Troubleshooting and debugging of Kubernetes issues

As applications increasingly move to containerized environments, debugging and troubleshooting them effectively has become a critical skill. Whether you're deploying microservices on Kubernetes or running a single container locally, identifying and resolving issues promptly ensures high availability and performance.

This guide provides a comprehensive set of resources to help developers, DevOps engineers, and system administrators diagnose and fix common issues encountered in containerized applications. It focuses on Kubernetes-native constructs like Pods, Services, and StatefulSets, and includes practical techniques for interpreting container logs, analyzing failure messages, and gaining interactive access to running containers for deeper inspection.

In the sections that follow, you will find focused instructions and best practices on how to:

- Debug failing or unresponsive **Pods**
- Investigate issues in **Services** that prevent proper communication
- Resolve problems in **StatefulSets**, especially in stateful workloads
- Identify the root cause of **Pod failures**
- Troubleshoot **Init Containers** that don't complete successfully
- Examine **running Pods** for runtime issues
- Access a **shell inside a container** for interactive debugging

By following the guidance in this document, you'll be better equipped to quickly find and fix issues in your containerized environments, leading to more stable and reliable deployments.

Debug Pods

🧪 **Step 1: Check Pod Status**

To see all the Pods and their status:

kubectl get pods

Sample Output:

nginx

NAME READY STATUS RESTARTS AGE

myapp-pod 0/1 CrashLoopBackOff 3 2m

This tells you that something is wrong (e.g. CrashLoopBackOff means the Pod keeps crashing).

📋 Get More Details

To see what's really happening:

kubectl describe pod myapp-pod

This shows events, errors, and resource usage.

🚨 Common Pod Problems

❌ Pod is Pending

Reason:

- Not enough CPU or memory
- Wrong settings in the YAML
- hostPort conflicts

Fix:

- Add more resources (or delete unused Pods)
- Update your YAML file to request less CPU/memory

yaml

```
resources:
  requests:
    cpu: "100m"
    memory: "200Mi"
```

❌ Pod is in Waiting State

Reason:

- The container image could not be pulled

Fix:

- Check logs:

kubectl describe pod myapp-pod

Look for an error like:

Failed to pull image "myapp:v1": image not found

Tips to Fix:

- Make sure the image name is correct
- Push your image to Docker Hub or another registry
- Try pulling the image manually:

docker pull myapp:v1

❌ Pod is Stuck in Terminating

Reason:

- Kubernetes can't delete it due to webhooks or finalizers

Fix:

1. **List webhooks:**

kubectl get mutatingwebhookconfigurations

kubectl get validatingwebhookconfigurations

2. **Force delete the Pod:**

kubectl delete pod myapp-pod --grace-period=0 --force

❌ CrashLoopBackOff Error

Reason:

- The container crashes repeatedly

Fix:

- Check container logs:

kubectl logs myapp-pod

Sample error:

Error: Missing database URL

- Add missing environment variables in your YAML:

yaml

```
env:
  - name: DATABASE_URL
    value: "postgres://..."
```

🧾 Common YAML Errors

❌ Typo in YAML Keys

Wrong:

yaml

```
commnd: ["npm", "start"]  # ❌ Typo
```

Correct:

```
command: ["npm", "start"]
```

Tip: Validate your YAML before applying:

```
kubectl apply --validate=true -f mypod.yaml
```

🔍 Compare YAML With Created Pod

Sometimes what you apply and what Kubernetes actually creates are different.

kubectl get pod myapp-pod -o yaml > created-pod.yaml

diff mypod.yaml created-pod.yaml

🛠️ Deployment Created, But No Pods?

Run:

kubectl describe deployment myapp-deployment

Check for:

- Image not found
- No replicas available
- Incorrect specs

🌐 Debug Services

Problem: Pod is Running But App Not Reachable

✅ Step 1: Check if the Service has Endpoints

kubectl get endpoints

If it returns nothing, the Service can’t find any Pods.

✅ Step 2: Do Pod Labels Match Service Selectors?

Example Service:

yaml

```
selector:
  app: myapp
```

Check if your Pods have the same label:

kubectl get pods --selector=app=myapp

✅ Example Working Pod YAML

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
```

```
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: myapp:v1
      command: ["npm", "start"]
      env:
        - name: DATABASE_URL
          value: "postgres://localhost:5432/mydb"
      ports:
        - containerPort: 3000
```

🔧 Run Commands Inside a Pod (for Testing)

To test network or DNS issues, you can create a simple Pod:

```
kubectl run -it --rm --restart=Never busybox --image=busybox sh
```

Inside the pod, you can run:

