The Handbook for **Kubernetes Errors**

komodor

komodor.com

Table of Contents

Introduction

Ever wanted a magic wand to solve all of your Kubernetes troubles? Have you ever hoped for an ultimate guide that aggregates the most common Kubernetes issues, and provides the quick solutions for each? Does each Kubernetes error you encounter take you deeper and further down a scary and winding rabbit hole to troubleshoot?

If you replied yes to any or all of these questions... Well, look no further, we've got just the thing for you.

We have done the Kubernetes troubleshooting heavy lifting, so you don't have to waste hours scouring stack overflow questions, blog posts, and forums to get the answers you need, when you really need to solve your Kubernetes production issues - NOW.

Introducing the Swiss Army Knife of Kubernetes troubleshooting. This will round up the most common Kubernetes issues, and the quickest solutions to get to the root cause much more rapidly.

Don't leave home without it.

Kubernetes Error #1: CrashLoopBackOff

What is it?

A CrashLoopBackOff means that you have a pod starting, crashing, starting again, and then crashing again.

What Could be the Issue?

When it comes to CrashLoopBackOff, unfortunately, it's not a simple one-issue solution. There are actually a few main suspects that can lead to your Kubernetes deployment throwing this error. These include:

Option 1: Out of memory.

One of the most common reasons this happens is because the pod is out of memory.

Every pod has a specified memory space and when it tries to consume more memory than what has been allocated to it, the pod will keep crashing. This can occur if the pod is allocated less memory than it actually requires to run or if there is an error in the pod and it keeps on consuming all the memory space while in its run state.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: memory-demo
  labels:
    app: memory-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: memory-demo
  template:
    metadata:
      labels:
        app: memory-demo
    spec:
       containers:
       - name: memory-demo-ctr
         image: polinux/stress
         resources:
           limits:
             memory: "200Mi"
           requests:
             memory: "100Mi"
         command: ["stress"]
         args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

Option 2: Probe failure.

Another common reason is probe failure. The kubelet uses liveness, readiness, and startup probes to keep checks on the container. If the liveness or the startup probe fails, the container gets restarted at that point.

Option 3: hostPort usage.

This scenario is less common but if you do use hostPort this could be the reason for the CrashLoopBackOff error. When you bind a Pod to a hostPort, it limits the number of places the Pod can be scheduled, because each <hostIP, hostPort, protocol> combination must be unique.

Option 4: Application failure.

The application within the container itself keeps crashing because of some error and that can cause the pod to crash repeatedly.

Steps to Take to Resolve the Issue

Start by running:

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

Then you should probably start asking the following questions:

- Has one of the probe configurations been changed recently?
- Has this code ever been deployed?
- Was there any other code change just now?

These questions will help you find, debug and fix the error.

If this does not help you resolve the issue, using the same command:

Start by figuring out if you're using hostPort. If you are indeed using it, you should consider using NodePort instead, according to Kubernetes official documentation best practices guide. This just might be the quickest way to resolve the issue.

If neither of these solved the issue, you should start to dig at this point a bit deeper, and head into the logs to see if there is an application level error.

First start by extracting the application log itself. Seeing as the pod itself crashed, running the following command will bring up the previous pod's log:

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

Applications can fail at any time for any number of reasons, and since Kubernetes is not application-aware, it throws a generic error message that can often be tough to figure out. Your best bet with fixing this one quickly is by searching for application-level errors in your logs. Most likely the error message will be found in the last few lines of the log. Once you find the application error, you will be able to debug and fix it.

Kubernetes Error #2: ImagePullBackoff

What is it?

The ImagePullBackOff error occurs when a Pod startup fails due to an inability to pull the image. “Backoff”, in this use case, means that Kubernetes will continue to try and pull the image, with an increasing back-off delay.

What Could be the Issue?

Like CrashLoopBackOff, its less popular sibling ImagePullBackOff, can also map to several different underlying issues.

Option 1: Manifest not found.

This indicates that the specific version of the Docker repository is not available. Confirm that the image is available, and update accordingly if this is the case.

Option 2: Repository does not exist.

