Kubernetes Troubleshooting – The Complete Guide

What is Kubernetes Troubleshooting?

Kubernetes troubleshooting is the process of identifying, diagnosing, and resolving issues in Kubernetes clusters, nodes, pods, or containers.

More broadly defined, Kubernetes troubleshooting also includes effective ongoing management of faults and taking measures to prevent issues in Kubernetes components.

Kubernetes troubleshooting can be very complex. This article will focus on:

- **Providing solutions to common errors**, including `CreateContainerConfigError`, `ImagePullBackOff`, `CrashLoopBackOff` and `Kubernetes Node Not Ready`.
- **Explaining initial diagnosis** of problems in Kubernetes pods and clusters.
- **Showing where to find logs** and other information required for deeper analysis.

The Three Pillars of Kubernetes Troubleshooting

There are three aspects to effective troubleshooting in a Kubernetes cluster: understanding the problem, managing and remediating the problem, and preventing the problem from recurring.

Understanding

In a Kubernetes environment, it can be very difficult to understand what happened and determine the root cause of the problem. This typically involves:

- Reviewing recent changes to the affected cluster, pod, or node, to see what caused the failure.
- Analyzing YAML configurations, Github repositories, and logs for VMs or bare metal machines running the malfunctioning components.
- Looking at Kubernetes events and metrics such as disk pressure, memory pressure, and utilization. In a mature environment, you should have access to dashboards that show important metrics for clusters, nodes, pods, and containers over time.
- Comparing similar components behaving the same way, and analyzing dependencies between components, to see if they are related to the failure.

To achieve the above, teams typically use the following technologies:

- **Monitoring Tools:** Datadog, Dynatrace, Grafana, New Relic
- **Observability Tools:** Lightstep, Honeycomb
- **Live Debugging Tools:** OzCode, Rookout
- **Logging Tools:** Splunk, LogDNA, Logz.io

Management

In a microservices architecture, it is common for each component to be developed and managed by a separate team. Because production incidents often involve multiple components, collaboration is essential to remediate problems fast.

Once the issue is understood, there are three approaches to remediating it:

- **Ad hoc solutions**—based on tribal knowledge by the teams working on the affected components. Very often, the engineer who built the component will have unwritten knowledge on how to debug and resolve it.
- **Manual runbooks**—a clear, documented procedure showing how to resolve each type of incident. Having a runbook means that every member of the team can quickly resolve the issue.
- **Automated runbooks**—an automated process, which could be implemented as a script, infrastructure as code (IaC) template, or Kubernetes operator, and is triggered automatically when the issue is detected. It can be challenging to automate responses to all common incidents, but it can be highly beneficial, reducing downtime and eliminating human error.

To achieve the above, teams typically use the following technologies:

- **Incident Management:** PagerDuty, Kintaba
- **Project Management:** Jira, Monday, Trello
- **Infrastructure as Code:** Amazon CloudFormation, Terraform

Prevention

Successful teams make prevention their top priority. Over time, this will reduce the time invested in identifying and troubleshooting new issues. Preventing production issues in Kubernetes involves:

- Creating policies, rules, and playbooks after every incident to ensure effective remediation

- Investigating if a response to the issue can be automated, and how
- Defining how to identify the issue quickly next time around and make the relevant data available—for example by instrumenting the relevant components
- Ensuring the issue is escalated to the appropriate teams and those teams can communicate effectively to resolve it

To achieve the above, teams commonly use the following technologies:

- **Chaos Engineering:** Gremlin, Chaos Monkey, ChaosIQ
- **Auto Remediation:** Shoreline, OpsGenie

Why is Kubernetes Troubleshooting so Difficult?

Kubernetes is a complex system, and troubleshooting issues that occur somewhere in a Kubernetes cluster is just as complicated.

Even in a small, local Kubernetes cluster, it can be difficult to diagnose and resolve issues, because an issue can represent a problem in an individual container, in one or more pods, in a controller, a control plane component, or more than one of these.

In a large-scale production environment, these issues are exacerbated, due to the low level of visibility and a large number of moving parts. Teams must use multiple tools to gather the data required for troubleshooting and may have to use additional tools to diagnose issues they detect and resolve them.

To make matters worse, Kubernetes is often used to build microservices applications, in which each microservice is developed by a separate team. In other cases, there are DevOps and application development teams collaborating on the same Kubernetes cluster. This creates a lack of clarity about division of responsibility – if there is a problem with a pod, is that a DevOps problem, or something to be resolved by the relevant application team?

In short – Kubernetes troubleshooting can quickly become a mess, waste major resources and impact users and application functionality – unless teams closely coordinate and have the right tools available.

