kubernetes

CRASH RECOVERY GUIDE

techopsexamples.com

Govardhana Miriyala Kannaiah

I'm Govardhana Miriyala Kannaiah, a DevOps and Multi-Cloud Architect with over 17 years of IT experience.

Over the past year, I’ve built a LinkedIn community of 96k+ followers and launched NeuVeu, a consulting firm that has served 17+ clients - from enterprises like Hearst and Stanford University to small-scale companies like MapDigital.

I’ve noticed that while many run Kubernetes in production, they often struggle with recovering from crashes effectively.

This ***Kubernetes Crash Recovery Guide*** takes you from diagnosing failures to applying proven recovery techniques, helping you build the skills needed to restore your clusters quickly and minimize downtime.

https://www.techopsexamples.com/

Table of Contents

Introduction 4

Kubernetes Architecture Simplified 5

Kubernetes Mistakes Side Effects 10

Kubernetes POD Lifecycle 13

Understanding Kubernetes Logs 15

How To Handle Most common errors 22

ImagePullBackOff 22

CreateContainerConfigError 27

CreateContainerError 32

RunContainerError 39

CrashLoopBackOff 44

OOMKilled 48

Node Disk Pressure 51

Node Not Ready 54

POD Troubleshooting Tactics 61

Must know Troubleshooting Commands 65

1. Introduction

This guide covers the essentials of Kubernetes crash recovery, helping you troubleshoot issues with confidence. Whether you're new or experienced, it provides a clear path to understanding failures, fixing common errors, and keeping your clusters running smoothly.

Target Audience

This guide is for:

- **Beginners** who need a solid foundation before diving into advanced Kubernetes troubleshooting.

- **Experienced users** looking to sharpen their skills and fix any knowledge gaps.

Resources

To discover real world use cases, tech updates, and learning resources, check out:

- **Full Editions:** https://www.techopsexamples.com/

2. Kubernetes Architecture Simplified

Kubernetes, often called K8s (short for "Kubernetes," with "8" representing the eight letters between "K" and "s"), started its journey at Google.

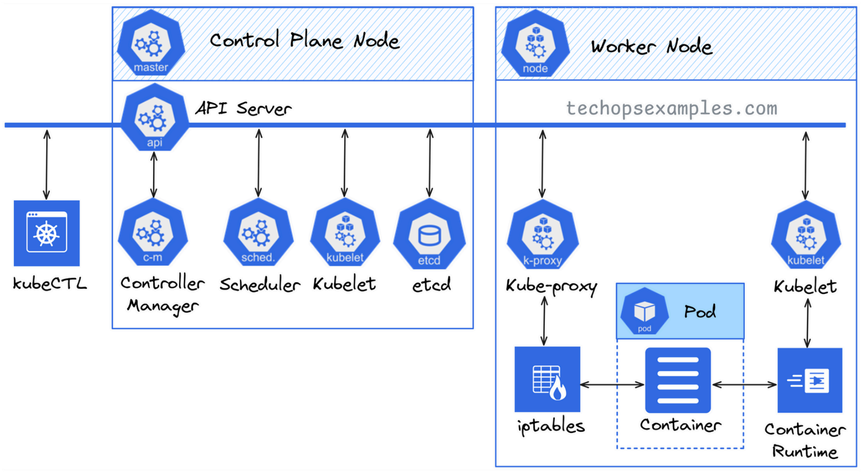
It was born from Google's internal system called Borg, which handled massive-scale workloads way before "cloud-native" was even a term.

In 2014, Google decided to open-source this powerful idea, and it became Kubernetes.

10 years ago, on June 6th, 2014, the first commit of Kubernetes was pushed to GitHub.

The Cloud Native Computing Foundation (CNCF) adopted it early on, turning Kubernetes into the poster child of cloud-native computing.

Let’s now break down the architecture, making sense of the key components with the below diagram.



The Control Plane Node: The Brain

This is where the decision-making happens. Every Kubernetes cluster has one or more control plane nodes that oversee everything in the cluster.

Here's how it all fits together:

- **API Server (api):** Think of it as the front desk of Kubernetes. Every *kubectl* command or internal component interaction goes through the API server. It validates your requests and routes them to the right place.

- **Controller Manager :** The automation genius. If your app's desired state (like 3 replicas) doesn’t match reality, the controller manager steps in to create, delete, or update resources.

- **Scheduler:** New pod? Cool. The scheduler finds the best worker node for it, considering factors like resources, affinity, and taints. It's all about optimal placement.

