Troubleshooting in Kubernetes

You face any issue in kubernetes,

first step is to check if kubernetes *self applications* are running fine or not.

Command:- ***kubectl get pods -n kube-system***

→ If you see any pod is crashing, check it's logs

→ if getting NotReady state error, verify network pod logs.

if not able to resolve with above, follow below steps:-

- [ ] kubectl get nodes - Check which node is not in ready state
- [ ] kubectl describe node nodename - nodename which is not in readystate
- [ ] ssh to that node
- [ ] execute systemctl status kubelet - Make sure kubelet is running
- [ ] systemctl status docker - Make sure docker service is running
- [ ] journalctl -u kubelet - To Check logs in depth

Most probably you will get to know about error here, After fixing it reset kubelet with below commands:-

systemctl daemon-reload

systemctl restart kubelet

In case you still didn't get the root cause, check below things:-

- [ ] Make sure your node has enough space and memory. Check for /var directory space especially. command to check: -df -kh, free -m
- [ ] Verify cpu utilization with top command and make sure any process is not taking an unexpected memory.

Will discuss, 5 commonaly known errors

#1. Resource Exhausted error

Possible Causes

1. **Insufficient Node Resources:** The cluster nodes do not have enough CPU or memory to schedule new pods.
2. **Improper Resource Requests and Limits:** Pods are requesting more resources than available or no resource limits are defined, leading to over-utilization.
3. **High Cluster Workload:** The cluster is running too many pods, causing resource contention.
4. **Evictions:** Nodes evict pods to reclaim resources due to resource pressure.

Step-by-Step Resolution

📌 Identify the Cause

- **Check Pod Events:**

kubectl describe pod <pod-name>

Look for messages like Insufficient CPU or Insufficient memory.

- **Check Node Status:**

kubectl get nodes

kubectl describe node <node-name>

Review node resource capacity and utilization.

- **Monitor Resource Usage:** Use metrics-server or tools like Prometheus and Grafana to monitor CPU and memory usage across the cluster.

📌 Verify Resource Requests and Limits

- Ensure that each pod has proper resource requests and limits defined.

```
resources:
  requests:
    memory: "128Mi"
    cpu: "500m"
  limits:
    memory: "256Mi"
    cpu: "1000m"
```

- Check existing pod configurations:

kubectl get pods -o yaml

📌 Adjust Resource Requests and Limits

- **Modify Resource Configuration:** Update resource requests and limits in the pod specification to match actual needs.

kubectl edit deployment <deployment-name>

📌 Scale the Application

- If the workload is high, scale up the application using replicas:

kubectl scale deployment <deployment-name> --replicas=<number>

📌 Add More Nodes to the Cluster

- **For On-Prem Clusters:** Add more nodes manually.
- **For Cloud Providers:** Enable cluster auto-scaling. Example (GKE):

gcloud container clusters update <cluster-name> --enable-autoscaling --min-nodes=<min> --max-nodes=<max>

