

# K8s Troubleshoot #02: Antidote for crashing or unschedulable pods

Check GitHub for helpful DevOps tools:

## Michael Robotics

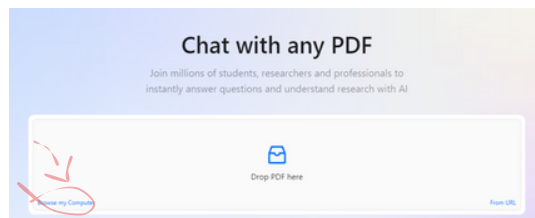
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

 <https://github.com/MichaelRobotics>



Ask Personal AI Document assistant to learn  
interactively (FASTER)!

- 1 Download PDF
  - 2 Go to website
  - 3 Browse file
  - 4 Chat with Document
- 1 <https://github.com/MichaelRobotics/DevOpsTools/blob/main/K8sTrouble02.pdf>
- 2 | Click there to go to ChatPdf website
- 3
- 4
- Ask questions about document!




## Completly new to Kubernetes?

If you are completely new to this topic, using a document assistant to understand the many definitions can be helpful. However, the best way to start is by watching this video, which I believe provides the best explanation for beginners starting their journey with Kubernetes

Kubernetes Crash Course for Absolute Beginners

Hands-On Kubernetes Tutorial | Learn Kubernetes in 1 Hour - Kubernetes Course for Beginners

 [https://www.youtube.com/watch?v=s\\_o8dwzRlu4&ab\\_channel=TechWorldwithNana](https://www.youtube.com/watch?v=s_o8dwzRlu4&ab_channel=TechWorldwithNana)



## What is Kubernetes Troubleshooting

Kubernetes troubleshooting involves diagnosing and resolving issues within a Kubernetes cluster, such as deployment failures, pod crashes, or network problems. It requires examining logs, monitoring system metrics, and analyzing cluster configurations to identify and fix the root causes of problems.



# kubernetes

## How Kubernetes troubleshooting is done?

Kubernetes troubleshooting involves using tools like **kubect**l to inspect pod logs, events, and resource statuses to diagnose issues. Additionally, analyzing cluster metrics and leveraging Kubernetes' built-in debugging features, such as **kubect**l **exec** to access pod shells, can help identify and resolve problems.

## When Kubernetes troubleshooting is done?

Kubernetes troubleshooting is performed when issues arise, such as

- ImagePullBackOff,
- CrashLoopBackOff errors,
- unschedulable pods,
- resource sharing conflicts,
- breaches of resource quotas or limits,
- problems with StatefulSets and Persistent Volumes
- security breaches due to faulty network policies.

This PDF will focus on second and third from the list: **CrashLoopBackoff** and **unschedulable pods**. I will touch briefly topic of **resource sharing conflicts**.

# System Requirements

- 2 CPUs or more
- 2GB of free memory
- 20GB of free disk space
- Internet connection
- ubuntu 22.04

**If you want to install it on a different cloud provider, ask in the comments and I'll provide a solution for you!**

## Kubernetes: Main components & packages

install kind

```
# For AMD64 / x86_64[ $(uname -m) = x86_64 ] && curl -Lo ./kind  
https://kind.sigs.k8s.io/dl/v0.24.0/kind-linux-amd64
```

mv binary

```
chmod +x ./kind  
sudo mv ./kind /usr/local/bin/kind
```

Create your kind cluster. Create file **kind-example-config.yaml**

```
# three node (two workers) cluster config
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```

Start your first kind cluster:

```
kind create cluster --config kind-example-config.yaml
```

If you've got error: **Failed joining worker nodes**

1) Update Docker

2) updated the ulimit for max\_user\_watches and max\_user\_instances to a higher value