- **etcd:** The brain's memory. This is a highly consistent key-value store that keeps track of everything in the cluster. If etcd is down, Kubernetes forgets the cluster's state.

- **kubelet on Control Plane:** Just like on worker nodes, the kubelet on the control plane ensures containers running here are healthy and up-to-date.

Worker Node: The Muscles

While the control plane is busy planning and deciding, the worker nodes do the actual work.

Here's what happens under the hood:

- **kubelet:** The node's manager. It takes orders from the API server and ensures that containers (running inside pods) are healthy and doing what they're supposed to. It's like the node's personal assistant.

- **Kube-proxy:** Handles networking. It ensures every pod can talk to other pods and services inside (and sometimes outside) the cluster. It uses *iptables* or similar tools to manage network rules.

- **Container Runtime:** This is what runs the actual containers. Whether it's Docker, containerd, or CRI-O, it's all about keeping your apps alive and isolated.

- **Pods and Containers:** Pods are the smallest deployable units in Kubernetes. Each pod wraps one or more containers and shares networking and storage. The containers inside do the heavy lifting - running your application code.

Putting It All Together

🧠 The **control plane** makes the rules, stores the cluster's state, and sends commands.

💪 The **worker nodes** execute these commands, run your applications, and handle network requests.

It’s a perfect balance of brains and brawn, ensuring that your apps are highly available, scalable, and self-healing.

Want to explore more?

Run these commands on your Kubernetes cluster and see the magic live:

```
# Check control plane components
$ kubectl get pods -n kube-system

# Check worker node details
$ kubectl describe node <worker-node-name>
```

3. Kubernetes Mistakes Side Effects

99 little bugs in the code.
99 little bugs.
Take one down, patch it around.
127 little bugs in the code...

This is so relatable, isn’t it?

The side effects are so impactful, especially the hidden nature of it, which makes it such a lethal experience that everyone wishes to avoid.

The sad reality is that no one can completely.

Coming to Kubernetes, considering its already complex nature, it requires a strong understanding of the system and awareness of the radius of its impact.

With almost 5.6 million+ developers using Kubernetes globally and over 60% of enterprises adopting it, I find it more surprising that not much material is available on this **proactive** aspect.

Also, considering the nature of too many parameters and aspects in Kubernetes, it is really hard to create an encapsulated form of this.

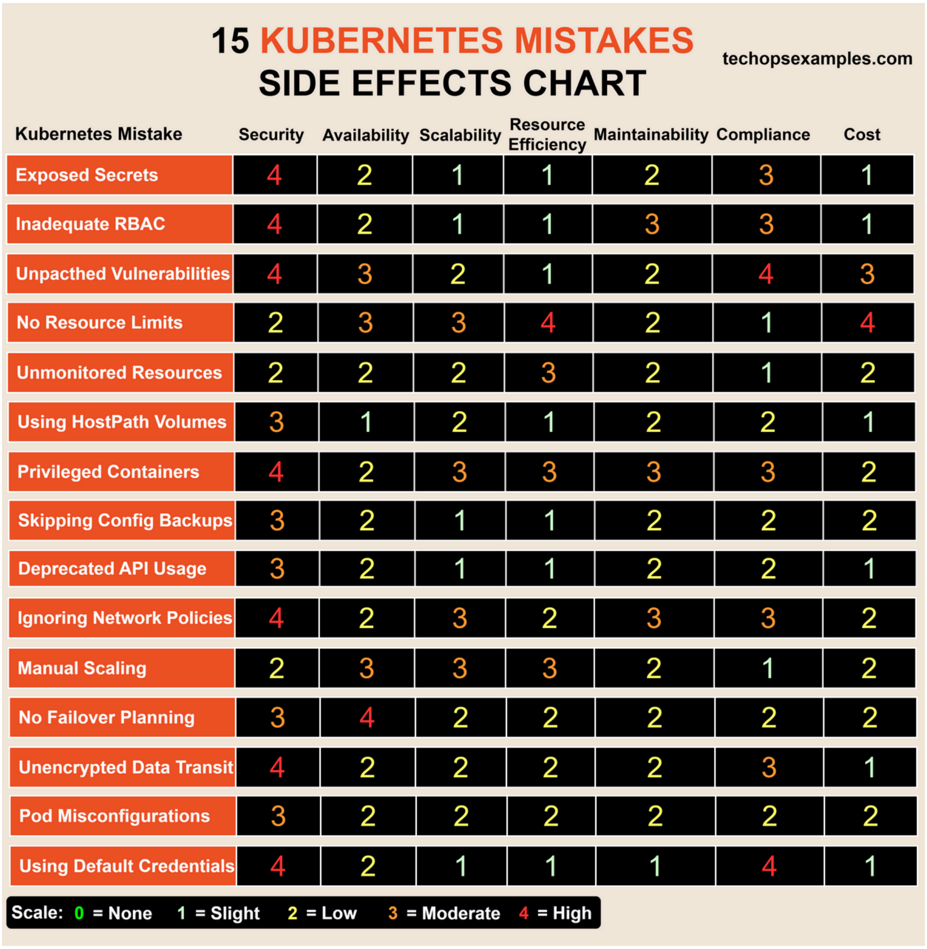
Of course, I am not a big fan of 100s of pages of theoretical explainer documents.