📌 Implement Resource Quotas

- Limit resource usage per namespace to prevent overuse:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota
namespace: <namespace>
spec:
hard:
  requests.cpu: "4"
  requests.memory: "8Gi"
  limits.cpu: "8"
  limits.memory: "16Gi"
```


📌 Enable Cluster Autoscaler

- Use Kubernetes Cluster Autoscaler to dynamically add nodes when resources are exhausted.

📌 Optimize Existing Resources

- Identify and terminate unused pods.
- Reduce resource limits for less critical applications.

📌 Monitor and Prevent Future Issues

- Set up alerting for high resource utilization.
- Use Vertical Pod Autoscaler (VPA) or Horizontal Pod Autoscaler (HPA) for dynamic scaling.

By identifying the root cause, ensuring proper resource configuration, and scaling your cluster appropriately, you can resolve and prevent "Resource Exhausted" errors. Regular monitoring and proactive resource management are essential to maintaining a stable Kubernetes environment.

#2 CrashLoopBackoff error

When a Kubernetes container repeatedly fails to start, it enters a 'CrashLoopBackOff' state, indicating a restart loop within a pod. This error often occurs due to various issues preventing the container from launching properly.

To confirm this error, execute

kubectl get pods and

verify if the pod status is '**CrashLoopBackOff**'.

How Does CrashLoopBackOff Work?

By default, a pod's restart policy is **Always**, meaning it should always restart on failure (Options: Never/OnFailure/Always). Depending on the restart policy defined in the pod template, Kubernetes might try to restart the pod multiple times.

When a Pod state is displaying CrashLoopBackOff, it means that it's currently waiting the indicated time before restarting the pod again.

Every time the pod is restarted, Kubernetes waits for a longer and longer time, known as a **backoff delay**. The delay between restarts is exponential (10s, 20s, 40s, ...) and is capped at five minutes. During this process, Kubernetes displays the CrashLoopBackOff error.

Causes of CrashLoopBackOff

CrashLoopBackOff can occur when a pod fails to start for some reason, because a container fails to start up properly and repeatedly crashes. Let's review the common causes of this issue.

Resource Overload or Insufficient Memory

One of the common causes of the CrashLoopBackOff error is resource overload or insufficient memory. Kubernetes allows setting memory and CPU usage limits for each pod, which means your application might be crashing due to insufficient resources. This might happen due to:

1. memory leaks in your program,
2. misconfigured resource requests and limits, or
3. your application requires more resources than are available on the node.

When the node that the pod is running on doesn't have enough resources, the pod can be evicted and moved to a different node. If none of the nodes have sufficient resources, the pod can go into a ***CrashLoopBackOff*** *state.*

To resolve this issue, you need to understand the resource usage of your application and set the appropriate resource requests and limits. You can use the kubectl describe pod [pod_name] command to check if the pod was evicted due to insufficient memory.

You can also monitor the memory and CPU usage of your pods using Kubernetes metrics server or other monitoring tools like Prometheus. If your application is consistently using more resources than allocated, you might need to optimize your application, allocate more resources, or change resources:limits in the Container's resource manifest.

Errors When Deploying Kubernetes

A common reason pods in your Kubernetes cluster display a CrashLoopBackOff message is that Kubernetes springs deprecated versions of Docker. You can reveal the Docker version using -v checks against the containerization tool.

A best practice for fixing this error is ensuring you have the latest Docker version and the most stable versions of other plugins. Thus, you can prevent deprecated commands and inconsistencies that trip your containers into start-fail loops.

When migrating a project into a Kubernetes cluster, you might need to roll back several Docker versions to meet the incoming project's version.

Issue with Third-Party Services (DNS Error)

Sometimes, the CrashLoopBackOff error is caused by an issue with one of the third-party services. If this is the case, upon starting the pod you'll see the message:

send request failure caused by: Post

Check the syslog and other container logs to see if this was caused by any of the issues we mentioned as causes of CrashLoopBackoff (e.g., locked or missing files). If not, then the problem could be with one of the third-party services.