```
nslookup hostnames
wget -qO- <Pod-IP>:<Port>
```

🔄 Create and Test a Sample Deployment

kubectl create deployment hostnames --image=registry.k8s.io/serve_hostname

kubectl scale deployment hostnames --replicas=3

Check running Pods:

kubectl get pods -l app=hostnames

🧪 Check Service & DNS

Does the Service Exist?

kubectl get svc hostnames

If not, create one:

kubectl expose deployment hostnames --port=80 --target-port=9376

Check DNS Resolution

Inside a Pod:

nslookup hostnames

Test Service By IP

wget -qO- <Service-IP>:80

Check EndpointSlices

kubectl get endpointslices -l k8s.io/service-name=hostnames

Make sure it lists IPs of your Pods.

🧱 **Debug a StatefulSet**

List StatefulSet Pods:

kubectl get pods -l app.kubernetes.io/name=MyApp

Write a Termination Message (for troubleshooting)

Create termination.yaml:

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
```

```
spec:
  containers:
    - name: termination-demo-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

Apply it:

```
kubectl apply -f termination.yaml
```

Check status:

```
kubectl get pod termination-demo
```

Get details including the termination message:

```
kubectl get pod termination-demo -o yaml
```

Look for:

yaml

```
lastState:
  terminated:
```

message: Sleep expired

Determine the Reason for Pod Failure

🌟 Understanding Why a Pod Failed in Kubernetes

When a container inside a Pod crashes or ends unexpectedly, it helps to know **why** it happened. Kubernetes has a built-in way to show you this info using something called a **termination message**.

✅ What is a Termination Message?

A **termination message** is a short text written by the container before it stops running. It explains why the container ended. Kubernetes saves this message so that tools (like kubectl, dashboards, or monitoring tools) can read it easily.

🛠️ Before You Start

Make sure:

- You have a working Kubernetes cluster.
- You have kubectl installed and connected to the cluster.
- The cluster has at least **2 worker nodes** (not just control-plane nodes).
- If you don’t have a cluster, try free playgrounds like:
    - Killercoda

- KodeKloud
- Play with Kubernetes

🧪 Let's Try It: Create a Pod That Writes a Termination Message

We'll create a simple Pod using this YAML file:

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
  - name: termination-demo-container
    image: debian
    command: ["/bin/sh"]
    args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

This Pod:

- Runs a Debian container.
- Sleeps for 10 seconds.
- Then writes **"Sleep expired"** to a special file called /dev/termination-log.

▶️ Run it:

kubectl apply -f https://k8s.io/examples/debug/termination.yaml

---

🔍 Check the Pod Status

Run this command until the Pod has **Completed**:

kubectl get pod termination-demo

Now, see more details about why it ended:

kubectl get pod termination-demo -o yaml

Look for a section like this:

yaml

```
lastState:
  terminated:
    exitCode: 0
    message: |
      Sleep expired
```

You can also extract **just the message**:

kubectl get pod termination-demo -o go-template="{{range .status.containerStatuses}}{{.lastState.terminated.message}}{{end}}"

👩‍🎨 What If There Are Multiple Containers?

Use this command to see which container ended and why:

```
kubectl get pod multi-container-pod -o go-template='{{range
.status.containerStatuses}}{{printf "%s:\n%s\n\n" .name
.lastState.terminated.message}}{{end}}'
```

🛠️ Customize Where to Write the Termination Message

By default, Kubernetes expects the message in this file:

/dev/termination-log

But you can change it using the terminationMessagePath field:

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: msg-path-demo
spec:
  containers:
  - name: msg-path-demo-container
    image: debian
    terminationMessagePath: "/tmp/my-log"
```

📌 Advanced Tip: Use Container Logs as a Backup

If the file is empty and the container fails, you can also tell Kubernetes to check the container logs using this setting:

yaml

```
terminationMessagePolicy: FallbackToLogsOnError
```

This means:

- If the file is empty **and** there's an error, Kubernetes will show the last part of the logs (up to 2048 bytes or 80 lines).

📚 **Summary**

- Use **termination messages** to find out why a container stopped.
- Kubernetes reads from /dev/termination-log by default.
- Customize the path or let Kubernetes use logs if needed.
- You can view these messages using kubectl.

Debug Init Containers

Introduction to Init Containers in Kubernetes

In Kubernetes, **Init Containers** are specialized containers that run before the main application containers in a Pod. They allow you to perform setup or initialization tasks that need to be completed before the main application starts. These tasks can include setting up files, preparing databases, performing migrations, or even waiting for other services to be available.

The main difference between Init Containers and regular containers is that Init Containers must complete successfully before the main containers are allowed to start. Init Containers run sequentially, meaning each one must complete before the next starts.

You can define Init Containers in the Kubernetes deployment.yaml file under the initContainers section. Once the Init Containers complete their tasks, the main containers (your application containers) will start.

