**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.

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

The Three Pillars of Kubernetes Troubleshooting:

**1. Understanding -**

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

**2. 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 quickly.

* 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 of 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, that 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

**3. 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

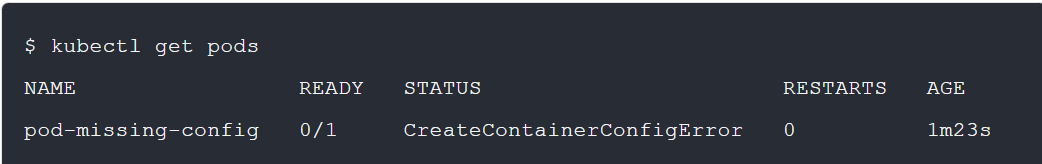
To achieve the above, teams commonly use the following technologies:

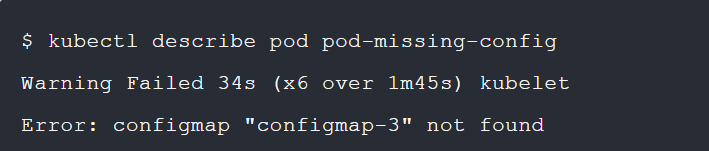
* Chaos Engineering: Gremlin, Chaos Monkey, ChaosIQ
* Auto Remediation: Shoreline, OpsGenie

Troubleshooting Common Kubernetes Errors

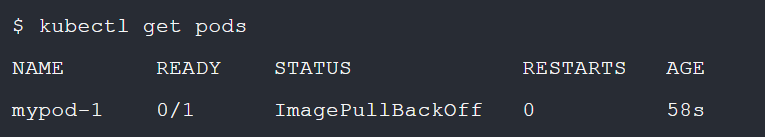
If you are experiencing one of these common Kubernetes errors, here’s a quick guide to identifying and resolving the problem:

1. 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.





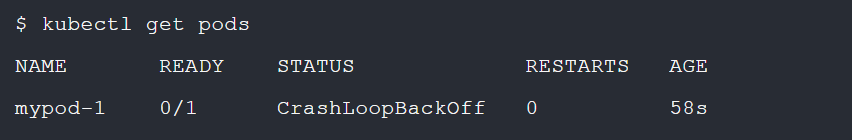
1. ImagePullBackOff or ErrImagePull - This status means that a pod could not run because it attempted to pull a container image from a registry, and failed. The pod refuses to start because it cannot create one or more containers defined in its manifest.



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.

1. 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.

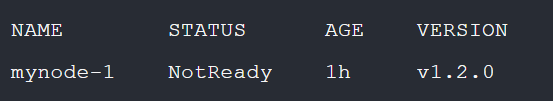


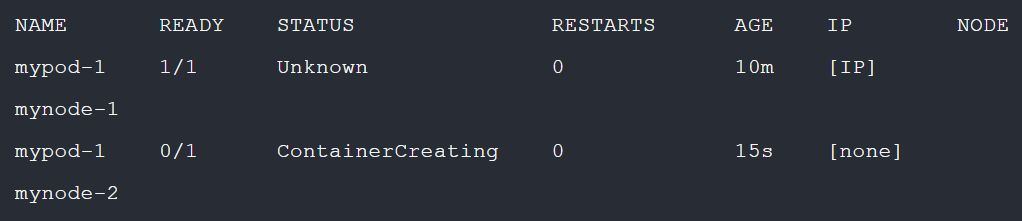
Insufficient resources—if there are insufficient resources on the node, you can manually evict pods from the node or scale up your cluster to ensure more nodes are available for your pods.

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.

1. 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 the status ContainerCreating.





**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:

The pod with Unknown status is deleted, and volumes are detached from the failed node. The pod is rescheduled on the new node, its status changes from Unknown to ContainerCreating, and the required volumes are attached.

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 the 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]

**Troubleshooting Kubernetes Pods:**

Command: kubectl describe pod [name]

**Highlighted the most important sections in the describe pod output:**

Name—below this line is basic data about the pod, such as the node it is running on, its labels, and 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.

**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 to your nodes, or add more nodes.
* Resource requirements—the pod may be difficult to schedule due to specific resource requirements. See if you can release some of the requirements to make the pod eligible for scheduling on additional nodes.