Troubleshooting Common Kubernetes Errors

If you are experiencing one of these common Kubernetes errors, here's a quick guide to identifying and resolving the problem:

CreateContainerConfigError

This error is usually the result of a missing Secret or ConfigMap. Secrets are Kubernetes objects used to store sensitive information like database credentials. ConfigMaps store data as key-value pairs, and are typically used to hold configuration information used by multiple pods.

How to identify the issue

Run `kubectl get pods`.

Check the output to see if the pod's status is `CreateContainerConfigError`

```
$ kubectl get pods
NAME                 READY   STATUS                       RESTARTS   AGE
pod-missing-config   0/1     CreateContainerConfigError   0
1m23s
```

Getting detailed information and resolving the issue

To get more information about the issue, run `kubectl describe [name]` and look for a message indicating which ConfigMap is missing:

```
$ kubectl describe pod pod-missing-config
Warning Failed 34s (x6 over 1m45s) kubelet
Error: configmap "configmap-3" not found
```

Now run this command to see if the ConfigMap exists in the cluster.

For example `$ kubectl get configmap configmap-3`

If the result is `null`, the ConfigMap is missing, and you need to create it. See the documentation to learn how to create a ConfigMap with the name requested by your pod.

Make sure the ConfigMap is available by running `get configmap [name]` again. If you want to view the content of the ConfigMap in YAML format, add the flag `-o yaml`.

Once you have verified the ConfigMap exists, run `kubectl get pods` again, and verify the pod is in status `Running`:

```
$ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
pod-missing-config   0/1     Running   0          1m23s
```

ImagePullBackOff or ErrImagePull

This status means that a pod could not run because it attempted to pull a container image from a registry, and failed. The pod refuses to start because it cannot create one or more containers defined in its manifest.

How to identify the issue

Run the `command kubectl get pods`

Check the output to see if the pod status is `ImagePullBackOff` or `ErrImagePull`:

```
$ kubectl get pods
NAME       READY    STATUS             RESTARTS   AGE
mypod-1    0/1      ImagePullBackOff   0          58s
```

Getting detailed information and resolving the issue

Run the `kubectl describe pod [name]` command for the problematic pod.

The output of this command will indicate the root cause of the issue. This can be one of the following:

- **Wrong image name or tag**—this typically happens because the image name or tag was typed incorrectly in the pod manifest. Verify the correct image name using `docker pull`, and correct it in the pod manifest.
- **Authentication issue in Container registry**—the pod could not authenticate with the registry to retrieve the image. This could happen because of an issue in the Secret holding credentials, or because the pod does not have an RBAC role that allows it to perform the operation. Ensure the pod and node have the appropriate permissions and Secrets, then try the operation manually using `docker pull`.

CrashLoopBackOff

This issue indicates a pod cannot be scheduled on a node. This could happen because the node does not have sufficient resources to run the pod, or because the pod did not succeed in mounting the requested volumes.

How to identify the issue

Run the command `kubectl get pods`.

Check the output to see if the pod status is `CrashLoopBackOff`

```
$ kubectl get pods
NAME       READY    STATUS             RESTARTS   AGE
mypod-1    0/1      CrashLoopBackOff   0          58s
```

Getting detailed information and resolving the issue

Run the `kubectl describe pod [name]` command for the problematic pod:

The output will help you identify the cause of the issue. Here are the common causes:

- **Insufficient resources**—if there are insufficient resources on the node, you can manually evict pods from the node or scale up your cluster to ensure more nodes are available for your pods.
- **Volume mounting**—if you see the issue is mounting a storage volume, check which volume the pod is trying to mount, ensure it is defined correctly in the pod manifest, and see that a storage volume with those definitions is available.
- **Use of hostPort**—if you are binding pods to a hostPort, you may only be able to schedule one pod per node. In most cases you can avoid using hostPort and use a Service object to enable communication with your pod.

Kubernetes Node Not Ready

When a worker node shuts down or crashes, all stateful pods that reside on it become unavailable, and the node status appears as `NotReady`.

If a node has a `NotReady` status for over five minutes (by default), Kubernetes changes the status of pods scheduled on it to `Unknown`, and attempts to schedule it on another node, with status `ContainerCreating`.

How to identify the issue

Run the command `kubectl get nodes`.

Check the output to see is the node status is `NotReady`

```
NAME        STATUS      AGE    VERSION
mynode-1    NotReady    1h     v1.2.0
```

To check if pods scheduled on your node are being moved to other nodes, run the command `get pods`.