Breakdown of deployment.yaml with Init Containers

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-web-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-web-app
  template:
    metadata:
      labels:
        app: node-web-app
    spec:
      # This is where the Init Containers are defined
```

```
  initContainers:
    - name: download-setup-files
      image: busybox
      command: ['sh', '-c', 'echo "Setup complete" > /tmp/setup.txt']
      volumeMounts:
        - name: setup-volume
          mountPath: /tmp
  containers:
    - name: node-web-app
      image: <your-dockerhub-username>/node-web-app
      ports:
        - containerPort: 3000
  volumes:
    - name: setup-volume
      emptyDir: {}
```

Explanation of Sections:

initContainers:

- **Purpose:** This section defines the Init Containers for your Pod. These containers run before the main application containers and perform

initialization tasks.

- **Example:**
    - Here, the Init Container is named download-setup-files.
    - It uses the busybox image to execute a simple shell command that writes "Setup complete" to a file (/tmp/setup.txt).
    - The Init Container mounts a volume (setup-volume) to /tmp, which allows data sharing with the main application containers.

containers:

- **Purpose:** This section defines the main application containers that will run after the Init Containers have successfully completed.
- **Example:**
    - In this case, the main container is named node-web-app, and it runs the Node.js application.
    - The node-web-app container listens on port 3000 and is configured to use an image from your Docker Hub account (<your-dockerhub-username>/node-web-app).

volumes:

- **Purpose:** Volumes are used to persist data between Init Containers and main containers. They are shared storage locations for containers within the same Pod.
- **Example:**

    - The setup-volume is an emptyDir volume, which exists temporarily during the Pod's lifecycle.

    - This volume is mounted at /tmp for both the Init Container and the main container, allowing them to share the setup.txt file created by the Init Container.

Key Points:

- **Init Containers** are defined within the spec.template.spec.initContainers section in the deployment.yaml file.

- **Sequential Execution:** Init Containers run sequentially before the main containers start. Each Init Container must complete successfully before the next starts.

- **Data Sharing:** Volumes allow Init Containers and main containers to share data or files.

- **Lifecycle:** Init Containers perform setup tasks, and once they finish, the main application containers start.

Debugging Init Containers in Kubernetes

If you're facing issues with Init Containers, here’s how you can troubleshoot and debug them:

Checking the Status of Init Containers:

To see the current status of the Init Containers, run the following command:

kubectl get pod <pod-name>

Example output:

NAME READY STATUS RESTARTS AGE

<pod-name> 0/1 Init:1/2 0 7s

This shows that one out of two Init Containers has completed successfully.

More Detailed Status Information:

To get more detailed information about the status of the Init Containers, use:

kubectl describe pod <pod-name>

Example output:

yaml

Init Containers:

<init-container-1>:

State: Terminated

Reason: Completed

Exit Code: 0

Started: ...

Finished: ...

<init-container-2>:

State: Waiting

Reason: CrashLoopBackOff

Last State:

Reason: Error

Exit Code: 1

If any Init Container fails, this output will help identify the issue.

Accessing Logs of Init Containers:

To see the logs for a specific Init Container, run:

kubectl logs <pod-name> -c <init-container-2>

his will show you the logs for that Init Container. If the Init Container runs a script, you can use set -x in the script to print each command as it is executed, which helps in debugging.

Understanding Pod Status Values:

- **Init:N/M:** N Init Containers have completed, M Init Containers total.
- **Init:Error:** One or more Init Containers have failed.
- **Init:CrashLoopBackOff:** An Init Container has failed multiple times.
- **Pending:** The Pod has not yet started executing Init Containers.
- **PodInitializing or Running:** All Init Containers have finished successfully, and the Pod is now running.

Advantages of Using Init Containers in Production:

1. **Separation of Concerns**:
    - Init Containers allow you to separate initialization tasks (like setting up configurations, verifying dependencies, etc.) from the main application container. This makes your application container simpler and more focused on its core responsibilities.
2. **Pre-Processing Tasks**:
    - You can use Init Containers for tasks like checking the readiness of external systems, initializing databases, preparing volumes, or fetching configuration files before the main container starts. This ensures your main container only starts when the environment is properly set up.
3. **Dependency Management**:
    - Init Containers help manage dependencies that your main containers rely on. For example, you can ensure that a database schema is created, or a service is available before the application begins running, reducing the chances of errors caused by missing dependencies.
