Kubernetes Troubleshooting

A hands-on practitioner's guide to solving some of Kubernetes' toughest operational issues

OpsCruise

TABLE OF CONTENTS

1 Kubernetes, Complexity, and Monitoring

It's a well known fact that Kubernetes is being adopted by organizations of all sizes across all verticals and is quickly becoming the new standard for application development. This is due to a few key reasons:

- The ability to remove the complexity of managing infrastructure by developers
- Simplicity of defining nearly "everything"-as-code
- Easy isolation of application environments with Namespaces
- Application deployments can be as simple or as complex as needed, with fine grained scheduling controls
- A broad API allowing tight control of deployments along with high extensibility
- Tight integration with cloud providers for uncomplicated deployments, simple network ingress definitions, native cloud storage access, among others
- Automated workload lifecycle management by the orchestrator makes having highly available, self-scaling, and self-recovering applications easier than ever

With this explosive growth of Kubernetes adoption, many of the previous generations of application's problems have decreased, but, as with any tool, new challenges arise.

Kubernetes applications are often ephemeral, meaning they come and go often, sometimes in a matter of minutes. With everything defined as code, there are times when a specific resource or configuration is missed. The tight integration with cloud providers means that, in some scenarios, problems can lie just outside of the Kubernetes environment. Finally, with the modular building blocks that form not just Kubernetes applications, but all modern applications, complexity has skyrocketed. This means that finding the fault domain and failing entity in modern environments is closer to finding needles in cloud-and-datacenter haystacks.

With this explosive growth of Kubernetes adoption, many of the previous generations of application's problems have decreased, but, as with any tool, new challenges arise.

From a Few Fat Processes to Many Thin Ones

Microservices exhibit different performance profiles from traditional monolithic applications in a variety of ways. Consider, for example, the impact of "the network within" – an important aspect of microservice applications.

- *Large Number of Thin Components* – Instead of a smaller number of thick components, the application now comprises a large number of thin components. They are ephemeral in many cases, can dynamically change in number of running instances, and can exist within multiple servers across data centers and clouds. On top of this, multiple versions can co-exist for A/B testing, canary deployments, and the like.
- *East-West Communication* – These components communicate frequently with each other by sending data in serial fashion and not by switching memory around. This East-West communication can cross server and data center boundaries, as well as Internet boundaries, to reach other microservices appearing as North-South traffic.
- *Communication to Computation Ratio* – The communication to computation ratio in the application can shift dramatically as the application spends significantly more time in networking, increasing to as much as 41% from 18% before [1]. Unlike monolithic applications, bottlenecks in microservices components in the chain can shift from one service to another as load conditions change.

[1] Yu Gan and Christina Delimitrou, "The Architectural Implications of Cloud Microservices, "IEEE COMPUTER ARCHITECTURE LETTERS, VOL. 17, NO. 2, JULY-DECEMBER 2018

The above changes raise multiple questions around application performance. You may ask:

- As a number of microservices communicate with each other and rely on a hierarchy of virtual machines, where is the performance bottleneck?
- How do I know if the problem is due to a change in the application or a change in the infrastructure?
- As workload functionality changes, how does the application behave?
- How do code updates impact the microservices?

These are the questions that keep performance and production Ops awake at night.

Like Wine, Kubernetes Errors Have Unique Noses

Kubernetes problems come in many different variants. When troubleshooting, it's important to understand the type of problem scenario you're dealing with.

Typically, problems fall into one (or more) of these common scenarios;

- Configuration of prerequisites
- Access to required resources
- Resource Allocation
- Performance Issues
- Functional failures

Crashes, Loops, and BackOffs

Many of these problem scenarios mentioned above are surfaced as a CrashLoopBackOff, which is what we'll focus on for most of this document. What is a Crash, a Loop, and a Backoff ?

- A Crash is the voluntary or involuntary exit of a container. This can be caused by a variety of reasons, such as a process exiting or Out of Memory (OOM) kills. Additional detail will be covered later. Understanding a crash's cause allows us to fix the problem.
- A Loop is implicit in the way Kubernetes is designed. You specify intent, running a Pod, for example. Kubernetes tries to ensure a copy of the Pod is always running. In case it crashes, the Kubelet in the Node detects this and the scheduler tries to bring it up again, over and over. This is the Loop.
- A Backoff is a way to ensure that excessive resources aren't consumed when going through a loop that is failing continuously. After retrying a few times, there's a wait period before trying again. This is the Backoff. This wait period keeps increasing and finally resets when the Pod returns to a healthy state.

