesAvailable` error occurs when Kubernetes is unable to find a suitable node to schedule a pod due to resource constraints, taints, or node selector misconfigurations.

2. What Needs to Be Analyzed

Check the pod status to confirm if the pod is stuck in `Pending` state:

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

Describe the pod to investigate why it isn't being scheduled:

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

Verify if there are available nodes by checking the node status:

```
kubectl get nodes
```

3. How to Resolve Step by Step

Check Pod Events for Scheduling Failures
Run:

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

1. Look for messages that indicate why the pod couldn't be scheduled, such as resource limits, node taints, or insufficient resources.

Verify Node Resources
Make sure there are enough resources (CPU, memory) on the available nodes to accommodate the pod. Check node resource usage:

```
kubectl top nodes
```

2. If the nodes are overloaded, either free up resources or add new nodes to the cluster.

Check Node Taints and Pod Tolerations
Taints on nodes can prevent pods from being scheduled unless the pod has matching tolerations. Check for node taints:

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

If there are taints, ensure your pod includes the correct tolerations:

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

Verify Node Selectors
If the pod specification includes a node selector, ensure it matches the labels on available nodes. Check the node selector in the pod YAML:

```
nodeSelector:
  disktype: ssd
```

Verify that the label `disktype=ssd` is applied to one or more nodes:

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

Check for Pod Affinity or Anti-Affinity Rules
If the pod has affinity or anti-affinity rules, verify that they don't restrict pod scheduling unnecessarily. Check the pod spec for any affinity configuration:

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: "role"
              operator: In
              values:
                - "frontend"
        topologyKey: "kubernetes.io/hostname"
```

Check for Pod Resource Requests and Limits
Ensure that the pod's resource requests are not too high for the available nodes. For example, reduce the CPU or memory requests in the pod specification:

```
resources:
  requests:
```

```
    cpu: "100m"
    memory: "128Mi"
  limits:
    cpu: "500m"
    memory: "1Gi"
```

3. **Scale Up the Cluster**
   If the cluster is under-resourced, consider scaling up the number of nodes to provide more capacity for scheduling pods.

Reschedule the Pod
If you've adjusted the configuration, apply the changes and reschedule the pod:

```
kubectl apply -f <deployment.yaml>
```

Manually Delete Stuck Pods
If the pod is still in `Pending`, you can delete it and let Kubernetes reschedule it:

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

4. Skills Required to Resolve This Issue

- Node management and scheduling
- Resource management and optimization
- Kubernetes taints, tolerations, and affinity rules
- YAML configuration for pod specifications

5. Conclusion

The `NoNodesAvailable` error happens when Kubernetes can't schedule a pod due to resource constraints, node availability, taints, or misconfigurations. Resolving the issue involves analyzing node resources, configuring appropriate taints and tolerations, and adjusting the pod's resource requests or node selectors.

Error 25: ResourceQuotaExceeded Error

1. Problem Statement

The `ResourceQuotaExceeded` error occurs when a namespace exceeds its defined resource limits, such as CPU, memory, or the number of resources (pods, services, etc.) in use, as set by a `ResourceQuota` object.

2. What Needs to Be Analyzed

Check the pod status:

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

Verify the resource quota for the namespace:

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

- Check if any resources have exceeded the limits set in the `ResourceQuota` object.

3. How to Resolve Step by Step

Check the Resource Quota
Run:

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

1. This will show the resource usage and the limits imposed by the `ResourceQuota`. Look for fields like `cpu`, `memory`, and other resource types.
2. **Review Resource Usage**
   Compare the current usage to the defined quotas. If the usage exceeds the quota, either scale down the resources or increase the quota limits.

Modify the Resource Quota (if necessary)
If you need to increase the quota, update the `ResourceQuota` definition. For example, to increase CPU and memory limits:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-quota
  namespace: <namespace>
```

```
spec:
  hard:
    requests.cpu: "10"
    requests.memory: "20Gi"
    limits.cpu: "20"
    limits.memory: "40Gi"
```

Apply the changes:

```
kubectl apply -f <resource-quota.yaml>
```

Clean Up Unused Resources
If possible, remove unused resources (pods, services, etc.) to free up space. Use commands like:

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

Ensure Pods are Properly Scoped
Ensure that each pod is within the limits defined by the `ResourceQuota`:

```
resources:
  requests:
    cpu: "100m"
    memory: "128Mi"
  limits:
    cpu: "500m"
    memory: "1Gi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

3. **Monitor Resource Usage**
   Set up resource usage monitoring tools like Prometheus to keep track of resource consumption and avoid future quota violations.

Reapply the Changes
After modifying the quota or cleaning up resources, ensure the system is updated by applying the modified configurations:

```
kubectl apply -f <namespace-config.yaml>
```

4. Skills Required to Resolve This Issue

- Kubernetes resource management
- YAML configuration for resource quotas
- Monitoring and troubleshooting resource usage
- Namespace management

5. Conclusion

The `ResourceQuotaExceeded` error occurs when the namespace exceeds its resource limits. To resolve this, you can either modify the `ResourceQuota` or clean up unused resources to bring the usage back within the allowed limits.

Error 26: NetworkPolicyDenied Error

1. Problem Statement

The `NetworkPolicyDenied` error occurs when a Kubernetes pod cannot communicate with another pod or service due to a restrictive `NetworkPolicy` that denies the necessary traffic.

2. What Needs to Be Analyzed

Check the pod and service communication status:

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

```
kubectl get services -n <namespace>
```

Describe the pod and look for network-related errors:

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

Review the existing `NetworkPolicy` in the namespace:

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

3. How to Resolve Step by Step

Check for Existing Network Policies
Run:

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

1. Review any existing `NetworkPolicy` objects. Look for policies that may be restricting ingress or egress traffic to or from the pod.

Analyze the NetworkPolicy Definition
If a `NetworkPolicy` is in place, describe it to see its exact rules:

```
kubectl describe networkpolicy <policy-name> -n <namespace>
```

2. Ensure that the policy allows the necessary ingress and egress traffic between the pods and services.

Adjust the NetworkPolicy (if needed)
Modify the `NetworkPolicy` to allow the desired communication. For example, to allow all pods in the namespace to communicate with each other, the policy might look like:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
  namespace: <namespace>
spec:
  podSelector: {}
  ingress:
    - {}
  egress:
    - {}
  policyTypes:
    - Ingress
    - Egress
```

Apply the changes:

```
kubectl apply -f <network-policy.yaml>
```

Ensure the Pods Match the NetworkPolicy
Verify that the pods are correctly labeled to match the selectors in the `NetworkPolicy`. For instance, if the `NetworkPolicy` targets pods with a certain label:

```yaml
podSelector:
  matchLabels:
    app: frontend
```

Test the Network Connectivity
After modifying the policy, test the connectivity between pods and services using `curl` or `ping` inside the containers:

```
kubectl exec -it <pod-name> -n <namespace> -- curl <service-name>:<port>
```

3. **Verify the CNI Plugin Configuration**
   If you are using a specific CNI (Container Network Interface) plugin, ensure that it is properly configured and supports the desired network policies.

Check for Network Policy Logs
If the problem persists, review the network logs for any issues related to the application of the network policies:

```
kubectl logs <network-policy-controller-pod> -n kube-system
```

Reapply the NetworkPolicy
If you've made changes, reapply the network policy to ensure it's in effect:

```
kubectl apply -f <updated-network-policy.yaml>
```

4. Skills Required to Resolve This Issue

- Network troubleshooting in Kubernetes
- Understanding of NetworkPolicy concepts
- YAML configuration for network policies
- CNI plugin configuration
- Pod and service communication

5. Conclusion

The `NetworkPolicyDenied` error occurs when a restrictive `NetworkPolicy` prevents necessary communication between pods or services. Resolving this requires reviewing and adjusting the `NetworkPolicy` to ensure it allows the required traffic, as well as verifying pod selectors and CNI plugin configurations.