```
echo fs.inotify.max_user_watches=655360 | sudo tee -a
/etc/sysctl.conf
echo fs.inotify.max_user_instances=1280 | sudo tee -a
/etc/sysctl.conf
sudo sysctl -p
```

3) If nothing works, reinstall kind and do step 1) and 2)

# K8s Troubleshoot #02: CrashLoopBackOff

## 1) Understand popular case study

A company deploys a microservice-based application in Kubernetes. One of the services starts showing the CrashLoopBackOff error soon after deployment. The service worked fine in the development environment, but once deployed to the production cluster, it keeps restarting.

## 2) Error explanation

CrashLoop refers to the cycle in which a container starts, crashes, and is then restarted repeatedly by Kubernetes. This cycle indicates that the container is consistently failing shortly after it is started, resulting in multiple restarts.

The BackOff part indicates that Kubernetes will keep trying to pull the image, with an increasing back-off delay.

Kubernetes raises the delay between each attempt until it reaches a compiled-in limit, which is 300 seconds (5 minutes).

## 3) Problem causes

After investigating, the team discovered

- Misconfigurations
- Errors in the Liveness Probes
- The Memory Limits are too low
- Wrong command line arguments
- Bugs & Exceptions

#### 4) Solution #1 - Check for misconfigurations

Misconfigurations can encompass a wide range of issues, from incorrect environment variables to improper setup of service ports or volumes. These misconfigurations can prevent the application from starting correctly, leading to crashes. For example, if an application expects a certain environment variable to connect to a database and that variable is not set or is incorrect.

Check this manifest file, I highlighted line which is responsible for error generation

```
apiVersion: v1
kind: Pod
metadata:
  name: example-app
spec:
  containers:
  - name: app-container
    image: myapp:latest
    env:
    - name: DATABASE_URL
      value: "wrong-database-url" # Incorrect database URL, causing a crash
    ports:
    - containerPort: 8080
```

#### 5) Solution #2 - Errors in the Liveness Probes

Liveness probes in Kubernetes are used to check the health of a container. If a liveness probe is incorrectly configured, it might falsely report that the container is unhealthy, causing Kubernetes to kill and restart the container repeatedly. For example, if the liveness probe checks a URL or port that the application does not expose or checks too soon before the application is ready, the container will be repeatedly terminated and restarted.

Check this manifest file; I have highlighted the lines that define the liveness probe. If the probe is set up incorrectly or returns an error status code, the pod will enter a CrashLoopBackOff state

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp-container
          image: nginx:1.21
          ports:
            - containerPort: 80
livenessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```



## 6) Solution #3 - The Memory Limits are too low

If the memory limits set for a container are too low, the application might exceed this limit, especially under load, leading to the container being killed by Kubernetes. This can happen repeatedly if the workload does not decrease, causing a cycle of crashing and restarting. Kubernetes uses these limits to ensure that containers do not consume all available resources on a node, which can affect other containers.

Check this manifest file. Container will experience an Out of Memory (OOM) error due to low memory and CPU limits and the Pod will enter a CrashLoopBackOff

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: crashloop-example
  labels:
    app: crashloopleftesting
spec:
  replicas: 1
  selector:
    matchLabels:
      app: crashloopleftesting
  template:
    metadata:
      labels:
        app: crashloopleftesting
    spec:
      containers:
        - name: crashloopleftesting
          image: abhishekf5/crashloopleftest:v2
          ports:
            - containerPort: 8000
      resources:
        limits:
          cpu: "25m"
          memory: "25Mi"
```

## 7) Solution #4 - Wrong Dockerfile command lines arguments

Containers might be configured to start with specific command-line arguments. If these arguments are wrong or lead to the application exiting (for example, passing an invalid option to a command), the container will exit immediately. Kubernetes will then attempt to restart it, leading to the `CrashLoopBackOff` status. An example would be passing a configuration file path that does not exist or is inaccessible.