Phases of a Container

Crashes take place at different points in the life of a container. These lifecycle stages where a Pod can fail (excluding a Pod's end-of-life itself) can generally be grouped into three phases: Preparation, Initialization, and Run.

- **Preparation Phase**
  In this phase, Kubernetes ensures all configured prerequisites are locked in and ready for the Pod to run. This is primarily Kubernetes' responsibility.
- **Initialization Phase**
  Here, the container receives the configuration and uses it to set up access to the various endpoints it interacts with, such as databases, read and set runtime parameters, and generally set up configuration. This is the application's responsibility.
- **Run Phase**
  When the Initialization phase is over, the application is ready to perform its tasks, such as sending and receiving requests, running a job, etc. This is the phase where a workload will spend most its life.

Next, let's look at common problems that occur in each of these phases of a Pod.

Errors Across Phases

In this section, we'll look at errors that commonly show up in each phase, as well as how to identify them and resolve them. Each phase can have many different classes of problems surfacing. Knowing how to identify them is key to quickly resolve any problems and outages Kubernetes applications are experiencing.

Preparation Phase Errors

Preparation phase errors reflect a Pod's inability to start. There may be issues in creating the container, pulling an image, finding configuration items, or even scheduling workloads. Some examples of errors during this phase include:

- CreateContainerConfigError
- ImagePullBackOff or ErrImagePull
- Node Not available
- PV not available
- Security
- Insufficient Compute Resources

Let's take a look at some of these error types in more detail.

Insufficient Compute Resources

Kubernetes allows you to specify how much compute resource your container needs. This means you can specify the minimum required amount of CPU or Memory (among other resources) that your container requires. Kubernetes will make intelligent decisions on where to schedule the Pods and containers based on these resource requirements. For example, a Pod requiring at least 2GB of memory to run will not be scheduled on a Worker Node that only has 1.5GB available. The orchestrator will find another suitable Node to place the Pod on.

This brings the possibility that a Pod's minimum resource request is not available, meaning that the containers cannot be scheduled. The Pod will remain in a pending state until the resource requirements are relaxed or the available resources on a Node change to allow enough to be assigned for the Pod.

How it Manifests

We've created a sample Pod with a manifest named `high-mem-request-deploy.yaml` that specifies a deployment named `nginx-deployment` with 3 replicas and that requires 64GB of memory.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: demo
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "64Gi"
```

We'll run `kubectl apply -f high-mem-request-deploy.yaml` to have Kubernetes create the deployment and Pods. Then we'll run `kubectl get pods -n demo` to get the Pods in the `demo` namespace. We see the three replicas attempting to start, but stuck in a `Pending` state.

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-bb5cd76d-5xkj6     0/1     Pending   0          16m
nginx-deployment-bb5cd76d-prdkj     0/1     Pending   0          16m
nginx-deployment-bb5cd76d-vtptl     0/1     Pending   0          16m
```

We can get additional detail by copying the name of one of the pods into our clipboard then running `kubectl describe pod nginx-deployment-bb5cd76d-5xkj6 -n demo`. Under the Events section, we can see that there are no Nodes available for scheduling.

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  20m   default-scheduler  0/5 nodes are available: 1
node(s) had taint that the pod didn't tolerate, 4 Insufficient memory.
```

One of the Nodes has a taint (it's the Master Node, and workloads usually don't get scheduled on these, unless you provide a Toleration and the other 4 have insufficient memory to allocate to the Pod.

How to Fix It

To fix the problem, we can do one of three things: **1)** we can increase our Node's memory size or create a Node with over 64GB of memory and add it to the cluster, or **2)** we can reduce the request size for memory in the Pod manifest to a value that our Node can schedule, or **3)** we can rebalance the cluster.

If your workload absolutely requires that high of a resource allocation, reducing the request size is not an option. You will have to either add a larger Node to your cluster or, if you already have a large enough Node but its resources are being used by other workloads, you can re-balance the node by using Node Selectors, Taints and Tolerations, and other scheduling techniques.

For our example, we are okay with reducing the required memory. We'll change the request to 2GB as shown below.