Error 27: Liveness Probe Failed

1. Problem Statement

The `LivenessProbe` failure occurs when a Kubernetes pod continuously restarts due to failing health checks. This means the application inside the container is not responding to the health check defined in the `livenessProbe`, causing Kubernetes to restart the pod repeatedly.

2. What Needs to Be Analyzed

Check the status of the pod:

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

- Look for `CrashLoopBackOff` or `Restarting` statuses.

Describe the pod to check probe failure logs:

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

- Look for messages like `Liveness probe failed: HTTP probe failed with status code 500`.

Check pod logs for application errors:

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

- Review the liveness probe configuration in the pod specification.

3. How to Resolve Step by Step

Identify the Liveness Probe Configuration
Run:

```
kubectl get pod <pod-name> -o yaml -n <namespace>
```

Look for the `livenessProbe` section:

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 3
```

1. **Check if the Application is Actually Running**

Use `kubectl exec` to enter the container:

```
kubectl exec -it <pod-name> -n <namespace> -- /bin/sh
```

Try manually accessing the health endpoint:

```
curl http://localhost:8080/health
```

If the service is unreachable or returns an error, check application logs:

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

Adjust the Liveness Probe Configuration
If the application takes time to start, increase the `initialDelaySeconds`:

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 15
  failureThreshold: 5
```

Apply the updated configuration:

```
kubectl apply -f <deployment.yaml>
```

2. **Verify the Application Health Endpoint**
    - Ensure that the `/health` endpoint is correctly implemented in the application.

Test it inside the container using:

```
curl http://localhost:8080/health
```

    - If the endpoint is missing or incorrect, update the application code.

Switch to a More Reliable Probe Type
If using HTTP, consider using a TCP or command probe if the application does not expose an HTTP health check:

```
livenessProbe:
  tcpSocket:
    port: 8080
```

3. **Restart and Monitor the Pod**

Delete the pod to force a fresh start:

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

Monitor the pod status:

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

Review Kubernetes Events for Further Clues

```
kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>
```

4. Look for repeated liveness probe failures.

4. Skills Required to Resolve This Issue

- Kubernetes health checks (`livenessProbe`, `readinessProbe`)
- Application debugging inside containers
- YAML configuration for Kubernetes
- Log analysis and troubleshooting

5. Conclusion

The `Liveness Probe Failed` error usually happens due to a misconfigured health check, an incorrect endpoint, or an application that takes too long to start. Adjusting the `livenessProbe` settings, debugging the application, and ensuring the health endpoint is working correctly can help resolve the issue.

Error 28: Readiness Probe Failed

1. Problem Statement

The `ReadinessProbe` failure occurs when a Kubernetes pod is marked as **Not Ready**, preventing it from receiving traffic. This usually happens when the application inside the container does not respond successfully to the readiness probe.

2. What Needs to Be Analyzed

Check the pod status:

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

- If the pod is stuck in the `Not Ready` state, it means the readiness probe is failing.

Describe the pod for more details:

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

Look for messages like:

```
Readiness probe failed: HTTP probe failed with status code 503
```

Check the container logs for errors:

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

- Review the readiness probe configuration in the pod specification.

3. How to Resolve Step by Step

Identify the Readiness Probe Configuration
Run:

```
kubectl get pod <pod-name> -o yaml -n <namespace>
```

Look for the `readinessProbe` section:

```
readinessProbe:
```

```
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 3
```

1. **Check If the Application Is Ready**

Use `kubectl exec` to enter the container:

```
kubectl exec -it <pod-name> -n <namespace> -- /bin/sh
```

Try manually accessing the readiness endpoint:

```
curl http://localhost:8080/ready
```

If the service is unreachable or returns an error, check application logs:

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

Adjust the Readiness Probe Configuration
If the application takes time to start, increase the `initialDelaySeconds`:
yaml

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 15
  failureThreshold: 5
```

Apply the updated configuration:

```
kubectl apply -f <deployment.yaml>
```

2. **Verify the Readiness Endpoint in the Application**
    - Ensure that the `/ready` endpoint is correctly implemented in the application.

Test it inside the container using:

```
curl http://localhost:8080/ready
```

    - If the endpoint is missing or incorrect, update the application code.

Switch to a Different Probe Type If Needed
If using HTTP, consider using a TCP or command probe:

```yaml
readinessProbe:
  tcpSocket:
    port: 8080
```

3. **Restart and Monitor the Pod**

Delete the pod to force a fresh start:

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

Monitor the pod status:

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

Check Kubernetes Events for More Information

```
kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>
```

4. Look for readiness probe failure messages.

4. Skills Required to Resolve This Issue

- Kubernetes health checks (`readinessProbe`, `livenessProbe`)
- Application debugging inside containers
- YAML configuration for Kubernetes
- Log analysis and troubleshooting

5. Conclusion

The `Readiness Probe Failed` error usually happens due to an incorrect health check, an

unavailable application endpoint, or an application that takes too long to initialize. Adjusting the probe settings, debugging the application, and ensuring the readiness endpoint works correctly can help resolve the issue.

Error 29: CrashLoopBackOff

1. Problem Statement

A pod is stuck in the `CrashLoopBackOff` state, meaning the container inside the pod repeatedly crashes and restarts. This issue prevents the pod from running properly.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12   0/1     CrashLoopBackOff   5          3m
```

Describe the pod for more details:

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

- Look for failure messages.

Check the container logs for error messages:

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

Get events for more debugging information:

```
kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>
```

3. How to Resolve Step by Step

1. **Identify the Root Cause**

Run:

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

   - Look for errors such as:
     - Application startup failure (missing dependencies, misconfigurations)
     - Crash due to missing environment variables
     - Segmentation faults or runtime errors
2. **Check for Misconfigurations**

Get the pod's YAML configuration:

```
kubectl get pod <pod-name> -o yaml -n <namespace>
```

- Look for:
  - Incorrect command or entrypoint in `command` or `args` sections
  - Missing required environment variables
  - Wrong volume mounts

3. **Verify Image and Startup Command**

Check if the container image is valid:

```
kubectl get pod <pod-name> -o=jsonpath='{.spec.containers[0].image}' -n
<namespace>
```

Try running it locally:

```
docker run --rm -it <image-name>
```

- Ensure the application starts correctly outside Kubernetes.

4. **Check Resource Limits**

If the pod crashes due to out-of-memory (OOM) errors, check:

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

- Look for `OOMKilled` events.

Adjust memory limits in the deployment YAML:

```
resources:
  limits:
    memory: "512Mi"
  requests:
    memory: "256Mi"
```

Apply the changes:

```
kubectl apply -f <deployment.yaml>
```

5. **Disable Liveness and Readiness Probes Temporarily**

If probes are misconfigured, try disabling them in the YAML file to confirm:

yaml

```
livenessProbe: null

readinessProbe: null
```

Restart the pod:

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

6. **Force Restart the Pod**

If needed, delete the pod and let Kubernetes recreate it:

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

7. **Check Persistent Volume Mount Issues**
    - If the pod uses persistent volumes, ensure the storage path is correct.

Describe the pod:

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

    - Look for mount failures.

Review Kubernetes Events for More Clues

```
kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>
```

    - Look for errors related to container failures.

4. Skills Required to Resolve This Issue

- Kubernetes troubleshooting (`kubectl logs`, `kubectl describe pod`)
- Debugging containerized applications
- YAML configuration analysis
- Memory and CPU resource management
- Persistent volume debugging

5. Conclusion

The `CrashLoopBackOff` error is often caused by an application crash, misconfiguration, missing dependencies, or resource exhaustion. Analyzing logs, checking configurations, and testing images locally can help diagnose and fix the issue.