Check the output to see if a pod appears twice on two different nodes, as follows:

```
NAME       READY    STATUS              RESTARTS      AGE    IP
NODE
mypod-1    1/1      Unknown             0             10m    [IP]
mynode-1
mypod-1    0/1      ContainerCreating   0             15s    [none]
mynode-2
```

Resolving the issue

If the failed node is able to recover or is rebooted by the user, the issue will resolve itself. Once the failed node recovers and joins the cluster, the following process takes place:

1. The pod with Unknown status is deleted, and volumes are detached from the failed node.
2. The pod is rescheduled on the new node, its status changes from `Unknown` to `ContainerCreating` and required volumes are attached.
3. Kubernetes uses a five-minute timeout (by default), after which the pod will run on the node, and its status changes from `ContainerCreating` to `Running`.

If you have no time to wait, or the node does not recover, you'll need to help Kubernetes reschedule the stateful pods on another, working node. There are two ways to achieve this:

- **Remove failed node from the cluster**—using the command `kubectl delete node [name]`
- **Delete stateful pods with status unknown**—using the command `kubectl delete pods [pod_name] --grace-period=0 --force -n [namespace]`

Learn more about Node Not Ready issues in Kubernetes.

Troubleshooting Kubernetes Pods: A Quick Guide

If you're experiencing an issue with a Kubernetes pod, and you couldn't find and quickly resolve the error in the section above, here is how to dig a bit deeper. The first step to diagnosing pod issues is running `kubectl describe pod [name]`.

Understanding the Output of the kubectl describe pod Command

Here is example output of the describe pod command, provided in the Kubernetes documentation:

```
Name:                nginx-deployment-1006230814-6winp
Namespace:        default
Node:                kubernetes-node-wul5/10.240.0.9
Start Time:         Thu, 24 Mar 2016 01:39:49 +0000
Labels:                app=nginx,pod-template-hash=1006230814
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","
apiVersion":"v1","reference":{"kind":"ReplicaSet","namespace":"
default","name":"nginx-deployment-1956810328","uid":"14e607e7-8ba1-11e7-
b5cb-fa16" ...
Status:                Running
IP:                10.244.0.6
Controllers:         ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID:
docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb114
9
    Image:                nginx
    Image ID:
docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e516370
7
    Port:                80/TCP
    QoS Tier:
      cpu:        Guaranteed
      memory:        Guaranteed
    Limits:
      cpu:        500m
      memory:        128Mi
    Requests:
```

```
      memory:                128Mi
      cpu:                500m
    State:                Running
      Started:                Thu, 24 Mar 2016 01:39:51 +0000
    Ready:                True
    Restart Count:        0
    Environment:        [none]
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-
5kdvl (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-4bcbi:
    Type:        Secret (a volume populated by a Secret)
    SecretName:        default-token-4bcbi
    Optional:   false
QoS Class:      Guaranteed
Node-Selectors: [none]
Tolerations:    [none]
Events:
  FirstSeen        LastSeen        Count
From
SubobjectPath                Type                Reason
Message
  ---------        --------        -----
----
-------------                --------        ------
-------
  54s                54s                1        {default-scheduler
}                                        Normal
Scheduled        Successfully assigned nginx-deployment-1006230814-
6winp to kubernetes-node-wul5
  54s                54s                1        {kubelet kubernetes-
node-wul5}        spec.containers{nginx}        Normal
Pulling                pulling image "nginx"
  53s                53s                1        {kubelet kubernetes-
node-wul5}        spec.containers{nginx}        Normal
Pulled                Successfully pulled image "nginx"
  53s                53s                1        {kubelet kubernetes-
node-wul5}        spec.containers{nginx}        Normal
Created                Created container with docker id 90315cc9f513
  53s                53s                1        {kubelet kubernetes-
node-wul5}        spec.containers{nginx}        Normal
Started                Started container with docker id 90315cc9f513
```

We bolded the most important sections in the `describe pod` output:

- `Name`—below this line are basic data about the pod, such as the node it is running on, its labels and current status.
- `Status`—this is the current state of the pod, which can be:
  - Pending
  - Running
  - Succeeded
  - Failed
  - Unknown
- `Containers`—below this line is data about containers running on the pod (only one in this example, called nginx),
- `Containers:State`—this indicates the status of the container, which can be:
  - Waiting
  - Running
  - Terminated
- `Volumes`—storage volumes, secrets or ConfigMaps mounted by containers in the pod.
- `Events`—recent events occurring on the pod, such as images pulled, containers created and containers started.

Continue debugging based on the pod state.