```
      …
resources:
  requests:
    memory: "2Gi"
```

Then, we'll run `kubectl apply -f high-mem-request-deploy.yaml` again to update the deployment. Executing kubectl get pods -n demo now shows us the containers successfully created.

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-54fb9cb4bc-rftnv   1/1     Running   0          24s
nginx-deployment-54fb9cb4bc-q2l8z   1/1     Running   0          19s
nginx-deployment-54fb9cb4bc-2lkh6   1/1     Running   0          18s
```

CreateContainerConfigError

Kubernetes Pods have native access to 2 commonly used configuration resources: ConfigMaps and Secrets. ConfigMaps are Kubernetes resources used to store non-confidential data. Pods can then use the data in ConfigMaps as environment variables, application arguments, or as configuration files mounted to a storage volume. Secrets provide similar capabilities, with the core difference being that they're meant for data that should be confidential.

These resources decouple your container images from your configuration, making your applications more portable.

When these resources are specified in a manifest, Kubernetes makes sure that they exist when starting a Pod. A missing ConfigMap or Secret will cause a Pod startup failure, with a CreateContainerConfigError listed as the cause of the failure.

How it Manifests

We've created a sample Pod with a manifest named `missing_cm.yaml` that references a ConfigMap named `key-cm` to set a Pod's environment variables.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: demo
  name: missing-cm
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/echo", "$(MY_KEY1) $(MY_KEY2)" ]
      env:
        - name: MY_KEY1
          valueFrom:
            configMapKeyRef:
              name: key-cm
              key: MY_KEY1
        - name: MY_KEY2
          valueFrom:
            configMapKeyRef:
              name: key-cm
              key: MY_KEY2
```

We'll run `kubectl apply -f missing_cm.yaml` to have Kubernetes create the Pod. Then we'll run `kubectl get pods -n demo` to get the Pods in the demo namespace. We see the results below.

```
NAME          READY           STATUS                       RESTARTS       AGE
missing-cm    0/1             CreateContainerConfigError   0              15s
```

We can get additional detail by running `kubectl describe pod missing-cm -n demo`. Scrolling to the bottom of the output will show us the cause of the error. Under the Events section, we can see that there's an error finding the configmap.

```
Events:
  Type      Reason    Age    From       Message
  ----      ------    ----   ----       -------
  Warning   Failed    89s    kubelet    Error: configmap "key-cm" not found
```

How to Fix It

To fix the problem, we simply need to create the ConfigMap for the Pod to utilize. Running `kubectl get configmap -n demo` shows us that the ConfigMap doesn't exist in the namespace.

```
NAME   DATA   AGE
kube-root-ca.crt    1            265d
```

We'll create a simple ConfigMap in a file called `key-cm.yaml` containing the expected keys.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: key-cm
  namespace: demo
data:
  MY_KEY1: "First Key"
  MY_KEY2: "Second Key"
```

We'll run `kubectl apply -f key-cm.yaml` next. Once it has been created, the Pod will start up properly. Then, we'll run `kubectl get configmap -n demo` again to verify it's been created. The output now shows the `key-cm` ConfigMap.

```
NAME                  DATA                  AGE
key-cm                2                     9s
kube-root-ca.crt      1                     265d
```

If we run `kubectl get pods -n demo` now, we'll see the Pod is now in a running state.

```
NAME           READY   STATUS              RESTARTS       AGE
missing-cm     0/1     ContainerCreating   0              0s
missing-cm     1/1     Running             0              4s
```

Initialization Phase Errors

The Initialization phase is responsible for setting up the configuration of the application. At this point, the Infrastructure and Kubernetes layers are up and running successfully. Compute resources have successfully been assigned, the Pod has been scheduled, the container image has been pulled successfully, and now the application itself is running.

Each application is unique in its startup dependencies, but most have basic requirements and checks that they run to validate that they're running an adequate configuration. This could be something like checking that a dependent library exists, a connection to a database, certain startup parameters, or a myriad of others. The one thing they have in common is that all applications require those conditions to start up correctly. Missing or incorrect configurations will usually cause an application to not start up properly.

Examples of these type of error scenarios include:

- Unsuccessful database connection
- Missing or Invalid Configuration
- Inability to access required external services (authentication service, 3rd party API, etc.)
- Existence of required code libraries on the file system

We'll explore in more detail below.