4. **Improved Pod Reliability**:
    - Since Init Containers must complete successfully before the main containers start, they help ensure that the environment is fully ready, which can lead to fewer issues when the main application container starts.
5. **Isolation of Tasks**:
    - Init Containers run in isolation and don’t affect the runtime of the main container. If an Init Container fails, it doesn't affect the main application container, and it can be retried until successful. This

isolation simplifies debugging and recovery from errors.

6. **Resource Control**:
    - You can allocate separate resources (CPU, memory) to Init Containers, ensuring that they don't interfere with the resources needed by the main application container. This can be useful for tasks that require more compute power or time.

7. **Security**:
    - You can run initialization tasks with a different security context (e.g., more restricted permissions) compared to the main application containers. This is useful for tasks like downloading and verifying sensitive files or credentials.

Disadvantages of Using Init Containers in Production:

1. **Increased Startup Time**:
    - Since Init Containers must complete before the main containers can start, this introduces an additional delay in the startup process. In production, this could lead to increased downtime or slower service availability, especially if the Init Containers take a long time to finish.

2. **Complexity in Debugging**:
    - While Init Containers are isolated, debugging their failures can be more challenging. If an Init Container fails, the main application container won't start, and you may need to troubleshoot the Init Container using logs and status messages. If not configured properly, logging in Init Containers might be insufficient for effective

troubleshooting.

3. **Resource Consumption**:
    - Init Containers use resources (CPU, memory) during their execution. If not properly managed, this can lead to resource contention and could affect the performance of other components running on the same cluster, especially in resource-constrained environments.

4. **Stateful Dependencies**:
    - If Init Containers are used for tasks like creating or modifying databases, you must ensure that these tasks are idempotent (i.e., they don't create conflicts if they are repeated). Failure to design Init Containers in this way could cause inconsistent state or errors if tasks fail or are retried unexpectedly.

5. **Risk of Initialization Failures**:
    - If an Init Container fails (e.g., due to network issues, incorrect configurations, or dependency problems), the entire Pod's initialization process will fail, and the application won't start. This could lead to production downtime if not handled properly. A retry mechanism or monitoring system is needed to prevent permanent failures.

6. **Limited Support for Parallelism**:
    - Init Containers run sequentially. If you have multiple tasks to be completed before the main application starts, they will execute one after the other. This can be time-consuming if there are many independent initialization tasks that could be run in parallel.

7. **Limited Use for Dynamic Configurations**:

    - Init Containers are useful for static initialization tasks, but they are not suitable for dynamically changing configurations after the Pod has been started. For dynamic configurations that may change at runtime, other methods, such as environment variables or sidecars, are typically more appropriate.

Conclusion:

Advantages:

- Improve Pod reliability, manage dependencies, and isolate tasks.
- Simplify initialization by ensuring that tasks like database setup and external service checks are done before the main app starts.
- Enhance security by running tasks with different permissions.

Disadvantages:

- Increase startup time and complexity in debugging failures.
- Consume resources during initialization, potentially causing performance issues.
- Failures in Init Containers prevent the main container from starting, which can lead to production downtime if not managed properly.

Overall, **Init Containers are beneficial for specific use cases in production**, such as preparing the environment before the main application starts. However, they should be carefully implemented to avoid drawbacks like increased startup times and potential resource constraints.

Debug Running Pods

This guide explains how to debug Pods that are running or crashing on a Node in Kubernetes. It assumes that your Pod is already scheduled and running. If your Pod isn't running yet, you should first look into **debugging Pods**.

Prerequisites

- Your Pod must already be running. If it is not, start with debugging Pods that aren't scheduled or are in a pending state.

- You may need to know the Node on which the Pod is running and have shell access to that Node for some advanced debugging steps, but for basic steps, you only need kubectl access.

Step 1: Create a Pod using a Deployment

Let's start by creating a Deployment that runs two Pods. This example uses an **nginx** image.

yaml

application/nginx-with-request.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