Here, I’ve tried my best to cover most Kubernetes aspects and scale the impact of a mistake across the seven prime categories - Security, Availability, Scalability, Resource Efficiency, Maintainability, Compliance, and Cost.

The idea here is to bridge these gaps and be aware of the impact on your system when you fail to do so.

An outage saved is proactive.

An outage addressed is reactive.



15 KUBERNETES MISTAKES
SIDE EFFECTS CHART

techopsexamples.com

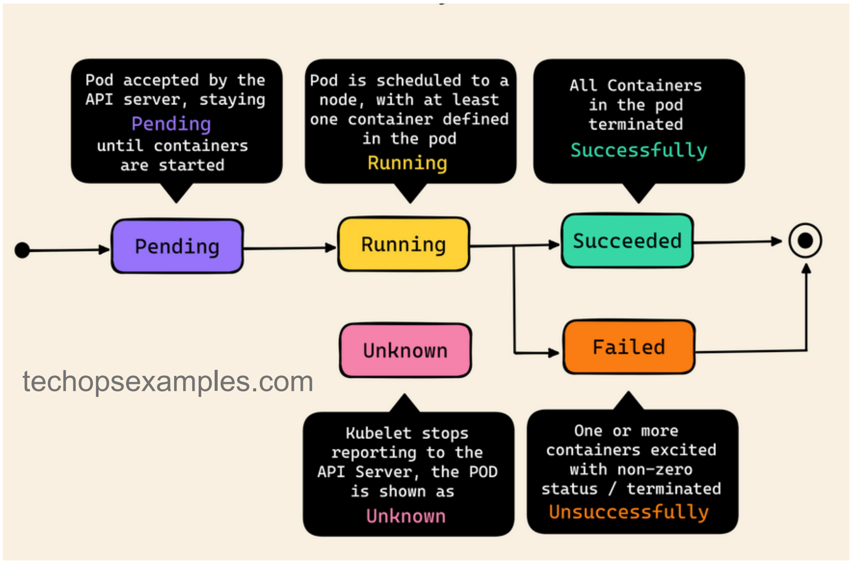
Kubernetes Mistake	Security	Availability	Scalability	Resource Efficiency	Maintainability	Compliance	Cost
Exposed Secrets	4	2	1	1	2	3	1
Inadequate RBAC	4	2	1	1	3	3	1
Unpacthed Vulnerabilities	4	3	2	1	2	4	3
No Resource Limits	2	3	3	4	2	1	4
Unmonitored Resources	2	2	2	3	2	1	2
Using HostPath Volumes	3	1	2	1	2	2	1
Privileged Containers	4	2	3	3	3	3	2
Skipping Config Backups	3	2	1	1	2	2	2
Deprecated API Usage	3	2	1	1	2	2	1
Ignoring Network Policies	4	2	3	2	3	3	2
Manual Scaling	2	3	3	3	2	1	2
No Failover Planning	3	4	2	2	2	2	2
Unencrypted Data Transit	4	2	2	2	2	3	1
Pod Misconfigurations	3	2	2	2	2	2	2
Using Default Credentials	4	2	1	1	1	4	1

Scale: 0 = None 1 = Slight 2 = Low 3 = Moderate 4 = High

4. Kubernetes POD Lifecycle

Understanding the lifecycle of a Kubernetes Pod is crucial for managing workloads effectively. It helps you anticipate how your applications behave, troubleshoot issues faster, and ultimately ensure smooth deployments in your clusters.

Below is a breakdown of the lifecycle stages that every Kubernetes practitioner should be aware of.