Missing or Invalid Configuration

Applications each have their own set of required files for starting up correctly. What each one needs is unique, but they all have similar requirements, whether it's a configuration directory or file, certain set of configuration entries in said configuration files, the existence of a shared library accessible to the application, proper setting of environment variables, and other similar entries.

A bad configuration or missing configuration will usually cause an application to fail to start properly, meaning a crash before a full startup or faulty behavior at runtime leading to a crash. In Kubernetes, this type of crash will cause the Pod to be recreated, and it will then go into the Loop described above until the root cause of configuration is fixed.

How it Manifests

We've deployed Bitnami's Apache Web Server deployment using Helm. While the Apache Web Server configuration is outside the scope of this document, the scenario we've created will highlight this type of scenario and applies to many other configuration problems, regardless of application.

Apache Web Server uses a configuration file called `httpd.conf` for its configuration. We created a ConfigMap in our demo namespace with the contents of this file by first creating it in our filesystem, then running `kubectl create configmap httpd-cm --from-file=httpd.conf -n demo`. We then used Helm to install Apache and use the ConfigMap by mounting it as the `httpd.conf` file by running the following:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

(While we do not provide details on how to use Helm, know that it is a package manager like `apt` or `yum` except it's for Kubernetes. It helps quickly deploy applications while allowing for high configurability. Learn more at https://helm.sh/).

By running `kubectl get pods -n demo --watch`, we can see the container getting created, running, then going into an error state and crashing, followed by going into the crash loop. At this point, the container did run, but the crash happened at the application level.

```
NAME                      READY       STATUS              RESTARTS     AGE
apache-88494c65b-b84mg    0/1         Running             1            6s
apache-88494c65b-b84mg    0/1         Error               1            6s
apache-88494c65b-b84mg    0/1         CrashLoopBackOff    1            7s
apache-88494c65b-b84mg    0/1         Running             2            21s
apache-88494c65b-b84mg    0/1         Error               2            22s
apache-88494c65b-b84mg    0/1         CrashLoopBackOff    2            23s
apache-88494c65b-b84mg    0/1         Running             3            54s
apache-88494c65b-b84mg    0/1         Error               3            55s
apache-88494c65b-b84mg    0/1         CrashLoopBackOff    3            56s
```

While running '`kubectl describe pod apache-88494c65b-w7dfw -n demo`' worked previously for startup related issues, because this pod has successfully started and is now running code, the only failure indicator is that of a backoff restart.

```
Type     Reason  Age                 From    Message
----     ------  ----                ----    -------
Warning  BackOff 38s (x422 over 85m) kubelet Back-off restarting failed container
```

To find a cause, we need to see container logs. Running `kubectl logs apache-88494c65b-p7lfd -f` shows us the container's logs.

```
apache 17:48:28.58 Welcome to the Bitnami apache container
apache 17:48:28.60
apache 17:48:28.61 INFO  ==> ** Starting Apache setup **
apache 17:48:28.71 INFO  ==> ** Apache setup finished! **
apache 17:48:28.75 INFO  ==> ** Starting Apache **
httpd: Syntax error on line 178 of /opt/bitnami/apache/conf/httpd.conf: Could not
open configuration file /opt/bitnami/apache/conf/Suite-extra/components.conf: No
such file or directory
```

The last line of the logs gives us a clear indication as to what is causing the container to crash. There is a missing file at `/opt/bitnami/apache/conf/Suite-extra/components.conf` configured at line 178 of the `httpd.conf` we created. Our application's configuration is faulty and therefore causing our Pod to crash and loop.

How to Fix It

Looking at our `httpd.conf` file's line 178, we can see the entry for the missing file.

```
178 Include conf/Suite-extra/components.conf
```

The `Include` directive in the configuration of Apache allows you to include additional configuration files. In this case, the file referenced components.conf does not exist. There are a couple of different ways to solve the issue. Usually, this means making sure the file exists and is accessible. In this scenario, we've carried over the `Includes` directive from another environment, and this line is not required for our application. First, we'll comment out the line in the reference filesystem's `httpd.conf`.