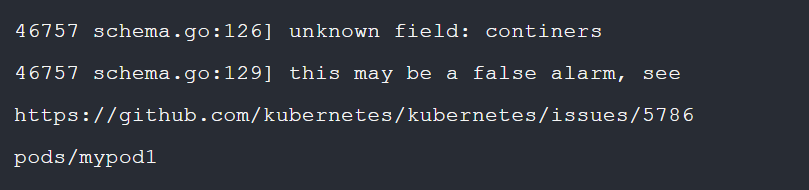
**Pod Stays Waiting** - If a pod’s status is Waiting, this means it is scheduled on a node, but unable to run. Look at the describe pod output, in the ‘Events’ section, and try to identify reasons the pod is not able to run. Most often, this will be due to an error when fetching the image. If so, check 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: an error in the pod manifest, or a 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



**Checking for a mismatch between the 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.



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



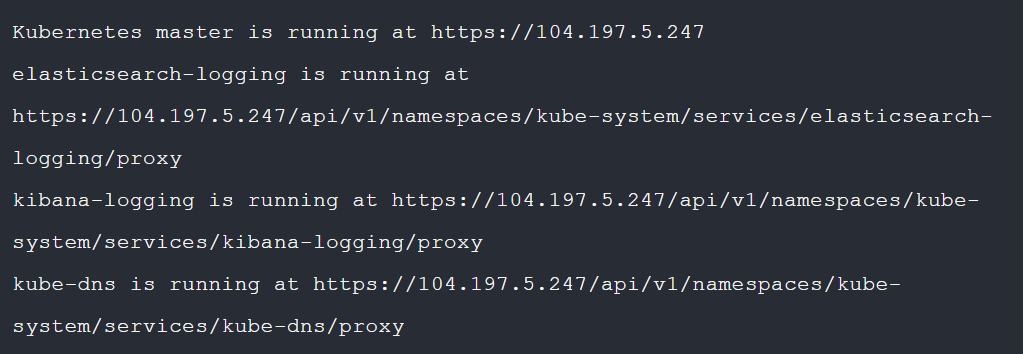
* Debugging with an Ephemeral Debug Container
  + Create an ephemeral container using kubectl debug -it [pod-name] --image=[image-name] --target=[pod-name].
* Running a Debug Pod on the Node
  + Run a special debug pod on your node using kubectl debug node/[node-name] -it --image=[image-name].

**Troubleshooting Kubernetes Clusters:**

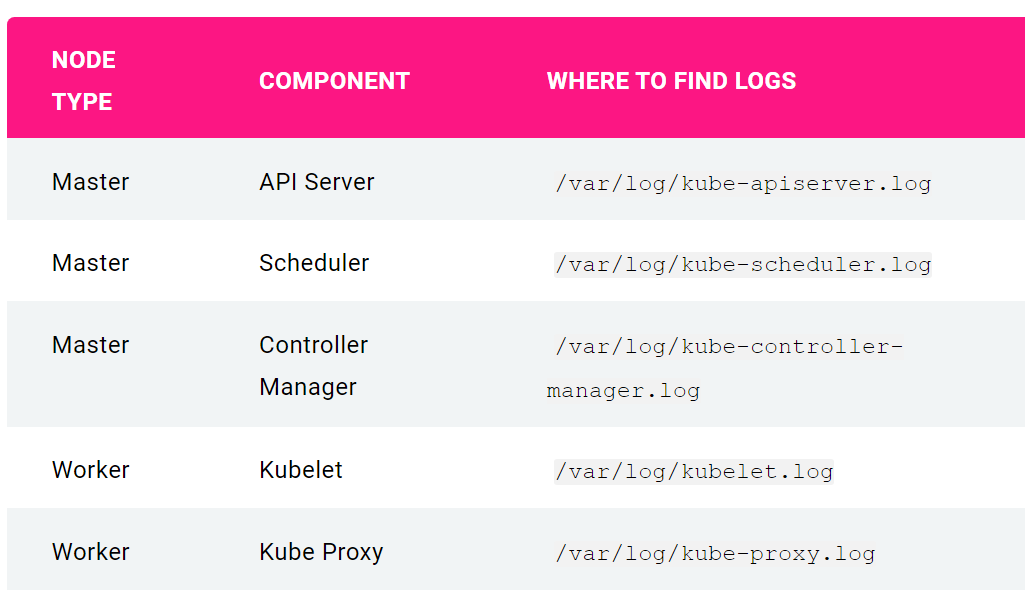
The first step to troubleshooting container issues is to get basic information on the Kubernetes worker nodes and Services running on the cluster. To see a list of worker nodes and their status.

**kubectl get nodes --show-labels**

**kubectl cluster-info**



**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.



**Common Cluster Failure Scenarios**

* 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 crash, the impact is the same as the 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 the backup.
  + Prevention: Implement a solution to automatically review and correct configuration errors in your Kubernetes clusters.