Pod accepted by the API server, staying Pending until containers are started
Pod is scheduled to a node, with at least one container defined in the pod Running
All Containers in the pod terminated Successfully
Pending
Running
Succeeded
Unknown
Failed
techopsexamples.com
Kubelet stops reporting to the API Server, the POD is shown as Unknown
One or more containers excited with non-zero status / terminated Unsuccessfully

- **Pending:** The Pod is accepted by the API server but remains in a pending state until all required containers start running.
- **Running:** The Pod is scheduled on a node, and at least one container inside the Pod is running.
- **Succeeded:** All containers in the Pod have successfully completed their tasks and exited.
- **Failed:** One or more containers terminated unsuccessfully with a non-zero exit status.
- **Unknown:** Kubernetes has lost communication with the node, and the Pod's state cannot be determined.

Why It Matters:

Pods in Kubernetes are ephemeral by nature, understanding these stages allows you to configure more reliable health checks, graceful termination, and better handle failure scenarios like CrashLoopBackOffs.

5. Understanding Kubernetes Logs

Not every day you would be setting up clusters; however, there is a huge possibility that every day you would be involved in the operational and troubleshooting aspects of Kubernetes.

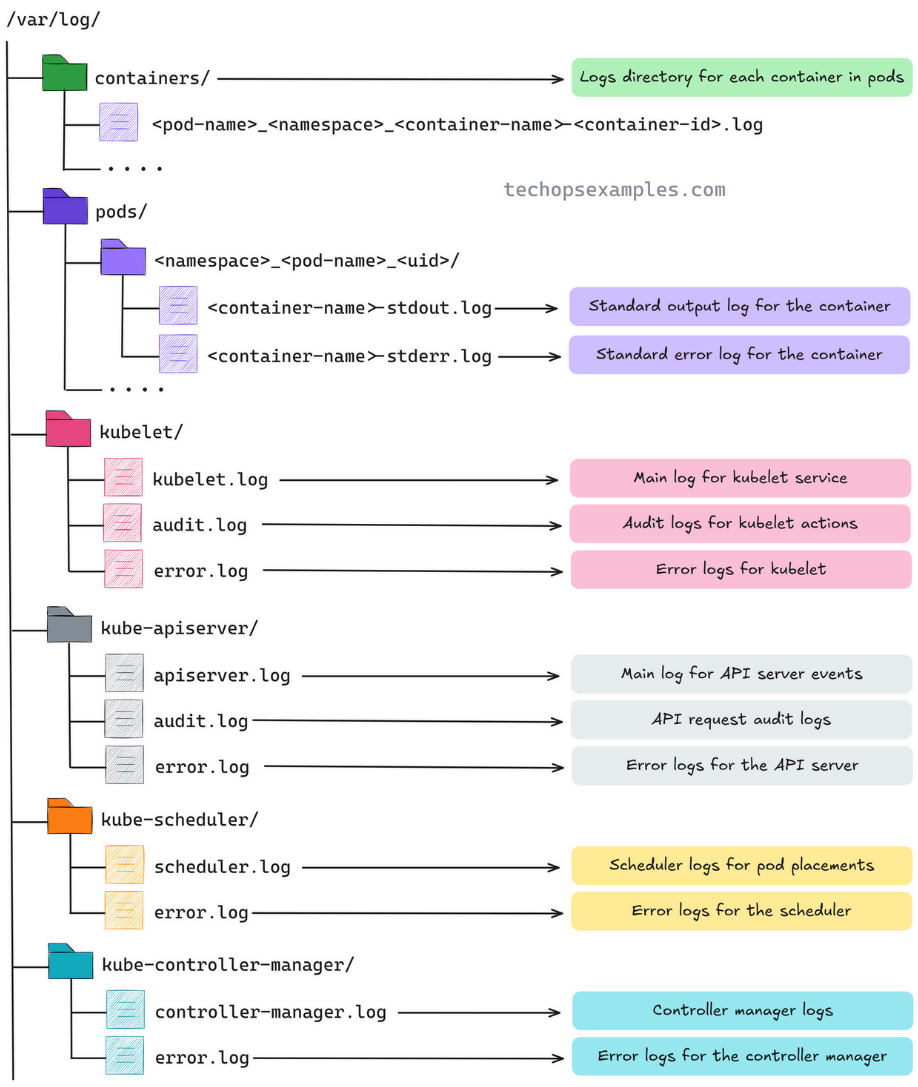
Understand the logs plays a crucial role in any Kubernetes practitioner's life.