```
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
```

```
    ports:
    - containerPort: 80
```

Now, apply the Deployment using:

kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml

This will create the deployment.

Step 2: Check Pod Status

To verify if the Pods are running correctly, run:

kubectl get pods

You should see something like this:

sql

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-67d4bdd6f5-cx2nz   1/1     Running   0          13s
nginx-deployment-67d4bdd6f5-w6kd7   1/1     Running   0          13s
```

This indicates both Pods are running.

Step 3: Inspect Pod Details Using kubectl describe pod

To get detailed information about a specific Pod, use:

kubectl describe pod nginx-deployment-67d4bdd6f5-w6kd7

Example output:

yaml

Name: nginx-deployment-67d4bdd6f5-w6kd7

Namespace: default

Node: kube-worker-1/192.168.0.113

Start Time: Thu, 17 Feb 2022 16:51:01 -0500

Status: Running

IP: 10.88.0.3

Controlled By: ReplicaSet/nginx-deployment-67d4bdd6f5

Containers:

nginx:

Container ID: containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616462a

Image: nginx

Port: 80/TCP

```
    State:          Running
    Started:        Thu, 17 Feb 2022 16:51:05 -0500
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     500m
      memory:  128Mi
    Requests:
      cpu:        500m
      memory:     128Mi
Events:
  Normal  Scheduled  34s   default-scheduler  Successfully assigned
default/nginx-deployment-67d4bdd6f5-w6kd7 to kube-worker-1
  Normal  Pulling    31s   kubelet            Pulling image "nginx"
  Normal  Pulled     30s   kubelet            Successfully pulled image "nginx"
  Normal  Started    30s   kubelet            Started container nginx
```

This gives you the status, resource requests, and events of the Pod.

Step 4: Debugging Pending Pods

Sometimes Pods may stay in a **Pending** state due to insufficient resources on the Node. For instance, a Pod may request more resources than available. You can see the status using:

kubectl get pods

Example:

nginx-deployment-1370807587-fz9sd 0/1 Pending 0 1m

To understand why the Pod is stuck in Pending, run:

kubectl describe pod nginx-deployment-1370807587-fz9sd

This will show an event like:

yaml

Events:

FirstSeen LastSeen Count From Type Reason Message

--- --- --- --- --- --- ---

1m 48s 7 {default-scheduler} Warning FailedScheduling pod (nginx-deployment-1370807587-fz9sd) failed to fit in any node

fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, requested: 1000, used: 1420, capacity: 2000

fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, requested: 1000, used: 1100, capacity: 2000

This tells you that the **Pod** could not fit on the **Node** due to insufficient resources, such as CPU.

Step 5: Scaling the Deployment

If you encounter a resource issue, you can scale the Deployment down to fit within available resources using:

kubectl scale deployment nginx-deployment --replicas=4

Step 6: View Events Across All Namespaces

If you want to see all events from all namespaces, run:

kubectl get events --all-namespaces

Step 7: View Pod Information in YAML

For a detailed view of a Pod, including its annotations and other configurations, use:

kubectl get pod nginx-deployment-1006230814-6winp -o yaml

This gives you complete information about the Pod's lifecycle, including resource limits, environment variables, and volume mounts.

By following these steps, you can effectively debug and understand the state of your Kubernetes Pods. Whether they are running or pending, the kubectl describe and kubectl get commands are powerful tools for troubleshooting.

Debug Running Pods

Before You Begin

Make sure your Pod is already scheduled and running. If your Pod isn't running yet, you should start by following steps to **debug Pods** first.

In some advanced cases, you'll need to know which Node your Pod is running on, and you may need shell access to that Node to run commands. However, you don’t need this access for the basic debugging steps we'll cover here using kubectl.

Using kubectl describe pod to Fetch Pod Details

We'll use a **Deployment** to create two Pods, similar to a previous example.

Here’s the YAML for creating a Deployment:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
```

```
  matchLabels:
    app: nginx
 replicas: 2
 template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
      ports:
      - containerPort: 80
```

Create the deployment by running the following command:

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml
```

This creates the deployment, and you can check the status of your Pods with:

```
kubectl get pods
```

The output might look like this:

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-67d4bdd6f5-cx2nz   1/1     Running   0          13s
nginx-deployment-67d4bdd6f5-w6kd7   1/1     Running   0          13s
```

To get detailed information about a specific Pod, use the kubectl describe pod command. For example:

kubectl describe pod nginx-deployment-67d4bdd6f5-w6kd7

This will display detailed information about the Pod, including:

- **Pod state**: Whether the Pod is running, waiting, or terminated.
- **Container status**: Information like when the container started, if it's ready, and how many times it has restarted.
- **Resource usage**: CPU and memory limits and requests.
- **Events**: Logs of any important system events related to the Pod.

For example, the description might show this:

```
Containers:
  nginx:
    Container ID:
containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616
462a
    Image:          nginx
    Port:           80/TCP
    State:          Running
      Started:      Thu, 17 Feb 2022 16:51:05 -0500
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     500m
      memory:  128Mi
    Requests:
      cpu:        500m
```

```
    memory:     128Mi
```

Example: Debugging a Pending Pod

Sometimes, Pods may fail to start or stay in a **Pending** state. For example, if you've created a Deployment with 5 replicas, but the requested resources (e.g., CPU or memory) are more than the available capacity on the nodes, one or more Pods might not be scheduled.

You can use the following command to check the status of all Pods:

```
kubectl get pods
```

You might see something like this:

```
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1       Running   0          7m
nginx-deployment-1006230814-fmgu3   1/1       Running   0          7m
nginx-deployment-1370807587-6ekbw   1/1       Running   0          1m
nginx-deployment-1370807587-fz9sd   0/1       Pending   0          1m
nginx-deployment-1370807587-fmgu3   0/1       Pending   0          1m
```

If a Pod is stuck in Pending state, use kubectl describe pod to understand the cause. For example:

```
kubectl describe pod nginx-deployment-1370807587-fz9sd
```

The output might show something like this:

```
Events:
  FirstSeen     LastSeen     Count   From                SubobjectPath   Type    Reason
Message
  ---------     ---------    -----   ----                -------------   -----   ------       -------
```

1m 48s 7 {default-scheduler} Warning FailedScheduling pod failed to fit in any node
fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, requested: 1000, used: 1420, capacity: 2000
fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, requested: 1000, used: 1100, capacity: 2000

In this case, the Pod couldn't fit on the nodes because there weren't enough available resources (CPU). To fix this, you might need to reduce the number of replicas or adjust the resource requests.

Viewing Events

If you need to see all events across your cluster, use:

kubectl get events

For events in a specific namespace, use:

kubectl get events --namespace=my-namespace

To see events from all namespaces, add --all-namespaces:

kubectl get events --all-namespaces

Additional Debugging Information

Another useful command is kubectl get pod -o yaml, which outputs all the information about a Pod in **YAML** format. This gives even more detailed data about the Pod than kubectl describe pod.

For example:

kubectl get pod nginx-deployment-1006230814-6winp -o yaml

This command will give you detailed YAML output, including annotations, volumes, and more. It’s especially useful for advanced debugging.

By following these steps and commands, you can effectively troubleshoot Pods in Kubernetes. The most common causes of issues like Pending Pods or crashing Pods can often be traced back to resource allocation, configuration issues, or node availability, and these tools will help you identify the root causes.

When working with containers, sometimes you need to troubleshoot or inspect what's going on inside them. If the container image includes debugging tools (common with Linux and Windows OS-based images), you can run commands directly within the container using kubectl exec. This allows you to interact with a running container inside a pod.

To use kubectl exec, the basic command looks like this:

kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}

Where:

- ${POD_NAME} is the name of your pod.
- ${CONTAINER_NAME} is the name of the container in the pod. This is optional if your pod has only one container.
- ${CMD} and ${ARGs} are the commands you want to run inside the container.

For example, to look at the logs from a running Cassandra pod, you can run:

kubectl exec cassandra -- cat /var/log/cassandra/system.log

If you need to run a shell inside the container, you can use the -i (interactive) and -t (allocate a terminal) flags:

kubectl exec -it cassandra -- sh

This will give you a shell where you can run more commands interactively inside the container.

Debugging with an Ephemeral Debug Container

What is an Ephemeral Container?

Sometimes, you need to debug containers that either have no shell or don't have debugging tools, such as "distroless" images. In such cases, ephemeral containers can be helpful. They are temporary containers that you can add to an already running pod for troubleshooting.

How to Use Ephemeral Containers

Starting with Kubernetes v1.25, ephemeral containers allow you to add a container specifically for debugging purposes.

Here's an example to get started:

First, create a simple pod with the pause container (it doesn't contain debugging utilities):

kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart=Never

Now, if you try to run kubectl exec on this pod, you'll see an error, because the pause image doesn't include a shell:

kubectl exec -it ephemeral-demo -- sh
Output: OCI runtime exec failed: exec failed: container_linux.go:346: starting container process caused "exec: \"sh\": executable file not found in $PATH": unknown

To troubleshoot this, you can add an ephemeral container with debugging tools, such as busybox, using kubectl debug. This command will attach the new container to the running pod:

kubectl debug -it ephemeral-demo --image=busybox:1.28 --target=ephemeral-demo

- --target=ephemeral-demo: This option ensures that the new container shares the process namespace of the existing container, which helps with debugging.

Once this command is executed, you'll be attached to the shell of the new ephemeral container:

To see the state of the newly created ephemeral container, you can use:

kubectl describe pod ephemeral-demo

You'll get details about the container, including its state, which will look something like this:

yaml

Ephemeral Containers:
  debugger-8xzrl:
    Container ID:
docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4028ffb2eb
    Image: busybox
    Image ID:
docker-pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f07a21d1c08084
    State: Running

When you're done with your debugging session, you can remove the pod by using:

kubectl delete pod ephemeral-demo

Key Notes:

- The -c ${CONTAINER_NAME} option is **optional** when your pod has only one container.

- Ephemeral containers are very useful when the original container image doesn’t have debugging tools, like with distroless images.

- The --target flag ensures that the ephemeral container shares the same process namespace as the target container for more accurate troubleshooting.

This way, even if the original container doesn't have the right tools or shell, you can still troubleshoot effectively using ephemeral containers.

🛠️ Debugging a Pod by Creating a Copy

Sometimes, a Pod doesn't work properly and it's hard to troubleshoot. For example:

- You can't run kubectl exec because the container doesn't have a shell.
- The container crashes as soon as it starts.