Pod Stays Pending

If a pod's status is Pending for a while, it could mean that it cannot be scheduled onto a node. Look at the describe pod output, in the Events section. Try to identify messages that indicate why the pod could not be scheduled. For example:

- **Insufficient resources in the cluster**—the cluster may have insufficient CPU or memory resources. This means you'll need to delete some pods, add resources on your nodes, or add more nodes.
- **Resource requirements**—the pod may be difficult to schedule due to specific resources requirements. See if you can release some of the requirements to make the pod eligible for scheduling on additional nodes.

Pod Stays Waiting

If a pod's status is Waiting, this means it is scheduled on a node, but unable to run. Look at the `describe pod` output, in the 'Events' section, and try to identify reasons the pod is not able to run.

Most often, this will be due to an error when fetching the image. If so, check for the following:

- **Image name**—ensure the image name in the pod manifest is correct
- **Image available**—ensure the image is really available in the repository
- **Test manually**—run a docker pull command on the local machine, ensuring you have the appropriate permissions, to see if you can retrieve the image

Pod Is Running but Misbehaving

If a pod is not running as expected, there can be two common causes: error in pod manifest, or mismatch between your local pod manifest and the manifest on the API server.

Checking for an error in your pod description

It is common to introduce errors into a pod description, for example by nesting sections incorrectly, or typing a command incorrectly.

Try deleting the pod and recreating it with `kubectl apply --validate -f mypod1.yaml`

This command will give you an error like this if you misspelled a command in the pod manifest, for example if you wrote `continers` instead of `containers`:

```
46757 schema.go:126] unknown field: continers
46757 schema.go:129] this may be a false alarm, see https://github.com
/kubernetes/kubernetes/issues/5786
pods/mypod1
```

Checking for a mismatch between local pod manifest and API Server

It can happen that the pod manifest, as recorded by the Kubernetes API Server, is not the same as your local manifest—hence the unexpected behavior.

Run this command to retrieve the pod manifest from the API server and save it as a local YAML file:

```
kubectl get pods/[pod-name] -o yaml > apiserver-[pod-name].yaml
```

You will now have a local file called `apiserver-[pod-name].yaml`, open it and compare with your local YAML. There are three possible cases:

- **Local YAML has the same lines as API Server YAML, and more**—this indicates a mismatch. Delete the pod and rerun it with the local pod manifest (assuming it is the correct one).
- **API Server YAML has the same lines as local YAML, and more**—this is normal, because the API Server can add more lines to the pod manifest over time. The problem lies elsewhere.
- **Both YAML files are identical**—again, this is normal, and means the problem lies elsewhere.

Diagnosing Other Pod Issues

If you weren't able to diagnose your pod issue using the methods above, there are several additional methods to perform deeper debugging of your pod:

- Examining Pod Logs
- Debugging with Container Exec
- Debugging with an Ephemeral Debug Container
- Running a Debug Pod on the Node

Examining Pod Logs

You can retrieve logs for a malfunctioning container using this command:

```
kubectl logs [pod-name] [container-name]
```

If the container has crashed, you can use the `--previous` flag to retrieve its crash log, like so:

```
kubectl logs --previous [pod-name] [container-name]
```

Debugging with Container Exec

Many container images contain debugging utilities—this is true for all images derived from Linux and Windows base images. This allows you to run commands in a shell within the malfunctioning container, as follows:

```
kubectl exec [pod-name] -c [container-name] -- [your-shell-commands]
```

Debugging with an Ephemeral Container

There are several cases in which you cannot use the `kubectl exec` command:

- The container has already crashed
- The container image is distroless, or purposely does not include a debugging utility

The solution, supported in Kubernetes v.1.18 and later, is to run an "ephemeral container". This is a container that runs alongside your production container and mirrors its activity, allowing you to run shell commands on it, as if you were running them on the real container, and even after it crashes.

Create an ephemeral container using `kubectl debug -it [pod-name] --image=[image-name] --target=[pod-name]`.

The `--target` flag is important because it lets the ephemeral container communicate with the process namespace of other containers running on the pod.

After running the debug command, kubectl will show a message with your ephemeral container name—take note of this name so you can work with the container:

```
Defaulting debug container name to debugger-8xzrl
```

You can now run `kubectl exec` on your new ephemeral container, and use it to debug your production container.

Running a Debug Pod on the Node

If none of these approaches work, you can create a special pod on the node, running in the host namespace with host privileges. This method is not recommended in production environments for security reasons.

Run a special debug pod on your node using `kubectl debug node/[node-name] -it --image=[image-name]`.