Error 30: ErrImagePull / ImagePullBackOff

1. Problem Statement

A pod fails to start because Kubernetes is unable to pull the container image from the registry. The pod status shows `ErrImagePull` or `ImagePullBackOff`.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12   0/1     ErrImagePull       0          1m
my-app-7d5f6b89d7-xyz13   0/1     ImagePullBackOff   1          2m
```

Describe the pod for more details:

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

Look for messages like:

```
Failed to pull image "my-private-registry.com/app:v1": unauthorized
```

- Check for authentication issues if a private registry is used.

3. How to Resolve Step by Step

1. **Verify the Image Name and Tag**

Get the image name from the pod spec:

```
kubectl get pod <pod-name> -o=jsonpath='{.spec.containers[0].image}' -n
<namespace>
```

Manually pull the image on a local machine:

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

If the image does not exist, correct the image reference in the deployment YAML.

2. **Check Network Connectivity**

Try pulling the image manually on a Kubernetes node:

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

- If the pull fails, check network issues or firewall rules blocking registry access.

3. **Verify Docker Hub or Private Registry Authentication**

If using a private registry, check for authentication errors:

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

Ensure a Kubernetes secret exists for private registries:

```
kubectl get secrets -n <namespace>
```

If missing, create a secret:

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

Update the deployment to use the secret:
yaml

```
imagePullSecrets:
  - name: my-registry-secret
```

4. **Check for Incorrect Image Pull Policy**

Ensure `imagePullPolicy` is set correctly:

```
imagePullPolicy: Always
```

Modify the deployment and apply:

```
kubectl apply -f <deployment.yaml>
```

5. **Force Kubernetes to Retry Image Pulling**

Delete the failing pod:

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

- Let Kubernetes recreate the pod and attempt pulling the image again.

6. **Check Kubernetes Node Disk Space**

If the node is out of disk space, image pulling may fail. Check disk usage:

```
df -h
```

- Free up space if needed and restart the node.

7. **Manually Pull the Image on Nodes (Workaround)**

SSH into the Kubernetes node:

```
ssh <node-ip>
```

Pull the image manually:

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

Retag the image if necessary:

```
docker tag <image-name> my-private-registry.com/app:v1
```

8. **Restart the Kubelet Service**

If the node is not correctly pulling images, restart `kubelet`:

```
systemctl restart kubelet
```

4. Skills Required to Resolve This Issue

- Kubernetes pod troubleshooting (`kubectl describe pod`, `kubectl get pods`)
- Docker image management (`docker pull`, `docker tag`)
- Network troubleshooting (`ping`, `curl`)
- Kubernetes secrets and authentication
- Disk space management on Kubernetes nodes

5. Conclusion

The `ErrImagePull` and `ImagePullBackOff` errors typically occur due to incorrect image names, authentication failures, or network issues. Checking logs, verifying registry credentials, and ensuring proper image references can help resolve the issue.

Error 31: CrashLoopBackOff

1. Problem Statement

A pod repeatedly crashes and restarts, showing a `CrashLoopBackOff` status.

2. What Needs to Be Analyzed

Check pod status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12   0/1     CrashLoopBackOff   5          3m
```

Describe the pod for failure details:

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

Look for messages like:

```
Back-off restarting failed container
```

Check pod logs:

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

3. How to Resolve Step by Step

1. **Check Application Logs for Errors**

View logs:

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

If the pod has multiple containers, check logs for each:

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

   - Identify application-specific errors (e.g., database connection failure, missing environment variables).

2. **Check the Exit Code of the Container**

Get detailed information:

```
kubectl get pod <pod-name>
```

```
-o=jsonpath='{.status.containerStatuses[0].state.terminated.exitCode}' -n
<namespace>
```

- Common exit codes:
  - `0`: Normal exit.
  - `1`: General application failure.
  - `137`: OOMKilled (Out of Memory).
  - `139`: Segmentation fault.

3. **Fix Configuration or Environment Variables**

If missing or incorrect environment variables are the issue:

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

Update deployment:

```
env:
  - name: DATABASE_URL
    value: "postgres://user:pass@db-host:5432/mydb"
```

Apply changes:

```
kubectl apply -f <deployment.yaml>
```

4. **Increase Resource Limits (If OOMKilled)**

Check resource usage:

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

If out of memory (`OOMKilled`), update `limits`:
yaml

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Apply changes:

```
kubectl apply -f <deployment.yaml>
```

5. **Check Readiness and Liveness Probes**

If misconfigured, update the probes:

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

- Apply changes and restart the pod.

Manually Restart the Pod

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

6. Kubernetes will recreate the pod.
7. **Check Persistent Volume and File Permissions**

If the application requires a volume:

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

If a file system permission issue occurs, adjust permissions in the Dockerfile:
dockerfile

```
RUN chmod -R 777 /app/data
```

8. **Check Node and Cluster Status**

Ensure the node is healthy:

```
kubectl get nodes
```

Check node disk space:

```
df -h
```

4. Skills Required to Resolve This Issue

- Kubernetes pod debugging (`kubectl logs`, `kubectl describe pod`)
- Resource allocation in Kubernetes (`limits`, `requests`)
- Application debugging (log analysis, exit codes)
- Persistent volume troubleshooting
- Health check probe configuration

5. Conclusion

The `CrashLoopBackOff` error occurs when a container repeatedly fails. By checking logs, exit codes, resource allocation, and configuration issues, you can systematically resolve the problem.

Error 32: ImagePullBackOff

1. Problem Statement

A pod is stuck in the `ImagePullBackOff` state, indicating Kubernetes cannot pull the required container image.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                        READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12     0/1     ImagePullBackOff   0          2m
```

Describe the pod to identify the cause:

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

Look for messages like:

```
Failed to pull image "myregistry.com/app:v1": image not found
```

Check Kubernetes events:

```
kubectl get events -n <namespace>
```

3. How to Resolve Step by Step

1. **Verify the Image Name and Tag**

Ensure the image name and tag in the deployment are correct:

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

   - Check the specified image is available on Docker Hub or the private registry.

2. **Test Image Pull Manually**

On a worker node, run:

```
docker pull <image-name>:<tag>
```

- If the image does not exist, rebuild and push it.

3. **Check Image Registry Credentials (For Private Registries)**

If using a private registry, ensure credentials are correctly configured:

```
kubectl get secret <secret-name> -n <namespace>
```

If missing, create a new secret:

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

Attach the secret to the deployment:

```
imagePullSecrets:
  - name: my-registry-secret
```

4. **Check Network Connectivity**

Ensure nodes can reach the registry:

```
ping <registry-url>
```

- If blocked, update firewall rules or proxy settings.

5. **Restart Kubelet on the Node (If Necessary)**

On the affected node:

```
systemctl restart kubelet
```

Manually Delete and Restart the Pod

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

4. Skills Required to Resolve This Issue

- Kubernetes pod debugging (`kubectl describe pod`, `kubectl get events`)
- Docker image management (`docker pull`, `docker tag`, `docker push`)
- Kubernetes secret management for private registries
- Basic networking for checking registry connectivity

5. Conclusion

`ImagePullBackOff` occurs when Kubernetes cannot pull an image due to incorrect names, missing credentials, or network issues. Verifying the image, credentials, and registry connectivity helps resolve the issue.

Error 33: ErrImageNeverPull

1. Problem Statement

A pod is stuck in the `ErrImageNeverPull` state, meaning Kubernetes is unable to pull the required image due to an image pull policy restriction.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                        READY   STATUS              RESTARTS   AGE
my-app-7d5f6b89d7-xyz12     0/1     ErrImageNeverPull   0          2m
```

Describe the pod to check its image pull policy:

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

Look for a message like:

```
Failed to pull image "my-app:v1": image pull policy prevents pulling image
```

3. How to Resolve Step by Step

1. **Check the Image Pull Policy in the Deployment**

Retrieve the current configuration:

```
kubectl get pod <pod-name>
-o=jsonpath='{.spec.containers[*].imagePullPolicy}' -n <namespace>
```

   - If the pull policy is `Never`, Kubernetes will not try to pull the image.