`httpd.conf`:

```
…
Include conf/Suite-extra/components.conf
```

Then, we'll delete and recreate the ConfigMap.

```
kubectl delete configmap httpd-cm -n demo
kubectl create configmap httpd-cm --from-file=httpd.conf -n demo
```

Finally, we'll redeploy the application.

```
helm uninstall apache
helm install apache bitnami/apache --set httpdConfConfigMap=httpd-cm
```

Running `kubectl get pods -n demo --watch now` shows us the container creating and staying healthy with no crashes.

```
NAME                      READY       STATUS              RESTARTS     AGE
apache-d4b5bc55c-6k4gs    0/1         ContainerCreating   0            1s
apache-d4b5bc55c-6k4gs    0/1         Running             0            4s
apache-d4b5bc55c-6k4gs    1/1         Running             0            40s
```

Run Phase Errors

Run phase errors show up once the Pod and container startup dependencies have been satisfied and once the initialization of the Pod and container has completed successfully. In this phase, the core functions of the application are in full swing. The reasons why the applications crash are quite varied, with the causes ranging from resource saturation in the underlying infrastructure, memory leaks, fatal exceptions in the container's running code, Pod evictions, and many, many more. Some examples of errors and crash causes during this phase include:

- Node Pressure Eviction
- Pod Priority Preemption
- NodeNotReady Pod Eviction
- Container Application Exit

Let's take a look at some of these error types in more detail.

Container Application Exit

Applications can exit/crash for a myriad of reasons. Some exits are expected, such as an application gracefully completing its execution, and others are unexpected, such as a fatal exception in the code from an unhandled error. In either of these scenarios, Kubernetes still handles these exits/crashes with its CrashLoop construct

Here we'll cover a scenario where a container that has already started and is in a running state makes an API call out to a service. When the call fails, the container exits. The CrashLoop then begins.

How it Manifests

We've deployed one of OpsCruise's own application components. It's what we call a Gateway, a lightweight Pod that receives metrics from Prometheus via remote write, filters, compresses, and then securely streams them out to a backend.

Since this Pod is part of a Helm deployment, we'll run the adequate command to get the Pod deployed.

```
helm install opscruise-bundle oc-repo/opscruise -f values.yaml
```

We now see the Pod successfully started and running.

```
NAME                                READY     STATUS              RESTARTS   AGE
opscruise-loggw-d649f47c5-nrl5q     1/1       Running             0          40m
```

However, after letting it run for some time, if we run **`kubectl get pods`**, we notice that the Pod is continually restarting in the CrashLoop.

```
NAME                                READY     STATUS              RESTARTS   AGE
opscruise-loggw-d649f47c5-nrl5q     0/1       CrashLoopBackOff    40         4h5m
```

As before, we can run **`kubectl describe`** to get details about what happened. Running **`kubectl describe pod opscruise-loggw-d649f47c5-nrl5q`** shows us the following:

```
    Containers:
      loggw-loki:
         Last State:   Terminated
         Reason:       Error
         Exit Code:    1
```

Exit Code 1 means the container suffered an application level error and exited. Knowing this is an application issue, it's time to look at the logs. Running **`kubectl logs opscruise-loggw-d649f47c5-nrl5q`** shows us the following:

```
2022-02-18 05:22:15 INFO - Endpoint: ws://opscruise-bundle-loki:3100
2022-02-18 05:22:15 INFO - Initializing loki websocket listener
2022-02-18 05:25:07 ERROR - IO Error while listening in web-socket connection
2022-02-18 05:25:07 WARN - Exiting. Could not connect to loki after 10 retries.
```

How to Fix It

The contents of the error provide us some indication as to the root cause. We see an attempted connection to a web-socket over port 3100. However, we see an ERROR level entry stating there was an IO error with the web-socket connection. Then, the container exits after trying 10 times to connect to the service.

The cause of connection errors can be too many to enumerate here. However, some examples include DNS resolution errors, routing errors, firewall errors, and misconfiguration errors, among others.

In this cluster, we happen to have Network Policies in place that only allow specifically defined traffic to travel across the cluster network. We can run **`kubectl get networkpolicies.projectcalico.org loggw-traffic -o yaml`** to see the policy that manages traffic for this Pod. We see the following:

```
    spec:
      egress:
      - action: Allow
      destination:
         ports:
         - 8443
         - 9093
      protocol: TCP
```

As a reminder, if a Network Policy contains an Egress rule, all traffic is automatically blocked, except for specifically allowed traffic. This behavior, combined with the output of the Network Policy become a clear indicator into the likely root cause: port 3100 is not a specifically allowed port for egress traffic.