You cannot pull the image because you didn't specify credentials or your credentials are not correct.

Option 3: Authorization failed.

You cannot pull the image because you didn't specify credentials or your credentials are not correct.

Steps to Take to Resolve the Issue

Gather information to determine which one of the three is the root cause by running:

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

First start by taking a look at the ***Events*** section in the results. Scan the results for errors such as `manifest not found` or `repository not found` or `may require authorization` - these are some of the most common errors that cause the `ImagePullBackoff.` This can be resolved by something as simple as pulling the image locally, as there may be network issues between your K8 node, and the registry, or any other underlying issue such as a missing namespace or even an incorrect secret.

Kubernetes Error #3: ErrImagePull

What is it?

An ErrImagePull error occurs when a Pod startup fails due to an inability to pull the image, similar to the previous error, but for different reasons.

What Could be the Issue?

This error often maps to these three most common scenarios:

Option 1: Manifest not found.

This indicates that the specific version of the Docker repository is not available. Confirm that the image is available, and update accordingly if this is the case.

Option 2: Repository does not exist.

This indicates that the image repository you have specified does not exist on the Docker registry that your cluster is pointed at.

Option 3: Authorization failed.

You cannot pull the image because you didn't specify credentials or your credentials are not correct.

Steps to Take to Resolve the Issue

So you may feel like you're seeing double, but sometimes similar underlying issues or mistakes can throw different errors in a complex, infinitely scalable system like Kubernetes.

You can follow the same steps you followed in ImagePullBackoff, and this will likely resolve the issue.

Kubernetes Error #4: CreateContainerConfigError

What is it?

This error usually occurs when Kubernetes tries to create a container in a pod, but fails before the container enters the running state.

What Could be the Issue?

Option 1: Missing ConfigMap

ConfigMaps are Kubernetes objects that let you store the configuration for other objects to use. A ConfigMap is a dictionary of key-value pairs of strings. ConfigMaps are helpful for storing and sharing non-sensitive, unencrypted configuration data. These allow you to decouple broad environment-specific configurations from your container images, making your application easily portable. If Kubernetes fails to create or find the ConfigMap, the deployment will fail.

Option 2: Missing Secret

Kubernetes Secrets are secure objects which store sensitive data, such as passwords, OAuth tokens, and SSH keys, etc. Using a secret means that you don't need to include confidential data in your application code. Secrets are similar to ConfigMaps but are specifically intended to hold confidential data. If Kubernetes fails to create or find the Secret, the deployment will fail.

Steps to Take to Resolve the Issue

Run

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

Check to see if you get an error such as

`secret ... not found` or `configmap ... not found.`

Then, run

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

and examine if the relevant resource is there. If not, create it. This could be also an issue with the content of the configmap/secret meaning that the resource exists in the correct namespace but there's an issue with its content such as a missing key in the configmap. In this case, you should update your configmap/secret and restart the pods. That simple.

Kubernetes Error #5: CreateContainerError

What is it?

This issue happens when Kubernetes tries to create a container but fails, which can happen for a couple of reasons.

What Could be the Issue?

Option 1: Issue with the container runtime

In the event that a container runtime error is suspected, the first order of business should be to try cleaning up old containers, changing the container name, or checking kubelet logs on the node the container is running on.

Option 2: Issue with starting up the container

This can also be an indication that the container cannot run at all. This could be related to the start up command, it might not be available to the image, not specified or not executable. In each case, the container would not be able to run.

Steps to Take to Resolve the Issue

With this error you will also need to run:

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

And then look in the ***Events*** section. Based on the errors that you see here (and there can be quite a few), you will know what you need to debug and fix. This can be anything from a startup command or a command within the pod, but this would be the first place to start looking to get to the root cause. Some examples include, "`container name [...] already in use by container`" or "`starting container process caused`" - if these errors appear in the Events section, there is a high likelihood that there is an issue with the container runtime, or container initialization, respectively.

Kubernetes Error #6: FailedScheduling

What is it?

This issue happens when some pods fail to get scheduled.

What Could be the Issue?