To verify this, you'll need to use a debugging container. A debug container works as a shell that can be used to login into the failing container. This works because both containers share a similar environment, so their behaviors are the same. Here is a link to one such shell you can use: ubuntu-network-troubleshooting.

Using the shell, log into your failing container and begin debugging as you normally would. Start with checking kube-dns configurations, since a lot of third-party issues start with incorrect DNS settings.

Missing Dependencies

The CrashLoopBackOff status can activate when Kubernetes cannot locate runtime dependencies (i.e., the var, run, secrets, kubernetes.io, or service account files are missing). This might occur when some containers inside the pod attempt to interact with an API without the default access token.

This scenario is possible if you manually create the pods using a unique API token to access cluster services. The missing service account file is the declaration of tokens needed to pass authentication.

You can fix this error by allowing all new –mount creations to adhere to the default access level throughout the pod space. Ensure that new pods using custom tokens comply with this access level to prevent continuous startup failures.

Changes Caused by Recent Updates

If you constantly update your clusters with new variables that spark resource requirements, they will likely encounter CrashLoopBackOff failures.

Suppose you have a shared master setup and run an update that restarts all the pod services. The result is several restart loops because Kubernetes must choose a master from the available options. You can fix this by changing the update procedure from a direct, all-encompassing one to a sequential one (i.e., applying changes separately in each pod). This approach makes it easier to troubleshoot the cause of the restart loop.

In some cases, CrashLoopBackOff can occur as a settling phase to the changes you make. The error resolves itself when the nodes eventually receive the right resources for a stable environment.

Container Failure due to Port Conflict

Let’s take another example in which the container failed due to a port conflict. To identify the issue you can pull the failed container by running docker logs [container id].

Doing this will let you identify the conflicting service. Using netstat, look for the corresponding container for that service and kill it with the kill command. Delete the kube-controller-manager pod and restart.

Troubleshoot CrashLoopBackOff

The best way to identify the root cause of the error is to start going through the list of potential causes and eliminate them one by one, starting with the most common ones first.

1. Check for “Back Off Restarting Failed Container”

Run kubectl describe pod [name].

⦿If you get a Liveness probe failed and Back-off restarting failed container messages from the kubelet, as shown below, this indicates the container is not responding and is in the process of

restarting.

From Message

----- -----

kubelet Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory

kubelet Back-off restarting failed container

⊘If you get the back-off restarting failed container message this means that you are dealing with a temporary resource overload, as a result of an activity spike. The solution is to adjust periodSeconds or timeoutSeconds to give the application a longer window of time to respond.

If this was not the issue, proceed to the next step.

2. Check Logs From Previous Container Instance

If Kubernetes pod details didn't provide any clues, your next step should be to pull information from the previous container instance.

You originally ran kubectl get pods to identify the Kubernetes pod that was exhibiting the CrashLoopBackOff error. You can run the following command to get the last 5 log lines from the pod:

kubectl logs --previous --tail 5

Search the log for clues showing why the pod is repeatedly crashing. If you cannot resolve the issue, proceed to the next step.

3. Check Deployment Logs

Run the following command to retrieve the kubectl deployment logs:

kubectl logs -f deploy/ -n

This may also provide clues about issues at the application level.

Failing all the above, the next step is to bash into the CrashLoop container to see exactly what happened.

Tips for Prevention

In most cases, restarting the pod and deploying a new version will resolve the problem and keep the application online.

However, it is important to identify the root cause of the CrashLoopBackOff error and prevent it in the first place.

Best practices that can help you prevent the CrashLoopBackOff error.

1. Configure and Recheck Your Files

A misconfigured or missing configuration file can cause the CrashLoopBackOff error, preventing the container from starting correctly. Before deployment, make sure all files are in place and configured correctly.