In such cases, **you can use kubectl debug to create a new version of the Pod that is easier to debug**.

🧪 Example 1: Add a Debugging Container to a Running Pod

Let's say you have a container running with a minimal image like busybox, but it doesn't have the tools you need.

Create a Pod using busybox:

kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d

Now, add a new container with Ubuntu (which has more tools) to the Pod copy:

kubectl debug myapp -it --image=ubuntu --share-processes
--copy-to=myapp-debug

What this does:

- Creates a copy of myapp called myapp-debug.
- Adds an Ubuntu container for debugging.
- Allows both containers to see each other's processes (--share-processes).

You'll get a terminal prompt inside the new Ubuntu container:

root@myapp-debug:/#

✅ **Note**: When you're done debugging, delete both Pods:

kubectl delete pod myapp myapp-debug

🧪 Example 2: Change the Command to Stop a Crashing App

Sometimes, your container crashes because the command is wrong or missing.

Let's simulate a crashing container:

kubectl run --image=busybox:1.28 myapp -- false

Check its status:

kubectl describe pod myapp

You'll see it crashes with CrashLoopBackOff.

To debug it, create a copy and change the command to run an interactive shell:

kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh

Now you can explore the container manually:

✅ Delete the Pods when finished:

kubectl delete pod myapp myapp-debug

🧪 **Example 3: Change the Container Image**

You may want to switch to a different image with debugging tools.

Create a Pod:

kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d

Now create a copy and use ubuntu image:

kubectl debug myapp --copy-to=myapp-debug --set-image=*=ubuntu

The *=ubuntu means: change **all** containers to use the Ubuntu image.

✅ Clean up:

kubectl delete pod myapp myapp-debug

🧪 **Debugging Directly on a Node**

If the Pod can't help you debug, you can open a shell directly on the Node (machine) it’s running on.

kubectl debug node/mynode -it --image=ubuntu

This creates a Pod on that Node. You'll get a prompt:

ruby

root@ek8s:/#

The Node’s root filesystem is available at /host.

✅ Clean up when done:

kubectl delete pod node-debugger-mynode-xxxxx

✅ Summary for Beginners

- kubectl debug is used to **create a copy of a Pod** with changes that make debugging easier.
- You can:
    - Add another container (like Ubuntu).
    - Change the image.
    - Change the command.
    - Even debug the Node directly.

If you're having issues with your application and the usual troubleshooting steps don't help, you can debug the **Node** (the machine where your application is running).

To do this, follow these steps:

Step 1: Run a Debug Pod on the Node

Use the following command to start an interactive Ubuntu shell on the Node named mynode:

kubectl debug node/mynode -it --image=ubuntu

- -it means you're starting an interactive terminal session.
- --image=ubuntu tells Kubernetes to use an Ubuntu container image.

When you run this, Kubernetes will:

- Create a new temporary Pod for debugging.
- Automatically name the Pod something like node-debugger-mynode-xxxxx.
- Mount the Node's file system under the path /host.
- Put the container into the host's network and process space (so it can "see" what's happening on the Node).

You'll see a prompt like this when it connects:

ruby

root@ek8s:/#

If you don’t see this prompt, press Enter.

Important Notes for Beginners:

- The **Node's file system** is mounted inside the debug container at /host.
- The container shares **network, process IDs, and IPC (inter-process communication)** with the Node.
- However, it’s **not privileged**, so:
    - Some process info might not be accessible.
    - Running chroot /host (to switch to the Node's file system) may not work.
- If you need full access (a privileged container), add --profile=sysadmin or create one manually.

Step 2: When You're Done, Delete the Debug Pod

Don’t forget to clean up by deleting the debug pod:

kubectl delete pod node-debugger-mynode-xxxxx

Replace node-debugger-mynode-xxxxx with the actual name shown when the pod was created.

Debugging a Pod or Node while applying a profile

When you're trying to debug a **Pod** or a **Node** in Kubernetes, you can use the kubectl debug command. This command lets you:

- Debug a **Node** using a special debug Pod
- Debug a **Pod** using an **ephemeral container** (a temporary container just for debugging)

While doing this, you can **apply a profile** to change how the debug container works. For example, you can give it more privileges or adjust its security settings.

There are **two types of profiles** you can use:

- **Static Profile** – Predefined settings (easiest to use)
- **Custom Profile** – Create your own settings in a YAML or JSON file

---

🔧 Static Profile

A **static profile** is a built-in template that sets up things like security options. You apply it using the --profile flag.

Available Profiles:

Profile	Description
legacy	Old settings (Kubernetes v1.22 behavior) – default if you don’t specify any profile
general	Recommended for general debugging
baseline	Follows PodSecurity baseline rules

restricted	Follows PodSecurity restricted rules
netadmin	Grants network admin capabilities
sysadmin	Grants full system admin (root) capabilities

> 📝 **Tip:** The legacy profile is default **but will be removed soon**. It's better to use general or another appropriate profile.

🧪 Example: Debug a Pod Using a Static Profile

1. **Create a Pod:**
   kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d

2. **Debug the Pod with an ephemeral container using the sysadmin profile:**
   kubectl debug -it myapp --image=busybox:1.28 --target=myapp --profile=sysadmin

3. **Inside the debug container, check the capabilities:**
   / # grep Cap /proc/$$/status

You should see something like:

makefile