After running the debug command, kubectl will show a message with your new debugging pod—take note of this name so you can work with it:

```
Creating debugging pod node-debugger-mynode-pdx84 with container
debugger on node [node-name]
```

Note that the new pod runs a container in the host IPC, Network, and PID namespaces. The root filesystem is mounted at /host.

When finished with the debugging pod, delete it using `kubectl delete pod [debug-pod-name]`.

Troubleshooting Kubernetes Clusters: A Quick Guide

Viewing Basic Cluster Info

The first step to troubleshooting container issues is to get basic information on the Kubernetes worker nodes and Services running on the cluster.

To see a list of worker nodes and their status, run `kubectl get nodes --show-labels`. The output will be something like this:

```
NAME      STATUS    ROLES    AGE     VERSION        LABELS
worker0   Ready     [none]   1d      v1.13.0        ...,kubernetes.io
/hostname=worker0
worker1   Ready     [none]   1d      v1.13.0        ...,kubernetes.io
/hostname=worker1
worker2   Ready     [none]   1d      v1.13.0        ...,kubernetes.io
/hostname=worker2
```

To get information about Services running on the cluster, run:

`kubectl cluster-info`

The output will be something like this:

```
Kubernetes master is running at https://104.197.5.247
elasticsearch-logging is running at https://104.197.5.247/api/v1
/namespaces/kube-system/services/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/namespaces
```

```
/kube-system/services/kibana-logging/proxy
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-
system/services/kube-dns/proxy
```

Retrieving Cluster Logs

To diagnose deeper issues with nodes on your cluster, you will need access to logs on the nodes. The following table explains where to find the logs.

NODE TYPE	COMPONENT	WHERE TO FIND LOGS
Master	API Server	`/var/log/kube-apiserver.log`
Master	Scheduler	`/var/log/kube-scheduler.log`
Master	Controller Manager	`/var/log/kube-controller-manager.log`
Worker	Kubelet	`/var/log/kubelet.log`
Worker	Kube Proxy	`/var/log/kube-proxy.log`

Common Cluster Failure Scenarios and How to Resolve Them

Let's look at several common cluster failure scenarios, their impact, and how they can typically be resolved. This is not a complete guide to cluster troubleshooting, but can help you resolve the most common issues.

API Server VM Shuts Down or Crashes

- **Impact:** If the API server is down, you will not be able to start, stop, or update pods and services.
- **Resolution:** Restart the API server VM.
- **Prevention:** Set the API server VM to automatically restart, and set up high availability for the API server.

Control Plane Service Shuts Down or Crashes

- **Impact:** Services like the Replication Controller Manager, Scheduler, and so on are collocated with the API Server, so if any of them shut down or crashes, the impact is the same as shutdown of the API Server.
- **Resolution:** Same as API Server VM Shuts Down.
- **Prevention:** Same as API Server VM Shuts Down.

API Server Storage Lost

- **Impact:** API Server will fail to restart after shutting down.
- **Resolution:** Ensure storage is working again, manually recover the state of the API Server from backup, and restart it.
- **Prevention:** Ensure you have a readily available snapshot of the API Server. Use reliable storage, such as Amazon Elastic Block Storage (EBS), which survives shut down of the API Server VM, and prefer highly available storage.

Worker Node Shuts Down

- **Impact:** Pods on the node stop running, the Scheduler will attempt to run them on other available nodes. The cluster will now have less overall capacity to run pods.
- **Resolution:** Identify the issue on the node, bring it back up and register it with the cluster.
- **Prevention:** Use a replication control or a Service in front of pods, to ensure users are not impacted by node failures. Design applications to be fault tolerant.

Kubelet Malfunction

- **Impact:** If the kubelet crashes on a node, you will not be able to start new pods on that node. Existing pods may or may not be deleted, and the node will be marked unhealthy.
- **Resolution:** Same as Worker Node Shuts Down.
- **Prevention:** Same as Worker Node Shuts Down.

Unplanned Network Partitioning Disconnecting Some Nodes from the Master

- **Impact:** The master nodes think that nodes in the other network partition are down, and those nodes cannot communicate with the API Server.
- **Resolution:** Reconfigure the network to enable communication between all nodes and the API Server.
- **Prevention:** Use a networking solution that can automatically reconfigure cluster network parameters.

Human Error by Cluster Operator

- **Impact:** An accidental command by a human operator, or misconfigured Kubernetes components, can cause loss of pods, services, or control plane components. This can result in disruption of service to some or all nodes.
- **Resolution:** Most cluster operator errors can be resolved by restoring the API Server state from backup.
- **Prevention:** Implement a solution to automatically review and correct configuration errors in your Kubernetes clusters.