In most cases, files are stored in /var/lib/docker. You can use commands like ls and find to verify if the target file exists.

2. Be Vigilant With Third-Party Services

If an application uses a third-party service and calls made to a service fail, then the service itself is the problem. Most errors are usually due to an error with the *SSL certificate or network issues*, so make sure those are functioning correctly.

You can log into the container and manually reach the endpoints using curl to check.

3. Check Your Environment Variables

Incorrect environment variables are a common cause of the CrashLoopBackOff error. A common occurrence is when containers require Java to run, but their environment variables are not set properly.

Use env to inspect the environment variables and make sure they're correct.

4. Check Kube-DNS

The application may be trying to connect to an external service, but the kube-dns service is not running.

You just need to restart the kube-dns service so the container can connect to the external service.

5. Check File Locks

As mentioned before, file locks are a common reason for the CrashLoopBackOff error. Ensure you inspect all ports and containers to see that none are being occupied by the wrong service. If they are, kill the service occupying the required port.

#3 OOMKilled error

OOMKilled (Out of Memory Killed) occurs in Kubernetes when a container exceeds the memory limit specified in its resource configuration. The Kubernetes scheduler, in coordination with the kubelet, monitors container resource usage.

When a container uses more memory than its allocated limit, the kernel's **Out of Memory (OOM) Killer** terminates the container to protect the system from running out of memory.

Possible Reasons Behind OOMKilled

1. **Insufficient Memory Limits**
   The memory limit specified in the container's resource configuration is too low for its actual workload.
2. **Memory Leak in Application**
   The application running inside the container has a memory leak, causing it to consume increasing amounts of memory over time.
3. **Unexpected High Workload**
   A sudden surge in traffic or workload might cause the application to use more memory than anticipated.
4. **Improper Resource Allocation**
   Containers are deployed without specifying resource limits, leading to unbounded memory usage and competition for system resources.
5. **Misconfigured Applications**
   Applications are configured to use more memory than what the container is allowed.
6. **Multiple Containers on the Same Node**
   If multiple containers are running on the same node, one container consuming excessive memory can lead to eviction of others.
7. **Node Resource Exhaustion**
   The node itself may not have enough memory to handle all the containers running on it.
8. **Unoptimized Code or Queries**
   Poorly optimized application code or inefficient database queries could lead to excessive memory usage.

How to Resolve OOMKilled

- **Analyze Logs**
  - Use kubectl logs <pod-name> -c <container-name> to check the container logs for any application-specific issues before the termination.
  - Investigate system logs or use kubectl describe pod <pod-name> to get more details about the OOMKilled event.
- **Increase Memory Limits**
  - Update the container's memory limits in the resources section of the Pod's YAML file:

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

- Apply the changes using kubectl apply -f <file.yaml>.
- **Optimize Application Code**
  - Fix memory leaks by identifying inefficient code or dependencies.
  - Perform load testing to ensure the application handles memory efficiently.
- **Use Vertical Pod Autoscaler (VPA)**
  - Deploy a **Vertical Pod Autoscaler** to dynamically adjust memory and CPU requests for the container.
- **Scale Out Workload**
  - Horizontal scaling can reduce memory usage per container by distributing the workload across multiple replicas:

kubectl scale deployment <deployment-name> --replicas=<number>

- **Monitor Resource Usage**
  - Use tools like **Prometheus**, **Grafana**, or **Kubernetes Dashboard** to monitor resource consumption.
  - Analyze metrics from kubectl top pod and kubectl top node.
- **Set Proper Resource Requests and Limits**
  - Ensure that requests and limits are set appropriately to prevent resource contention:

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


- **Use Quality of Service (QoS) Classes**
  - Assign QoS classes (Guaranteed, Burstable, BestEffort) by setting both requests and limits. Guaranteed pods are less likely to be OOMKilled.
- **Avoid Memory Overcommitment**