2. **Update the Image Pull Policy**

If the image is hosted in a registry, update the deployment:

```
image: myregistry.com/app:v1

imagePullPolicy: IfNotPresent
```

   - Available options for `imagePullPolicy`:
     - `Always`: Always pull the image from the registry.
     - `IfNotPresent`: Pull the image only if it does not exist on the node.

- `Never`: Never pull the image, assuming it's already present locally.

3. **Manually Load the Image If Using Local Development**

If running a local development environment, ensure the image exists:

```
docker images
```

If missing, build and load it manually:

```
docker build -t my-app:v1 .
```

```
kind load docker-image my-app:v1
```

Delete and Restart the Pod

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

4. Kubernetes will recreate the pod with the updated settings.
5. **Ensure Registry Access (If Using a Private Registry)**

If the image is private, ensure the correct `imagePullSecrets` is set in the deployment:

```
imagePullSecrets:
  - name: my-registry-secret
```

4. Skills Required to Resolve This Issue

- Kubernetes pod and deployment troubleshooting
- Docker image management
- Understanding image pull policies and private registry authentication

5. Conclusion

`ErrImageNeverPull` occurs due to a restrictive `imagePullPolicy`. Adjusting this setting or ensuring the image is available locally resolves the issue.

Error 34: CrashLoopBackOff

1. Problem Statement

A pod is stuck in a `CrashLoopBackOff` state, meaning the container repeatedly crashes and restarts.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12   0/1     CrashLoopBackOff   5          2m
```

Describe the pod to check events and errors:

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

Look for messages like:

```
Back-off restarting failed container
```

View logs of the crashing container:

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

3. How to Resolve Step by Step

1. **Check Application Logs for Errors**

Get logs for the failing container:

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

If the pod has multiple containers, specify the container:

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

- Look for error messages in the logs.

2. **Verify the Container Command and Arguments**

Check if the container is exiting due to an incorrect entrypoint:

```
command: ["/bin/sh", "-c"]
```

```
args: ["./start-app.sh"]
```

If the script is missing or has incorrect permissions, fix it:

```
chmod +x start-app.sh
```

3. **Inspect Resource Limits**

If the pod is exceeding memory limits, check with:

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

Modify resource limits in the deployment:
yaml

```
resources:
  limits:
    memory: "512Mi"
  requests:
    memory: "256Mi"
```

4. **Ensure Required Dependencies Are Available**

If the pod depends on a database, make sure it's running:

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

- If missing, update the deployment to wait for dependencies.

Manually Restart the Pod

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

Debug in an Interactive Shell

```
kubectl exec -it <pod-name> -n <namespace> -- /bin/sh
```

4. Skills Required to Resolve This Issue

- Kubernetes troubleshooting (`kubectl describe pod`, `kubectl logs`)
- Linux shell debugging (`exec`, `chmod`, `ls -l`)
- Understanding resource limits and application dependencies

5. Conclusion

`CrashLoopBackOff` occurs due to application failures, incorrect commands, or resource issues. Analyzing logs, checking dependencies, and adjusting configurations help resolve the issue.

Error 35: ImagePullBackOff

1. Problem Statement

A pod is stuck in the `ImagePullBackOff` state, meaning Kubernetes is unable to pull the required container image from the specified registry.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                          READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12       0/1     ImagePullBackOff   0          2m
```

Describe the pod to check for errors:

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

Look for error messages like:

```
Failed to pull image "my-app:v1": no such image
```

Verify if the image exists in the registry:

```
docker pull <image-name>:<tag>
```

3. How to Resolve Step by Step

1. **Check Image Name and Tag in Deployment**

Retrieve the current configuration:

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

- Ensure the correct image name and tag are used.

2. **Manually Pull the Image to Test**

Try pulling the image from a local machine:

```
docker pull <image-name>:<tag>
```

- If it fails, verify that the image exists in the registry.

3. **Verify Private Registry Authentication**

If using a private registry, ensure credentials are configured:

```
imagePullSecrets:
  - name: my-registry-secret
```

Create a secret if missing:

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

4. **Check Node Connectivity to the Registry**

Verify the cluster node can reach the registry:

```
curl -v https://index.docker.io/v1/
```

- If there's a network issue, troubleshoot firewall settings.

Manually Restart the Pod

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

If Using a Locally Built Image, Load It into Minikube or Kind

```
kind load docker-image <image-name>:<tag>
```

4. Skills Required to Resolve This Issue

- Kubernetes image management
- Docker image troubleshooting
- Networking and registry authentication

5. Conclusion

`ImagePullBackOff` occurs when Kubernetes cannot pull an image due to incorrect names, missing credentials, or connectivity issues. Ensuring registry access and image correctness resolves the issue.

Error 36: ErrImagePull

1. Problem Statement

A pod is stuck in the `ErrImagePull` state, indicating Kubernetes is unable to pull the specified container image from the registry due to various reasons such as authentication failure, network issues, or incorrect image references.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                        READY   STATUS         RESTARTS   AGE
my-app-7d5f6b89d7-xyz12     0/1     ErrImagePull   0          2m
```

Describe the pod to get detailed error messages:

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

Look for lines like:

```
Failed to pull image "my-app:v1": denied: requested access to the resource
is denied
```

Check if the image exists in the registry:

```
docker pull <image-name>:<tag>
```

- If it fails, the image might not exist or there might be an authentication issue.

3. How to Resolve Step by Step

1. **Verify the Image Name and Tag**

Check the exact image name and tag in your deployment:

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

```
<namespace>
```

- Ensure the correct spelling and tag version.

2. **Check Registry Access**

If using Docker Hub, try logging in and pulling manually:

```
docker login
```

```
docker pull <image-name>:<tag>
```

If using a private registry, verify authentication:

```
kubectl get secret my-registry-secret -n <namespace>
```

3. **Verify ImagePullSecrets Configuration**

Ensure your deployment references the correct image pull secret:

```
imagePullSecrets:
  - name: my-registry-secret
```

If missing, create a secret:

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

4. **Check Network Connectivity to the Registry**

Test if the Kubernetes node can access the registry:

```
curl -v https://index.docker.io/v1/
```

- If failing, check firewall rules and proxy settings.

Restart the Pod

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

If Using Minikube or Kind, Load the Image Locally

```
minikube image load <image-name>:<tag>
```

4. Skills Required to Resolve This Issue

- Kubernetes image pulling mechanisms
- Docker image troubleshooting
- Private registry authentication
- Network diagnostics

5. Conclusion

`ErrImagePull` usually happens due to incorrect image references, missing authentication, or connectivity issues. Verifying credentials, network access, and image existence helps resolve the problem.

Error 37: CrashLoopBackOff

1. Problem Statement

A pod is stuck in the `CrashLoopBackOff` state, meaning it keeps crashing and Kubernetes is repeatedly trying to restart it. This issue usually happens due to application failures, incorrect startup commands, missing dependencies, or insufficient resources.

2. What Needs to Be Analyzed

Check pod status:

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

Example output:

```
NAME                        READY   STATUS             RESTARTS   AGE
my-app-7d5f6b89d7-xyz12     0/1     CrashLoopBackOff   5          3m
```

Describe the pod to check for error messages:

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

Look for lines like:

```
Back-off restarting failed container
```

Check logs to identify why the container is crashing:

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

If the pod has multiple containers, specify the container name:

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

3. How to Resolve Step by Step

1. **Check Application Logs for Errors**
   - Look for stack traces or error messages in the logs.
   - If the application is exiting immediately, check if it's missing environment variables or configurations.
2. **Verify Startup Command in the Pod Definition**

Get the pod's YAML configuration:

```
kubectl get pod <pod-name> -o yaml -n <namespace>
```

- Ensure the `command` and `args` fields are correct.
- If the application needs a specific entrypoint, update it in the deployment.

3. **Check Resource Limits**

If the pod is running out of memory, increase its limits:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Apply the changes:

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

4. **Check for Missing Dependencies**
   - If the application relies on external services (databases, APIs), ensure they are running and accessible.

Test connectivity:

```
kubectl exec -it <pod-name> -- curl http://service-name:port
```

-

Manually Restart the Pod

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

Increase Restart Delay (If Necessary)
If the application needs time before restarting, adjust `livenessProbe` and `readinessProbe`:

```
livenessProbe:
```

```
  initialDelaySeconds: 10

readinessProbe:

  initialDelaySeconds: 15
```

4. Skills Required to Resolve This Issue

- Kubernetes debugging
- Application log analysis
- Resource allocation in Kubernetes
- Networking troubleshooting

5. Conclusion

`CrashLoopBackOff` occurs when a container crashes repeatedly. Checking logs, verifying startup commands, and adjusting resources can help resolve this issue.

Error 38: Pod Stuck in ContainerCreating

1. Problem Statement

A pod is stuck in the `ContainerCreating` state for an extended period, meaning Kubernetes is unable to successfully start the container. This usually happens due to volume mount issues, image pull failures, or networking problems.

2. What Needs to Be Analyzed

Check pod status:

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

Example output:

```
NAME                          READY   STATUS              RESTARTS   AGE
my-app-7d5f6b89d7-xyz12       0/1     ContainerCreating   0          5m
```

Describe the pod to find more details:

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

- Look for messages related to:
  - Volume mounts (`Unable to attach or mount volumes`)
  - Image pull issues (`Failed to pull image`)
  - Network delays (`Timed out waiting for container`)

Check the node where the pod is scheduled:

```
kubectl get pod <pod-name> -o wide -n <namespace>
```

- Verify if the node has any issues.

3. How to Resolve Step by Step

1. **Check for Volume Mount Issues**
   - Look for volume-related errors in `kubectl describe pod`.

Ensure the PersistentVolumeClaim (PVC) is bound:

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

   - If missing, reapply the PVC manifest.
2. **Check Image Pulling Issues**

If `ErrImagePull` is present, verify the image name, tag, and registry access:

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

Test pulling the image manually:

```
docker pull <image-name>:<tag>
```

3. **Verify Node Status**

Check if the assigned node is ready:

```
kubectl get nodes
```

If the node is `NotReady`, investigate:

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

Restart the Pod

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

Check Kubernetes Events for More Clues

```
kubectl get events --sort-by=.metadata.creationTimestamp -n <namespace>
```

4. **Check for Resource Constraints**

If the node lacks resources, schedule the pod on a different node:

```
nodeSelector:
  disktype: ssd
```

4. Skills Required to Resolve This Issue

- Kubernetes volume and storage troubleshooting
- Image pulling and registry authentication
- Node status analysis

5. Conclusion

A pod stuck in `ContainerCreating` indicates an issue with volume mounting, image pulling, or node readiness. Checking pod descriptions, node status, and event logs helps pinpoint the cause.

Error 39: Pending Pods (Pod Stuck in Pending State)

1. Problem Statement

A pod remains in the `Pending` state indefinitely, meaning Kubernetes is unable to schedule it onto a node. This usually occurs due to insufficient cluster resources, node taints, or scheduling constraints.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                        READY   STATUS    RESTARTS   AGE
my-app-7d5f6b89d7-xyz12     0/1     Pending   0          10m
```

Describe the pod to check the scheduling issues:

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

Look for messages like:

```
0/3 nodes are available: insufficient memory, node(s) tainted.
```

Check the nodes in the cluster:

```
kubectl get nodes
```

List pending pods along with their node scheduling reasons:

```
kubectl get pods --field-selector=status.phase=Pending -n <namespace>
```

3. How to Resolve Step by Step

1. **Check Resource Constraints**

If there are no available nodes with enough CPU/memory, increase the cluster size:

```
kubectl top nodes
```

If a pod is requesting too many resources, update the resource requests in the deployment YAML:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Apply the changes:

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

2. **Check for Node Taints and Tolerations**

If the pod is not being scheduled due to a node taint, check taints:

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

If necessary, add tolerations to the pod spec:

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "gpu"
  effect: "NoSchedule"
```

3. **Check Node Selectors and Affinity Rules**
    - If the pod has strict node affinity rules, they may prevent scheduling.

Check if the pod is restricted to a specific node type:

```
kubectl get pod <pod-name> -o yaml
```

    - Modify node affinity rules if needed.
4. **Manually Assign the Pod to an Available Node**

If a specific node has enough resources, assign the pod manually:

```
nodeSelector:
  kubernetes.io/hostname: <node-name>
```

5. **Check if the Cluster is at Capacity**
    - If no nodes are available and scaling is required, increase the cluster size using the cloud provider's auto-scaler.

Restart the Pod to Trigger Rescheduling

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

4. Skills Required to Resolve This Issue

- Kubernetes scheduling and resource allocation
- Cluster scaling and node management
- Debugging taints, tolerations, and affinity

5. Conclusion

A pod stuck in `Pending` usually means there are not enough resources or there are scheduling constraints. Checking node capacity, taints, and affinity rules can help resolve the issue.

Error 40: Node Not Ready

1. Problem Statement

A Kubernetes node is in the `NotReady` state, preventing it from running workloads. This could be due to network failures, disk pressure, memory exhaustion, or kubelet issues.

2. What Needs to Be Analyzed

Check node status:

```
kubectl get nodes
```

Example output:

```
NAME            STATUS     ROLES    AGE   VERSION
node-1          NotReady   worker   45d   v1.25.3
node-2          Ready      worker   50d   v1.25.3
```

Describe the problematic node:

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

- Look for messages such as:
  - `Kubelet stopped posting node status`
  - `NetworkUnavailable=True`
  - `MemoryPressure=True`
  - `DiskPressure=True`

Check system logs for kubelet errors:

```
journalctl -u kubelet -n 50
```

Verify available disk space:

```
df -h
```

Check for network connectivity:

```
ping <master-node-ip>
```

3. How to Resolve Step by Step

1. **Check and Restart Kubelet**

If the kubelet service is down, restart it:

```
systemctl restart kubelet
```

If it fails, check logs:

```
journalctl -u kubelet --no-pager | tail -50
```

2. **Resolve Disk Pressure Issues**

If the node has disk pressure, free up space:

```
du -sh /var/lib/docker
```

```
rm -rf /var/lib/docker/containers/*
```

   - Check `evictionThreshold` settings in kubelet config.

3. **Verify Network Configuration**

Restart the networking service:

```
systemctl restart networking
```

Ensure `cni` plugins are installed:

```
ls /opt/cni/bin/
```

4. **Drain and Rejoin the Node**

Drain workloads from the node:

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

Mark the node as schedulable again:

```
kubectl uncordon <node-name>
```

5. **Restart the Node If Necessary**

If all else fails, reboot the node:

```
reboot
```

4. Skills Required to Resolve This Issue

- Kubernetes node troubleshooting
- Linux system administration
- Networking and kubelet debugging

5. Conclusion

A node in the `NotReady` state can disrupt workload scheduling. Restarting the kubelet, fixing disk/network issues, and properly maintaining the node ensures stability.

Error 41: ImagePullBackOff (Pod Failing to Pull Image)

1. Problem Statement

A pod fails to start because Kubernetes is unable to pull the container image from the registry. The pod status shows `ImagePullBackOff` or `ErrImagePull`.

2. What Needs to Be Analyzed

Check the pod's status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-5678d89f6d-klm45   0/1     ImagePullBackOff   0          5m
```