Check this Dockerfile. Container runtime environment will exit immediately after first try to execute highlighted command, what will create `CrashLoopBackOff` error on the Pod level

```
# Leverage a bind mount to requirements.txt to avoid having to
copy them into
# into this layer.
RUN --mount=type=cache,target=/root/.cache/pip \
    --
    mount=type=bind,source=requirements.txt,target=requirements.txt
    \
    python -m pip install -r requirements.txt

# Switch to the non-privileged user to run the application.
USER appuser

# Copy the source code into the container.
COPY . .

# Expose the port that the application listens on.
EXPOSE 8000

# Run the application.
CMD python3 app!34.py
```

## 8) Solution #5 - Bugs & Exceptions

Bugs in the application code, such as unhandled exceptions or segmentation faults, can cause the application to crash. For instance, if the application tries to access a null pointer or fails to catch and handle an exception correctly, it might terminate unexpectedly.

Kubernetes, detecting the crash, will restart the container, but if the bug is triggered each time the application runs, this leads to a repetitive crash loop.

Check this flask app. Container runtime environment will exit immediately after first try to execute highlighted line, what will create CrashLoopBackOff error on the Pod level

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=8000)
```

# K8s Troubleshoot #02: Unschedulable pods

## 1) Understand popular case study

TechCorp Inc. is a mid-sized technology company that runs a microservices-based application on a Kubernetes cluster. Recently, the company experienced an issue where several critical pods were in a pending state with an "unschedulable" status.

## 2) Error explanation

Pending Pods: Pods in the cluster were not transitioning to the running state and remained in the "Pending" phase.

Error Messages: The Kubernetes dashboard and kubectl describe pod command showed errors related to "Unschedulable." phrase.

### 1) Solution #1 NodeSelector

Node Selector is a simple way to constrain pods to nodes with specific labels. It allows you to specify a set of key-value pairs that must match the node's labels for a pod to be scheduled on that node. Usage: Include a nodeSelector field in the pod's YAML definition to specify the required labels.

Typical use case: Pod should run on node with specific OS system or CPU architecture etc.

Check this manifest file; I have highlighted the lines that define nodeSelector. Only nodes with the label **bar** are designated to run this pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
      foo: bar
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

## 2) NodeAffinity

Node Affinity is a more expressive way to specify rules about the placement of pods relative to nodes' labels. It allows you to specify rules that apply only if certain conditions are met. Usage: Define nodeAffinity rules in the pod's YAML definition, specifying required and preferred node selectors.

Typical use case: A pod should run on a specific node, but if that node is not available, the pod can run on a different node.

Check this manifest file; I have highlighted the lines that define the preferred nodeAffinity. Nodes with the key and label **foo:bar** are preferred to run this pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      affinity:
      nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
        matchExpressions:
        - key: foo
          operator: In
          values:
          - bar
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

### 3) Taint

Taints are applied to nodes to repel certain pods. They allow nodes to refuse pods unless the pods have a matching toleration. Usage: Use `kubectl taint` command to apply taints to nodes. Include tolerations field in the pod's YAML definition to tolerate specific taints. 3 taints exist

**No schedule** - node becomes unscheduable

Typical use case: Specific node shouldnt have commisioned any nodes because update is performed.

```
kubectl taint nodes <node-name> key=value:NoSchedule
```

**No execute** - all pods on node are destroyed, node becomes unscheduable

Typical use case: Specific node shouldnt have commisioned any nodes and contains pods that are useless

```
kubectl taint nodes <node-name> key=value:NoExecute
```

**Prefered Noschedule** - Pods shouldnt be schedule on this node, unless there is not available node left

Typical use case: Specific node is underperforming and have some unresolved issues

```
kubectl taint nodes <node-name> key=value:PreferNoSchedule
```

## 4) Tolerations

Tolerations are applied to pods and allow them to schedule onto nodes with matching taints. They override the effect of taints. Usage: Include tolerations field in the pod's YAML definition to specify which taints the pod tolerates.

Typical use case: Some critical pods need to be run on specific nodes, even when they are tainted.

Check this manifest file, I highlighted all lines defines tolerations. Nodes with key and label **foo:bar** will have pods scheduled even if **NoSchedule** taint is active

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: no-schedule-toleration-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: nginx
      tolerations:
      - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
```



# Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.



<https://github.com/MichaelRobotics>



*PS.*

*If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!*