```
CapPrm:    000001ffffffffff
CapEff:    000001ffffffffff
```

This means the container has **full privileges**.

4. **Check if the container is privileged:**

kubectl get pod myapp -o jsonpath='{.spec.ephemeralContainers[0].securityContext}'

Expected output:

json

{"privileged":true}

5. **Clean up when done:**
   kubectl delete pod myapp

🧩 Custom Profile

A **custom profile** lets you define your own container settings in a YAML (or JSON) file.

> ⚠️ **Note:** You can only modify part of the container spec — not the image, command, volumes, etc.

🧪 Example: Debug with a Custom Profile

1. **Create a Pod:**
   kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d

2. **Create a custom profile file (YAML):** Create a file called custom-profile.yaml with the following content:

yaml

```
env:
  - name: ENV_VAR_1
    value: value_1
  - name: ENV_VAR_2
    value: value_2

securityContext:
  capabilities:
    add:
      - NET_ADMIN
      - SYS_TIME
```

3. **Run the debug Pod with both --profile and your --custom file:** kubectl debug -it myapp --image=busybox:1.28 --target=myapp --profile=general --custom=custom-profile.yaml

4. **Check if your custom environment variables were added:** kubectl get pod myapp -o jsonpath='{.spec.ephemeralContainers[0].env}'

Expected output:

json

[{"name":"ENV_VAR_1","value":"value_1"},{"name":"ENV_VAR_2","value":"value_2"}]

5. **Check if your custom capabilities were applied:**
   kubectl get pod myapp -o
   jsonpath='{.spec.ephemeralContainers[0].securityContext}'

Expected output:

json

{"capabilities":{"add":["NET_ADMIN","SYS_TIME"]}}

6. **Clean up the Pod:**

kubectl delete pod myapp

Get a Shell to a Running Container

This guide will help you learn how to open a shell (terminal) inside a running container using kubectl exec.

✅ **Prerequisites**

Before starting, you need:

- A Kubernetes cluster (e.g., from **minikube**, **Killercoda**, **KodeKloud**, or **Play with Kubernetes**)
- kubectl installed and configured to connect to your cluster

---

🚀 Step 1: Create a Sample Pod

We'll create a Pod that runs an **nginx** web server.

Here's the YAML file (shell-demo.yaml) used to define the Pod:

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  hostNetwork: true
  dnsPolicy: Default
```

🛠️ Create the Pod:

```
kubectl apply -f https://k8s.io/examples/application/shell-demo.yaml
```

🔍 Step 2: Check if the Pod is Running

kubectl get pod shell-demo

🐚 Step 3: Get a Shell Inside the Container

Use this command to open a shell:

kubectl exec --stdin --tty shell-demo -- /bin/

> Tip: The -- is used to separate kubectl options from the command you want to run inside the container.

🧪 Step 4: Try Commands Inside the Container

Once inside the shell, try these basic Linux commands:

ls /
cat /proc/mounts
apt-get update
apt-get install -y tcpdump
tcpdump
apt-get install -y lsof
lsof
apt-get install -y procps
ps aux
ps aux | grep nginx

🌐 Step 5: Add a Web Page to Nginx

Create a file that nginx can serve:

echo 'Hello shell demo' > /usr/share/nginx/html/index.html

Send a request to nginx:

apt-get update
apt-get install curl
curl http://localhost/

You should see this output:

nginx

Hello shell demo

To exit the shell:

exit

🛠️ Run Single Commands Without a Shell

You don’t always need to open a shell. You can run single commands directly like this:

kubectl exec shell-demo -- env
kubectl exec shell-demo -- ps aux
kubectl exec shell-demo -- ls /
kubectl exec shell-demo -- cat /proc/1/mounts

🧩 If Your Pod Has Multiple Containers

Use the --container (or -c) option to choose which container to run the command in.

Example:

kubectl exec -i -t my-pod --container main-app -- /bin/

This will open a shell in the main-app container of the my-pod Pod.