Option 1: Allocatable resources on the node are lower than the pod's requested resources

The Kubernetes scheduler will attempt to find a node that can allocate resources to satisfy the pod's requests. If it is unsuccessful, the pod will remain Pending until more resources become available. This could mean:

1. You have requested more CPU/memory available to any of the nodes. In this case, you can decrease the pod's CPU/memory request, free up CPU/memory on the nodes or provision larger nodes with more resources.
2. There is no more capacity in the cluster to allocate the CPU/memory you requested. In this case, you should add more nodes to the cluster.

Option 2: Unschedulable nodes

Kubernetes will mark a node as unschedulable if you cordon it. This is useful as a preparatory step before a node reboot or other maintenance. In this case, you should either uncordon the node or provision more nodes.

Option 3: Nodes taints

Kubernetes lets you add taints to nodes and tolerations to pods to influence scheduling decisions. Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. If you've added taints to a node, the node should only accept pods that tolerate these taints. If the scheduler could not find any suitable node, you will get a FailedScheduling message. How can you fix it? Ensure that the pod's tolerations match the node's taints.

Option 4: Nodes labels

There are circumstances where you may want to control which node the pod deploys to. One of the ways to do this is to use nodeSelector. nodeSelector is the simplest recommended form of node selection constraint. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels.

If the scheduler is unsuccessful in finding a node with a label matching the pod's nodeSelector, you will get a FailedScheduling message.

To solve this problem, you'll need to ensure that at least one available node includes a label that matches the pod's node selector.

Steps to Take to Resolve the Issue

Ok, we promise, we're not trolling you. This error most commonly happens when either there is not enough CPU, memory, or a combination of the two.

Understanding the CPU and Memory thresholds allocated can also be discovered by running:

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

Once you discover which is the culprit, you can ensure that sufficient memory and CPU is allocated for the deployment.

An important note:

When it comes to K8s errors and issues, memory issues are critical, and can be the cause for the application to be killed. In comparison, CPU issues are not considered production critical. While they can cause the application to slow down, it will likely not be a cause for the application or pod to be killed.

'Making the distinction between compressible and incompressible resources is important. If your containers consume too many compressible resources such as CPU, they are throttled, but if they use too many incompressible resources (such as memory), they are killed (as there is no other way to ask an application to release allocated memory).' (Kubernetes Patterns O'Reilly Book, Bilgin Ibryam, Roland Huß, 2019)

Kubernetes Error #7: NonZeroExitCode

What is it?

This issue happens when the container exists with a non healthy exit code. Buh-bye.

What Could be the Issue?

So just as the error implies different exit codes will provide you with greater clarity regarding the specific underlying issue.

Every exit codes maps to a different deployment error, and there are multiple exit codes to consider. Some examples include:

Exit Code 1:

Indicates that the container stopped due to either (a) an application error or (b) an invalid reference, meaning, the image specification is referring to a file which does not exist in the container image.

Exit Code 127:

Indicates that the command specified in the container specification refers to a non-existent file or directory.

Steps to Take to Resolve the Issue

Run our favorite kubectl command:

```
kubectl get pod <pod name> -n <namespace> -o json | grep exitCode
```

Once you know the exit code the container is throwing, you'll be able to quickly find and fix the underlying issue. If you're looking for a detailed piece on all exit codes, and how to resolve them, then make sure to check out this complete guide.

Kubernetes Error #8: OOMKilled

What is it?

When a pod has a memory limit defined and if the pod memory usage crosses the specified limit, the pod will get killed, and the status will be reported as OOMKilled.

If the node experiences an out of memory (OOM) event prior to the kubelet being able to reclaim memory, the node depends on the oom_killer to respond.

What Could be the Issue?

Remember our FailedScheduling error? Well, OOMKilled is often what happens when you have exceeded your memory limits and your pod cannot continue functioning safely.

Steps to Take to Resolve the Issue

The first thing you'd need to check is whether the memory limits and thresholds were recently changed, or even worse - if they were never configured in the first place. If you have not configured your memory limits, start there, and keep in mind the best practices listed below when allocating memory.