Describe the pod for error details:

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

Look for messages like:

```
Failed to pull image "myregistry.com/app:v1": error parsing HTTP 403
response body

Back-off pulling image "myregistry.com/app:v1"
```

Check if the image is available:

```
docker pull myregistry.com/app:v1
```

Verify container registry authentication:

```
kubectl get secret <secret-name> -n <namespace>
```

-

3. How to Resolve Step by Step

1. **Ensure the Image Name is Correct**

Verify that the image name and tag exist in the registry:

```
docker pull myregistry.com/app:v1
```

2. **Check Private Registry Authentication**

If pulling from a private registry, ensure you have an image pull secret:

```
kubectl create secret docker-registry regcred \
  --docker-server=<your-registry-server> \
  --docker-username=<your-username> \
  --docker-password=<your-password> \
  --docker-email=<your-email>
```

Attach it to the deployment:

```
imagePullSecrets:
  - name: regcred
```

3. **Verify Network Access to the Registry**

Check if the node can reach the registry:

```
curl -v https://myregistry.com/v2/
```

   - If access is denied, check firewall and proxy settings.

4. **Allow Kubernetes to Retry the Image Pull**

Restart the pod:

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

5. **Ensure the Image Policy is Correct**

Modify the deployment YAML if necessary:

```
imagePullPolicy: Always
```

4. Skills Required to Resolve This Issue

- Kubernetes image management
- Docker container registry
- Networking and authentication debugging

5. Conclusion

The `ImagePullBackOff` error typically occurs due to incorrect image names, registry authentication issues, or network connectivity problems. Ensuring proper authentication and access helps resolve it.

Error 42: CrashLoopBackOff (Pod Continuously Restarting)

1. Problem Statement

A pod is stuck in a `CrashLoopBackOff` state, meaning it repeatedly starts and crashes due to an underlying issue such as application failure, misconfiguration, or resource constraints.

2. What Needs to Be Analyzed

Check the pod's status:

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

Example output:

```
NAME                      READY   STATUS             RESTARTS   AGE
my-app-5678d89f6d-klm45   0/1     CrashLoopBackOff   5          10m
```

Describe the pod for error messages:

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

Look for reasons like:

```
Back-off restarting failed container
```

Check logs to diagnose the issue:

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

Example output:

```
Error: Cannot connect to database
```

Verify resource limits:

```
kubectl get pod <pod-name> -n <namespace> -o
jsonpath='{.spec.containers[*].resources}'
```

-

3. How to Resolve Step by Step

1. **Check Application Logs for Errors**

Identify the root cause by checking logs:

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

2. **Increase Resource Limits if Needed**

If the pod is running out of memory or CPU, update resource requests in the deployment YAML:

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

3. **Fix Configuration or Environment Variable Issues**

Verify if the pod requires missing environment variables:

```
kubectl get deployment <deployment-name> -n <namespace> -o yaml
```

   - Correct any misconfigured values.

4. **Ensure the Application Can Start Properly**

Run the container locally:

```
docker run my-app:v1
```

   - Check for missing dependencies or runtime issues.

5. **Check Health Probes**

If liveness or readiness probes are misconfigured, update them in the deployment:

```
livenessProbe:
```

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

6. **Restart the Pod**

If the issue is resolved, delete the pod to allow a fresh start:

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

4. Skills Required to Resolve This Issue

- Kubernetes pod debugging
- Application troubleshooting
- YAML configuration management

5. Conclusion

The `CrashLoopBackOff` error is often caused by application failures, misconfigured environment variables, or resource constraints. By analyzing logs and adjusting settings, the issue can be resolved efficiently.

Error 43: Pod Stuck in Terminating State

1. Problem Statement

A pod is stuck in the `Terminating` state for an extended period, preventing it from being deleted or replaced. This issue usually occurs due to finalizers, pending processes, or stuck network connections.

2. What Needs to Be Analyzed

Check the pod's status:

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

Example output:

```
NAME                      READY   STATUS        RESTARTS   AGE
my-app-5678d89f6d-klm45   0/1     Terminating   0          30m
```

Describe the pod for any finalizers or stuck processes:

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

Look for:

```
Finalizers: kubernetes.io/pvc-protection
```

Check if the pod is using a persistent volume that is preventing deletion:

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

Look for processes still running inside the container:

```
kubectl exec -it <pod-name> -n <namespace> -- ps aux
```

3. How to Resolve Step by Step

Try a Normal Pod Deletion

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

Force Delete the Pod if Normal Deletion Fails

```
kubectl delete pod <pod-name> -n <namespace> --force --grace-period=0
```

1. **Remove Finalizers Manually (If Applicable)**

Edit the pod's YAML to remove the `finalizers` section:

```
kubectl get pod <pod-name> -n <namespace> -o json > pod.json
```

Open `pod.json` and delete the `finalizers` section, then apply the changes:

```
kubectl replace --force -f pod.json
```

2. **Restart Kubelet on the Node (If the Pod Persists)**

Find the node where the pod is running:

```
kubectl get pod <pod-name> -n <namespace> -o wide
```

SSH into the node and restart Kubelet:

```
systemctl restart kubelet
```

Drain the Node if the Pod Is Still Stuck

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

Verify That the Pod Has Been Successfully Deleted

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

4. Skills Required to Resolve This Issue

- Kubernetes pod and node management
- YAML and JSON editing
- Kubernetes storage and finalizer handling

5. Conclusion

A pod stuck in the `Terminating` state is usually caused by finalizers, persistent volumes, or hanging processes. By force-deleting the pod, removing finalizers, or restarting Kubelet, you can resolve the issue.

Error 44: Node Not Ready

1. Problem Statement

A node is in the `NotReady` state, meaning it is not scheduling new pods and might be having issues with connectivity, resource exhaustion, or Kubelet failures.

2. What Needs to Be Analyzed

Check the node status:

```
kubectl get nodes
```

Example output:

```
NAME          STATUS     ROLES    AGE     VERSION
node-1        NotReady   worker   10d     v1.27.2
node-2        Ready      worker   15d     v1.27.2
```

Describe the node for detailed errors:

```
kubectl describe node node-1
```

Look for messages like:

```
Kubelet stopped posting node status
Out of disk space
Network unreachable
```

Check Kubelet logs on the node:

```
journalctl -u kubelet -f --no-pager
```

Verify disk space and memory usage:

```
df -h
```

```
free -m
```

Check if networking issues exist:

```
ping <control-plane-ip>
```

3. How to Resolve Step by Step

1. **Restart Kubelet**

SSH into the node and restart the service:

```
systemctl restart kubelet
```

2. **Check and Free Up Disk Space**

Remove unnecessary files:

```
du -sh /var/lib/docker /var/log
```

```
rm -rf /var/log/*.log
```

Clear container images:

```
docker system prune -a
```

3. **Verify Network Connectivity**

Restart network services:

```
systemctl restart networking
```

If using Calico or Flannel, restart the CNI plugin:

```
systemctl restart calico-node
```

4. **Drain and Rejoin the Node**

If the node is still not working, drain it:

```
kubectl drain node-1 --ignore-daemonsets --delete-emptydir-data
```

Remove and rejoin the node:

```
kubeadm reset

kubeadm join <control-plane-ip>:6443 --token <token>
--discovery-token-ca-cert-hash sha256:<hash>
```

Check Node Status Again

```
kubectl get nodes
```

4. Skills Required to Resolve This Issue

- Kubernetes node troubleshooting
- Linux system administration
- Networking debugging

5. Conclusion

A `NotReady` node can disrupt cluster operations. Checking logs, restarting services, and fixing networking or disk issues can help restore the node's functionality.