  - Ensure the node has sufficient memory to handle all scheduled containers without overcommitting resources.
- **Preemptive Scaling of Nodes**
  - Use **Cluster Autoscaler** to add nodes when resource limits on the current nodes are reached.
- **Enable Debugging Tools**
  - Tools like **kubectl-debug**, **kubectl exec**, and **kubectl cp** can help debug memory usage within a running container.
- **Investigate the Node**
  - Check the node's memory usage using kubectl describe node <node-name> to identify if the node itself is low on memory.

#4 Node Not Ready error

The Node Not Ready error in Kubernetes indicates a situation where a node within a Kubernetes cluster is not in a healthy state to accept pods for execution. This status is a crucial indicator for cluster administrators, as it signifies that the Kubernetes scheduler will not assign new pods to the affected node until it returns to a Ready state.

The **Node Not Ready** status in Kubernetes indicates that a node in your cluster is not functioning properly and cannot host any pods.

Nodes may enter a Not Ready state for a variety of reasons, ranging from network issues, resource exhaustion, misconfigurations, or underlying hardware problems. Understanding and resolving the root cause of this error is essential to maintain the operational efficiency and reliability of a Kubernetes cluster.

In Kubernetes, Nodes can be in one of several states, reflecting their current status and ability to accept workloads:

- A node in the **Ready** state is healthy and capable of accepting new pods.
- A node in the **Not Ready** state has encountered an issue that prevents it from functioning correctly.
- The **Unknown** state indicates that the Kubernetes master has lost communication with the node, and its status cannot be determined.

To determine if a node is experiencing a Node Not Ready error, and obtain the information necessary to solve the problem, follow these steps:

Troubleshooting the error

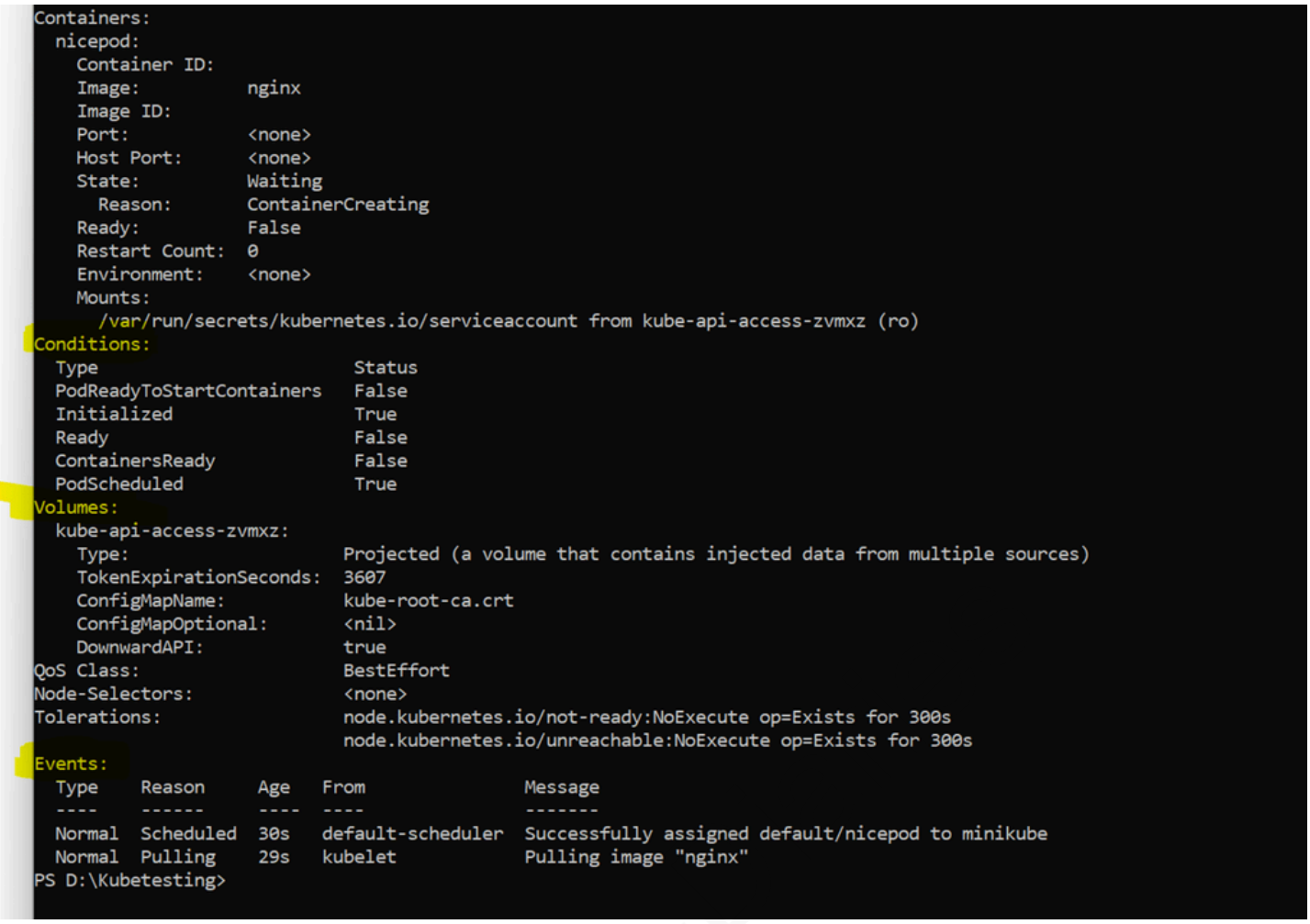
1. Checking Node State

The first step is to check the state of the nodes in the cluster. This can be done using the ***kubectl get nodes*** command, which lists all nodes and their statuses. A node marked as NotReady requires further investigation to understand the underlying issues.

2. Obtaining Node Details

kubectl describe node <node-name>

command provides comprehensive details about the node, including its conditions, events, and configuration. This information is useful for diagnosing the root cause of the Not Ready status, offering insights into any errors or warnings that the node might be experiencing. Analyzing this output helps pinpoint specific issues, guiding the troubleshooting and resolution processes.



Here are a few things to notice in the output, which could indicate the cause of the problem:

- **Conditions section:** This section lists various node health indicators.
- **Events section:** This section records significant events in the life of the node.

3. Checking System Logs

Logs from the kubelet, the primary component running on each node that communicates with the Kubernetes master, can provide insights into any errors or issues it is encountering.

You can access kubelet logs using journalctl or other logging utilities, depending on the node's operating system:

Possible Causes of the Node Not Ready Error

There are several conditions that can result in a node having a Not Ready status.

Possible Causes

1. **Node Issues:**
   - The kubelet service is not running.
   - Node has insufficient resources (CPU, memory, or disk).

- Node has networking or connectivity issues.
- Node is under maintenance or power-off.

2. **API Server Issues:**
   - The node cannot communicate with the Kubernetes API server.
3. **Component Issues:**
   - Missing or misconfigured critical components such as kubelet, container runtime (e.g., Docker, containerd), or kube-proxy.
   - Misconfigured cni (Container Network Interface) plugins.
4. **Configuration Errors:**
   - Outdated kubelet certificates or misconfigured kubelet on the node.
   - Issues with taints and tolerations.
5. **Cloud-Specific Problems:**
   - Cloud provider (e.g., AWS, GCP, Azure) misconfigurations.
   - Node not attached to the cluster due to IAM or role issues.

Also don't miss to check:

Scarcity of Resources

One common cause of the Node Not Ready error is the scarcity of resources, such as CPU or memory exhaustion. Monitoring resource usage can help identify if this is the cause. If the node is over-allocated, consider scaling down workloads or adding more nodes to the cluster.

kubelet Process

Restarting the kubelet might resolve some issues in the kubelet process. The command to restart the kubelet varies depending on the system manager in use.