Once you configure your memory properly, you can safely restart your pod - and if it is stateless (read more on this here), it should be a graceful restart and should function properly. If it's not, well that's a whole nother rabbit hole to go down.

A Useful Deep Dive: How Kubernetes Resource Allocation Works Under the Hood

Following the FailedScheduling error, you might be wondering what are some good practices for defining CPU and memory allocation so that your pods don't encounter such issues. Therefore, we've decided to add a small bonus section to do a quick overview of how resource allocation works in Kubernetes.

Like with all things in engineering, there is no universal standard for CPU and memory allocation - this is essentially dictated by the application or service's specific requirements. It is usually determined and ultimately defined by performing stress testing on the application to understand the minimum requirements for it to run properly.

The best way to do this for Kubernetes is by leveraging its resource model, and defining the required resources by application or container - this is determined by the Quality of Service (QoS) class.

The QoS class is a Kubernetes concept which determines the scheduling and eviction priority of pods.

QoS is assigned to pods by Kubernetes but it is something we can influence in the way we configure our pods, by understanding pod priorities. When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

1. Guaranteed
2. Burstable
3. Best Effort

Now, let's take a look at resource allocations in K8s. Resources are comprised of ***limits*** and ***requests*** whereby both of these are defined in CPU and memory thresholds - meaning, what is the maximum number of requests available to this pod (in CPU and memory), and what are the total limits of CPU and memory this pod can consume.

The requests amount is used by the scheduler when placing Pods to nodes. For a given Pod, the scheduler considers only nodes that still have enough capacity to accommodate the Pod and all of its containers by summing up the requested resource amounts. In that sense, the requests field of each container affects where a Pod can be scheduled or not.

Best effort quality of service is the lowest level of priority.

'[A] pod that does not have any requests and limits set for its containers. Such a Pod is considered as the lowest priority and is killed first when the node where the Pod is placed runs out of incompressible resources.' (Bilgin Ibryam, Roland Huß, 2019).

This will happen if there are no resources at all defined for the pod.

Burstable QoS is determined when there is some discrepancy between the resource allocation, and this is second priority for the K8s scheduler, meaning that burstable pods will only be killed if there are no best effort pods left to kill.

'[A] pod that has requests and limits defined, but they are not equal (and limits is larger than requests as expected). Such a Pod has minimal resource guarantees, but is also willing to consume more resources up to its limit when available. When the node is under incompressible resource pressure, these Pods are likely to be killed if no Best-Effort Pods remain.' (Bilgin Ibryam, Roland Huß, 2019).

Guaranteed quality of services means that the resources - requests and limits - are balanced. This guarantees that the pod will stay alive, and won't be killed by the K8s scheduler before burstable or best effort pods.

'[A] pod that has an equal amount of request and limit resources. These are the highest-priority Pods and guaranteed not to be killed before Best-Effort and Burstable Pods.' (Bilgin Ibryam, Roland Huß, 2019).

A Last Note on Nodes

Nodes also have allocatable memory and CPU (meaning, they also have resources that can be defined). When we create machines, we do so with the demands and load required by the application for CPU and Memory in mind. What happens sometimes is that the sum of the resources and limits defined for the pods is greater than the allocatable resources of the node itself. The node doesn't have sufficient resources to run the pods defined, as they're requesting more. When this happens, it can lead to errors such as the OOMKilled error which we discussed earlier.

We hope this quick runthrough helped you gain a better understanding of the inner workings of Kubernetes resource allocation, and best practices for getting this right in order to ensure your high priority pods and nodes receive the highest quality of service.

Conclusion

The Komodor team is continuously hard at work to make Kubernetes deployments and management in the long-term much more frictionless for everyone in the chain - from the most seasoned DevOps engineers to junior developers.

We rolled up our sleeves and compiled this guide of the most common Kubernetes issues to enable DevOps engineers to empower developers to be more autonomous when troubleshooting K8s-based deployments - enabling greater velocity for all teams.

We hope you found this helpful - let us know if there are any other errors you'd like to see in this report and we'll be happy to add them to the guide to keep it fresh and useful.