Error 45: Unable to Attach or Mount Volumes - Timed Out Waiting for Condition

1. Problem Statement

A pod is stuck in the `ContainerCreating` state with an error message related to volume attachment or mounting issues. This typically occurs due to storage misconfigurations, unresponsive storage backends, or node-related issues.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                     READY   STATUS              RESTARTS   AGE
my-app-5678d89f6d-klm45  0/1     ContainerCreating   0          10m
```

Describe the pod for error details:

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

Look for messages like:

```
Warning  FailedMount  2m (x10 over 5m)  kubelet  Unable to attach or mount volumes: timed out waiting for the condition
```

Check if the Persistent Volume (PV) and Persistent Volume Claim (PVC) are bound:

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

```
kubectl get pv
```

- Look for `Pending` status.

Verify if the node can access the storage:

```
df -h
```

```
lsblk
```

Check Kubelet logs for volume-related errors:

```
journalctl -u kubelet -f --no-pager
```

3. How to Resolve Step by Step

1. **Verify the Storage Class Configuration**

Check if the storage class is available:

```
kubectl get storageclass
```

   - Ensure the correct `provisioner` is configured in the PVC spec.

2. **Manually Rebind the PVC**

Delete and recreate the PVC if it's stuck:

```
kubectl delete pvc <pvc-name> -n <namespace>
```

```
kubectl apply -f <pvc-definition>.yaml
```

3. **Manually Attach the Volume to the Node (If Using External Storage)**

Check if the volume exists on the cloud provider:

```
kubectl get pv | grep <volume-id>
```

   - Manually attach the volume using the cloud provider CLI (AWS, GCP, Azure).

Restart Kubelet on the Affected Node

```
systemctl restart kubelet
```

Drain and Rejoin the Node (If the Issue Persists)

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

```
kubeadm reset
```

```
kubeadm join <control-plane-ip>:6443 --token <token>
--discovery-token-ca-cert-hash sha256:<hash>
```

Verify That the Volume Is Mounted Successfully

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

4. Skills Required to Resolve This Issue

- Kubernetes storage troubleshooting
- Persistent volume debugging

- Cloud provider-specific storage management

5. Conclusion

This error occurs when Kubernetes cannot mount a volume due to misconfigurations, node failures, or storage backend issues. Checking PV, PVC, and storage classes, and restarting affected nodes can resolve the problem.

Error 46: Pod Not Scheduling Due to Insufficient Resources

1. Problem Statement

A pod is not being scheduled on any node due to insufficient resources like CPU, memory, or disk space. The pod remains in the `Pending` state and fails to find an available node to run on.

2. What Needs to Be Analyzed

Check the pod's status:

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

Example output:

```
NAME                     READY   STATUS    RESTARTS   AGE
my-app-5678d89f6d-klm45  0/1     Pending   0          10m
```

Describe the pod to get detailed error information:

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

Look for messages like:

```
Warning  FailedScheduling  1m (x5 over 10m)  default-scheduler  0/3 nodes
are available: 1 Insufficient cpu, 1 Insufficient memory.
```

Check resource requests and limits in the pod specification:

```
kubectl get pod <pod-name> -n <namespace> -o yaml
```

Inspect node resource availability:

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

- Look for available CPU and memory capacity in the node description.

3. How to Resolve Step by Step

1. **Ensure Adequate Resource Requests and Limits for the Pod**

- Check if the pod's resource requests exceed the available capacity on the nodes.

Modify the pod resource requests and limits to fit within available resources:

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

- Apply the changes to the pod spec.

2. **Check Node Resources and Schedule Pod on a Different Node**

If specific nodes have insufficient resources, either add more nodes to the cluster or schedule the pod to nodes with sufficient resources:

```
kubectl get nodes
```

```
kubectl label node <node-name> node-type=high-cpu
```

Then modify the pod spec to match the new node label:

```
nodeSelector:
  node-type: high-cpu
```

3. **Increase Node Capacity (If Possible)**
   - If nodes are consistently low on resources, consider scaling the cluster by adding more nodes or upgrading existing nodes.
4. **Enable Resource Autoscaling**

Use the Horizontal Pod Autoscaler (HPA) to automatically adjust the number of pods based on resource usage:

```
kubectl autoscale deployment <deployment-name> --cpu-percent=50 --min=1
--max=10
```

5. **Ensure Proper Affinity Rules and Anti-Affinity Settings**

Review the pod's affinity settings to ensure that the pod can be scheduled properly across available nodes:
yaml

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

- ○

Verify Pod Scheduling After Fixes

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

4. Skills Required to Resolve This Issue

- Kubernetes resource management
- Node resource allocation and scaling
- Affinity and anti-affinity configuration

5. Conclusion

Pods failing to schedule due to insufficient resources can be resolved by adjusting resource requests, using proper affinity rules, scaling the cluster, or modifying node configurations to provide adequate resources for pod scheduling.

Error 47: Pod CrashLoopBackOff

1. Problem Statement

A pod is repeatedly crashing and restarting, displaying a `CrashLoopBackOff` error. This typically occurs when the application inside the container fails to start or encounters errors after starting, causing Kubernetes to restart the container multiple times.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                     READY   STATUS             RESTARTS   AGE
my-app-5678d89f6d-klm45  0/1     CrashLoopBackOff   5          10m
```

Describe the pod to get detailed error information:

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

Look for messages like:

```
Back-off restarting failed container
```

Check the logs of the failing container:

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

If the pod is part of a deployment, check the deployment status:

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

3. How to Resolve Step by Step

1. **Check Application Logs for Errors**
   - Review the container logs to identify the root cause of the application crash. Common issues include incorrect environment variables, missing dependencies, or configuration errors.

2. **Increase the Startup Probe Timeout**

If the container takes longer to start, increase the `initialDelaySeconds` and `timeoutSeconds` for the `livenessProbe` and `readinessProbe` in the pod spec:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds:
```

3. **Ensure Correct Resource Requests and Limits**

Verify that the resource limits and requests are appropriately set. If the container is being killed due to resource exhaustion, adjust the resource allocations:

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

4. **Check for Missing Configurations or Secrets**
    - Ensure that all necessary environment variables, ConfigMaps, and Secrets are correctly configured and available to the pod.
5. **Debug with Interactive Shell**

If the pod starts but fails after some time, you can run an interactive shell inside the pod to debug further:

```
kubectl exec -it <pod-name> -n <namespace> -- /bin/bash
```

6. **Verify Container Image and Version**

Confirm that the container image and tag are correct, and try pulling the image manually to see if the issue is related to the image itself:

```
docker pull <image-name>:<tag>
```

7. **Review Kubernetes Events**

Check the Kubernetes events for more clues:

```
kubectl get events -n <namespace>
```

Verify Pod After Fixes

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

4. Skills Required to Resolve This Issue

- Application debugging
- Kubernetes liveness and readiness probes
- Resource management in Kubernetes
- Container and image troubleshooting

5. Conclusion

A `CrashLoopBackOff` error typically indicates application-level issues such as misconfiguration or resource exhaustion. Identifying the root cause via logs, resource adjustments, and configuration checks can help resolve the problem and stabilize the pod.

Error 48: Pod Termination Due to OOMKilled (Out of Memory)

1. Problem Statement

A pod terminates unexpectedly with the `OOMKilled` status, indicating that the container has exceeded its memory limits and was terminated by the Kubernetes out-of-memory (OOM) killer.

2. What Needs to Be Analyzed

Check the pod status:

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

Example output:

```
NAME                     READY   STATUS      RESTARTS   AGE
my-app-5678d89f6d-klm45  0/1     OOMKilled   3          15m
```

Describe the pod to get detailed error information:

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

Look for messages like:

```
State:          Terminated
  Reason:       OOMKilled
  Exit Code:    137
  Memory Usage: 500Mi
```

Check the pod's resource requests and limits:

```
kubectl get pod <pod-name> -n <namespace> -o yaml
```

Check the logs for memory-related errors or application logs:

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

3. How to Resolve Step by Step

1. **Increase Memory Limits and Requests**

Ensure that the pod's memory requests and limits are correctly set to accommodate the application's

memory requirements. For example:

```
resources:
  requests:
    memory: "1Gi"
    cpu: "500m"
  limits:
    memory: "2Gi"
    cpu: "1"
```

- Apply the updated spec.

2. **Optimize Application Memory Usage**
   - If the application is memory-intensive, review its code and optimize memory usage. This could involve:
     - Reducing memory consumption per request.
     - Avoiding memory leaks.
     - Implementing caching mechanisms to reduce memory overhead.
3. **Enable Resource Requests and Limits on All Containers**
   - Ensure that every container in the pod has both resource `requests` and `limits` set. Not setting them might cause resource contention and lead to OOMKill events.
4. **Use Memory Limits and Horizontal Pod Autoscaling**

Enable horizontal scaling for the application to handle higher loads by scaling the number of pods instead of increasing memory limits. You can configure the Horizontal Pod Autoscaler (HPA) to scale the application based on CPU or memory usage:
sh

```
kubectl autoscale deployment <deployment-name> --cpu-percent=80 --min=2
--max=5
```

-

5. **Monitor Resource Usage**
   - Monitor memory usage with tools like `kubectl top pods` or Prometheus/Grafana dashboards to track memory usage trends and adjust limits accordingly.
6. **Check for Memory Leaks**
   - If the issue persists, consider analyzing the application for memory leaks or inefficiencies in memory handling.

Verify Pod After Fixes

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

4. Skills Required to Resolve This Issue

- Application memory profiling
- Kubernetes resource management
- Horizontal Pod Autoscaling (HPA)
- Container troubleshooting

5. Conclusion

Pods terminating with `OOMKilled` are usually a result of memory resource issues. By adjusting resource limits, optimizing memory usage, or scaling the application, the pod's stability can be restored.

Error 49: Kubernetes Node Not Ready

1. Problem Statement

A node in the Kubernetes cluster is marked as `NotReady`. This prevents pods from being scheduled on the node, which may lead to disruptions in the cluster's overall functionality.

2. What Needs to Be Analyzed

Check the node status:

```
kubectl get nodes
```

Example output:

```
NAME                STATUS   ROLES    AGE   VERSION
node-1.example.com  NotReady  <none>   10d   v1.20.7
```

Describe the node to gather more detailed information:

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

Look for messages under the `Conditions` section like:

```
Condition:
  Type:              Ready
  Status:            False
  LastHeartbeatTime: <timestamp>
  Reason:            KubeletNotReady
  Message:           Kubelet stopped posting node status.
```

Check the node's logs:

```
journalctl -u kubelet -f
```

3. How to Resolve Step by Step

1. **Verify Kubelet Service**

Ensure that the `kubelet` service is running on the node:

```
systemctl status kubelet
```

If the kubelet service is down, restart it:

```
sudo systemctl restart kubelet
```

2. **Check Node Resources**
   - Ensure that the node has enough available resources (CPU, memory, disk space). If resources are exhausted, you may need to scale the node or free up resources.
3. **Verify Network Connectivity**
   - Check for network issues that may prevent the node from communicating with the Kubernetes API server. Verify that the node can reach the API server by pinging it or checking firewall rules.
4. **Investigate Kubelet Logs**

If the kubelet is not responding, check the logs for issues related to network or configuration problems:

```
journalctl -u kubelet -f
```

5. **Check for Disk Pressure**
   - Verify that the node is not under disk pressure, which can prevent kubelet from posting the node status. If disk space is low, clean up unused Docker images or increase disk capacity.
6. **Reboot Node**

As a last resort, if the issue persists after troubleshooting, reboot the node to clear potential temporary issues:

```
sudo reboot
```

Check Node Status After Fixes

```
kubectl get nodes
```

4. Skills Required to Resolve This Issue

- Kubernetes node troubleshooting
- Systemd service management (kubelet)
- Resource management and monitoring
- Network and connectivity troubleshooting

5. Conclusion

A node marked as `NotReady` can be caused by multiple issues, including kubelet failures, resource exhaustion, or network issues. By systematically investigating logs, services, and node resources, the issue can be resolved and the node brought back to a `Ready` state.

Error 50: Persistent Volume Claim (PVC) Pending

1. Problem Statement

A Persistent Volume Claim (PVC) remains in the `Pending` state for an extended period, which means the requested volume cannot be provisioned by Kubernetes. This prevents workloads from running that depend on persistent storage.

2. What Needs to Be Analyzed

Check the PVC status:

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

Example output:

```
NAME          STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
AGE

my-pvc        Pending   <none>   <none>     RWO            standard
10m
```

Describe the PVC to get detailed error messages:

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

Look for events like:

```
Warning  ProvisioningFailed  3m (x3 over 10m)  persistentvolume-controller
Failed to provision volume with StorageClass "standard": no matches for kind
"StorageClass"
```

Check available Persistent Volumes (PVs) and their status:

```
kubectl get pv
```

- Look for unbound PVs or ones that don't match the PVC's requested storage class or access modes.

3. How to Resolve Step by Step

1. **Check StorageClass Configuration**
   - Ensure that the PVC requests the correct `StorageClass`. If no `StorageClass` is specified, the default one is used.

Verify that the requested `StorageClass` exists by checking:

```
kubectl get storageclass
```

2. **Verify Available PVs**
   - Check if there are any unbound PVs available that can satisfy the PVC's request. If none are available, create a new PV that matches the PVC's requirements or configure dynamic provisioning.
3. **Dynamic Provisioning**
   - If the cluster supports dynamic provisioning, ensure the storage class has the correct provisioner configured. For example, the `nfs-provisioner` or `aws-ebs` provisioner should be defined.
   - Check the provisioner's configuration to ensure it is working correctly.
4. **Check PV Access Modes and Size**
   - Ensure that the PV's `accessModes` and `storage` capacity match the PVC's request. PVC requests, for example, may require `ReadWriteOnce` (RWO) access, but a PV may only support `ReadOnlyMany` (ROX).
5. **Provision a New PV**

If no suitable PV exists, create a new PV that meets the PVC's requirements:

```
apiVersion: v1

kind: PersistentVolume

metadata:

  name: my-new-pv

spec:

  capacity:

    storage: 10Gi

  accessModes:

    - ReadWriteOnce

  persistentVolumeReclaimPolicy: Retain

  storageClassName: standard

  hostPath:

    path: /mnt/data
```

6. **Check PVC and PV Binding**

Ensure the PVC and PV are correctly bound. You can manually bind them by setting the `volumeName`

in the PVC spec if needed:

```
volumeName: my-new-pv
```

Monitor PVC Status After Fixes

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

4. Skills Required to Resolve This Issue

- Persistent Storage Management in Kubernetes
- Understanding of StorageClass, PV, and PVC concepts
- Troubleshooting Kubernetes storage issues
- Dynamic volume provisioning

5. Conclusion

A PVC remaining in the `Pending` state typically indicates an issue with volume provisioning, either due to a lack of available PVs or mismatched configuration. By ensuring the correct storage class, available volumes, and correct provisioning configuration, the PVC can be successfully bound to a persistent volume.


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

For any RealTime Handson Projects

And for more tips like this

Follow

Like & ReShare

@careerbytecode