sudo systemctl restart kubelet

This command restarts the kubelet service, potentially resolving issues that prevent the node from reaching a Ready state.

kube-proxy

Issues with kube-proxy, the network proxy running on each node, can also affect node readiness. Checking the status of kube-proxy and restarting it if necessary can help:

sudo systemctl status kube-proxy

This command checks the status of the kube-proxy service. If it's not running as expected, it can be restarted with:

sudo systemctl restart kube-proxy

Restarting kube-proxy can resolve network-related issues affecting the node's ability to communicate with the cluster, potentially resolving the Not Ready error.

Preventive Measures

We can have some preventive measures to take, so that we can safeguard our cluster to fall into the trap of this error.

- Monitor node health using tools like **Prometheus** or **Kubernetes Dashboard**.
- Set up resource limits and requests for your pods to prevent resource exhaustion.
- Regularly upgrade Kubernetes components and renew certificates.
- Automate disk cleaning or scaling using tools like Cluster Autoscaler.

#5 Gateway Timeout error

A Gateway Timeout error in a Kubernetes environment usually means that a service or application within the cluster is taking too long to respond, or it is unreachable. Here's a step-by-step guide to help you troubleshoot and resolve this issue, along with the possible causes.

Possible Reasons

1. **Application Issues**
   - The backend service is slow or unresponsive due to high CPU or memory utilization.
   - Long-running queries or insufficient resources are causing delays.
2. **Networking Issues**
   - Misconfigured network policies or ingress rules.
   - A misbehaving load balancer or DNS resolution issues.
3. **Service Configuration Issues**
   - Timeout settings on the Ingress Controller, Service, or Load Balancer are too short.
   - Service or pod selectors in Kubernetes are misconfigured, leading to no backend pods being targeted.
4. **Pod Issues**
   - Pods are in a CrashLoopBackOff, Pending, or Terminating state.
   - Horizontal Pod Autoscaler (HPA) is not scaling correctly to handle the traffic.
5. **Ingress or Load Balancer Issues**
   - Misconfigured ingress annotations.
   - Load balancer health checks are failing due to incorrect paths or ports.

Steps to Resolve

1. **Check Application Health**
2. Verify if the backend application is responsive.
3. Check logs of the application pods using:

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

Use kubectl exec to probe application readiness manually:

- *kubectl exec -it <pod-name> -n <namespace> -- curl http://localhost:<port>*

2. **Inspect Pod Status**

Check if pods are running and healthy:

- *kubectl get pods -n <namespace>*

For unhealthy pods, describe the pod to get more details:

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

3. **Examine Service Configuration**

Check if the Service is routing traffic to the correct pods:

- *kubectl get svc -n <namespace>*
- *kubectl describe svc <service-name> -n <namespace>*

Ensure the selector in the service matches the pod labels.

4. **Review Ingress/Load Balancer**

-Verify the ingress rules:

kubectl describe ingress <ingress-name> -n <namespace>

-Look for timeout settings in the ingress annotations (e.g., nginx.ingress.kubernetes.io/proxy-read-timeout).

-If using a Load Balancer, confirm the health check settings are correct.

5. **Check Resource Limits**

Ensure the application has sufficient resources:

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

Adjust resource requests and limits in the deployment if necessary.

6. **Increase Timeout Values**

Update the timeout settings in the ingress controller or load balancer annotations. For NGINX ingress:

annotations: nginx.ingress.kubernetes.io/proxy-connect-timeout: "30"
nginx.ingress.kubernetes.io/proxy-read-timeout: "30"

7. **Debug Networking**

Test connectivity from the ingress controller to the pods:

kubectl exec -it <ingress-pod> -n <namespace> -- curl http://<service-name>:<port> Check network policies:

kubectl get networkpolicy -n <namespace>

8. **Autoscaling**

If traffic is high, verify the HPA:

kubectl get hpa -n <namespace>

Scale the deployment manually if needed:

kubectl scale deployment <deployment-name> --replicas=<count> -n <namespace>

Best Practices to Avoid Gateway Timeout

1. Use readiness probes to avoid sending traffic to unready pods.
2. Configure appropriate resource requests and limits for your applications.
3. Implement autoscaling to handle sudden traffic spikes.
4. Monitor cluster health using tools like Prometheus and Grafana.
5. Optimize backend application response times.