To fix it, we can simply add port 3100 to the egress rule of the Network Policy. Now when we run **`kubectl get networkpolicies.projectcalico.org loggw-traffic -o yaml`** again, we see that port 3100 has been added to the allowed list.

```
    spec:
      egress:
      - action: Allow
      destination:
         ports:
         - 8443
         - 9093
         - 3100
      protocol: TCP
```

Running **`kubectl logs opscruise-loggw-d649f47c5-nrl5q`** now shows a successful connection to port 3100 of the Loki endpoint.

```
2022-02-18 07:02:13 INFO - Endpoint: ws://opscruise-bundle-loki:3100
2022-02-18 07:02:13 INFO - Initializing loki websocket listener
2022-02-18 07:02:15 INFO - opening websocket
2022-02-18 07:02:15 INFO - Successfully initiated log tailing using web-socket
```

After letting it run for some time, if we run **`kubectl get pods`**, we now see that although additional time has passed, the restart counter has stopped increasing and our Pod is no longer crashing.

```
NAME                                READY        STATUS       RESTARTS     AGE
opscruise-loggw-d649f47c5-nrl5q     1/1          Running      41           5h3m
```

Healthy Pod Eviction and Preemption

Kubernetes has the ability to "pre-empt" and evict Pods. The reasons why it might do so are related to scheduling and maintaining the health of the worker nodes.

When a worker node's resources, such as memory, have been over-allocated and the node is running low, the Kubernetes scheduler has to make decisions around what workloads should be terminated to keep resource utilization healthy. Kubernetes has soft and hard eviction thresholds that trigger Pod eviction in order to reclaim the resource. Below are the list of hard eviction thresholds:

- memory.available < 100Mi
- nodefs.available < 10%
- imagefs.available < 15%
- nodefs.inodesFree < 5% (Linux nodes)

Kubernetes also has the concept of workload prioritization. With a Priority Class, users can define a Pod's scheduling priority in the cluster. When there are not enough resources to allocate a new Pod with a higher priority relevant to some of the running workloads in the cluster, the Kubernetes scheduler pre-empts (removes) an existing Pod of lower priority to make room for the higher priority Pod.

In both cases, Pods might be perfectly healthy in the context of the running application, but because of the way Kubernetes handles scheduling and priority, Pods may still be terminated.

Below, we'll cover an instance where a running, errorless Pod is terminated.

How it Manifests

We've created a Pod inside a Deployment that consumes most of a worker node's memory.

```
high-mem-no-priority.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: demo
  name: high-mem-no-priority
   spec:
      containers:
        - name: no-priority
          image: k8s.gcr.io/busybox
          command: [ "/bin/sh", "-c", "while [ true ]; do echo 'hello'; sleep 5;
done" ]
          resources:
            requests:
              memory: "5Gi"
      nodeSelector:
        kubernetes.io/hostname: worker3
```

You can see the Pod running by executing `kubectl get pods`.

```
NAME                                    READY     STATUS       RESTARTS       AGE
high-mem-no-priority-56d99dddff-95fjm   1/1       Running      0              75s
```

In addition, we've scheduled a second Pod that has a higher memory request than the first. Since the newer Pod has a higher request, we would expect it not to be scheduled, as there are not enough resources. However, when running `kubectl get pods`, we notice that the first Pod has gone into a Pending state, while the second Pod is now running successfully.

```
NAME                                    READY       STATUS       RESTARTS       AGE
high-mem-no-priority-56d99dddff-lkx7h   0/1         Pending      0              50s
high-mem-with-priority-74dcf856c-tnwgc  1/1         Running      0              52s
```

How to Fix It

Up front, we knew the worker node did not have enough memory to schedule both workloads. At first glance, it would seem like the second Pod would have to wait until resources freed up before it would be scheduled. However, as we saw, the second Pod immediately came up, pushing the first one aside. Why did this happen?

Let's take a look at the second Pod's manifest.

```
high-mem-with-priority.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: demo
  name: high-mem-with-priority
spec:
  selector:
    matchLabels:
      app: high-mem-with-priority
  template:
    metadata:
```

```
 labels:
        app: high-mem-with-priority
    spec:
      containers:
        - name: high-mem-with-priority-container
          image: k8s.gcr.io/busybox
          command: [ "/bin/sh", "-c", "while [ true ]; do echo 'hello'; sleep 5;
done" ]
          resources:
            requests:
              memory: "6Gi"
      nodeSelector:
        kubernetes.io/hostname: worker3
      priorityClassName: high-priority
```

There are two things to call out:

1. The second Pod utilizes a Priority Class named "high-priority"
2. The second Pod requests more memory (6Gi) than the first Pod (5Gi)

Priority Classes assign workloads a numeric priority in the cluster, and the Kubernetes scheduler takes these into account when deciding where to schedule and unschedule Pods. (Default priority for all Pods is 0.)

Below is the Priority Class manifest.

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 100
description: "High Priority Class"
```

You can see the created Priority Class by running `kubectl get priorityclass`.

```
NAME                      VALUE          GLOBAL-DEFAULT        AGE
high-priority             100            false                 53m
system-cluster-critical   2000000000     false                 7d2h
system-node-critical      2000001000     false                 7d2h
```

The Priority Class is the key as to why the first Pod was preempted, even when it was already running and had a lower memory request. As mentioned earlier, the default priority for all Pods is 0. When the second Pod requests to be scheduled, the lower priority Pods running on the worker node that the scheduler has decided to place the newer Pod on will be preempted.

The fix is now clear, though there are a few options. A few solutions include:

- Reduce the request of one or both Pods to allow the node to run both workloads
- Change the nodeSelector configuration to allow the workload to be scheduled on other nodes
- Scale up the selected nodes so that there is enough compute for both workloads
- Match the priority on both workloads so scheduling one doesn't preempt (kick out) the other

Kubernetes Troubleshooting with OpsCruise

2

Troubleshooting workloads in Kubernetes is complex and, without the right tools, can be stressful, ineffective and time-consuming. This is why we created OpsCruise, a tool that helps Development and Operations teams by acting as a single source of truth for all of your K8s troubleshooting needs.

Scenarios like the ones above are extremely common in modern environments. OpsCruise automates and reduces the amount of manual work in these and other scenarios, providing immediate insights an, in many cases, immediate root cause analysis, freeing up your team, your time, and helping resolve costly outages faster. OpsCruise also offers:

KUBERNETES AS THE FOCUS.

OpsCruise is designed with the view of Kubernetes as the foundation of the application architecture. It is not an add-on to legacy monitoring designs. On-prem and cloud infrastructure is wired into the cluster which includes the ingress elements like Load Balancers, SaaS external services and Cloud services.

CURATED KUBERNETES KNOWLEDGE, BUILT-IN.

Like having a dedicated SME, OpsCruise's ingrained understanding of Kubernetes means precise insights into Out of Memory killed containers, CPU Throttling, Node Health and Balancing, Container Restarts, Deployment Health, Autoscaling, and more.

REAL-TIME DISCOVERY.

Because of its autoscaling and dynamic structure capabilities, all monitoring of Kubernetes *must* be automatic and immediate. OpsCruise auto-discovers your container, cloud, and multiple Kubernetes cluster environments, without proprietary agents, sidecars, or requiring code instrumentation.

OPERATIONAL FLOW TRACING.

A Kubernetes cluster is a distributed application environment, with 30-40% of an application spent in networking. OpsCruise provides a unique auto-instrumented eBPF based flow tracing feature that tracks topology changes in real time.

CLUSTER BALANCING INSIGHTS.

CI/CD deployments into the Kubernetes cluster can dynamically change the demands on the infrastructure resources. Kubernetes Pod limits may be invalid for the resources needed for the new container images. OpsCruise's real-time monitoring of the new Pod's usage in Pod Balancing view immediately identifies if the Pod is under-provisioned and should be migrated to nodes that have adequate capacity.

Learn More & Get Started

3

OpsCruise Intelligent Application Observability

OpsCruise's patented intelligent application observability platform is provided as a SaaS solution with gateways that sit on your infrastructure and collect metrics, logs, traces and configuration data from many popular open source monitoring tools (e.g., Prometheus, Elastic, Loki, Jaeger, etc.).

Our platform's deep understanding of Kubernetes, coupled with our unique ML-based behavior profiling empowers your entire team to predict performance degradations and instantly surface their cause. All at a third of the cost of the current monitoring stack and without the need to instrument code, deploy agents, or maintain open-source tools.

Get Started (for free) in Your Environment

You can sign up for our full feature *freemium* version and be up and running in less than 15 minutes.

GET STARTED