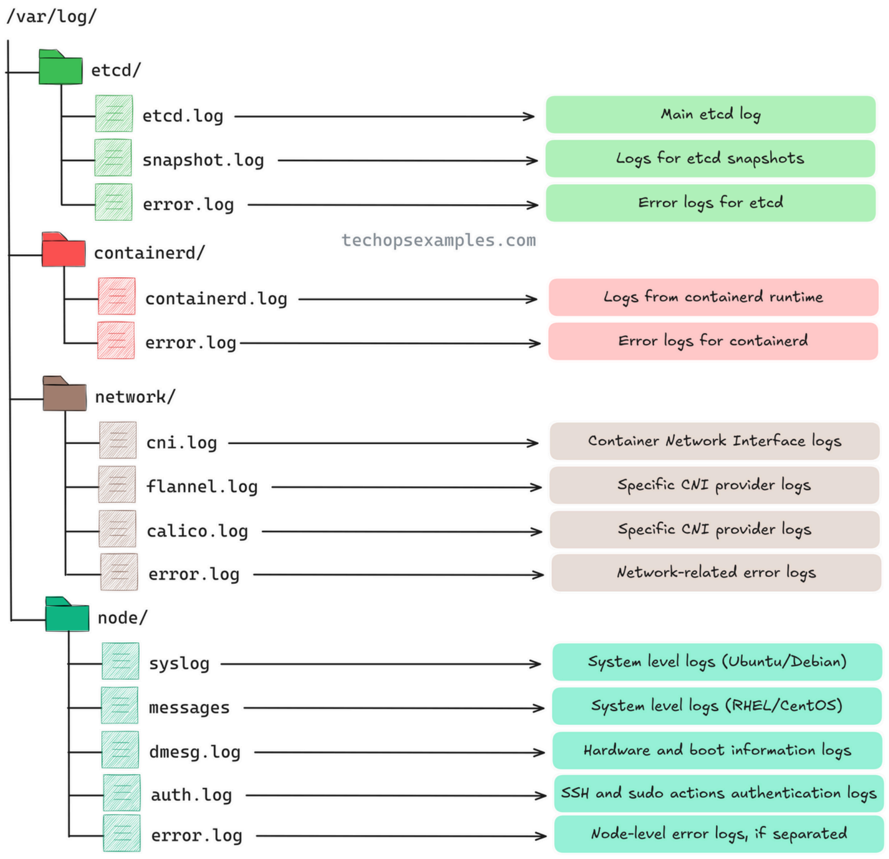
You can’t miss it or skip it.

I broke down the typical Kubernetes logs directory for easy understanding.

Like other directories, you can park the application logs under '/var/log/', which can further be organized by 'namespace’ and provide insights into the internal statuses of your application.

They are especially valuable for troubleshooting issues and tracking cluster events.


/var/log/
containers/
Logs directory for each container in pods
<pod-name>_<namespace>_<container-name>-<container-id>.log
....
techopsexamples.com
pods/
<namespace>_<pod-name>_<uid>/
<container-name>-stdout.log
Standard output log for the container
<container-name>-stderr.log
Standard error log for the container
....
kubelet/
kubelet.log
Main log for kubelet service
audit.log
Audit logs for kubelet actions
error.log
Error logs for kubelet
kube-apiserver/
apiserver.log
Main log for API server events
audit.log
API request audit logs
error.log
Error logs for the API server
kube-scheduler/
scheduler.log
Scheduler logs for pod placements
error.log
Error logs for the scheduler
kube-controller-manager/
controller-manager.log
Controller manager logs
error.log
Error logs for the controller manager


/var/log/
etcd/
etcd.log
Main etcd log
snapshot.log
Logs for etcd snapshots
error.log
Error logs for etcd
techopsexamples.com
containerd/
containerd.log
Logs from containerd runtime
error.log
Error logs for containerd
network/
cni.log
Container Network Interface logs
flannel.log
Specific CNI provider logs
calico.log
Specific CNI provider logs
error.log
Network-related error logs
node/
syslog
System level logs (Ubuntu/Debian)
messages
System level logs (RHEL/CentOS)
dmesg.log
Hardware and boot information logs
auth.log
SSH and sudo actions authentication logs
error.log
Node-level error logs, if separated

The log directory structure is quite straightforward; rather than spending time going through each one, let's talk about 'cluster level logging' - a less discussed and crucial aspect.

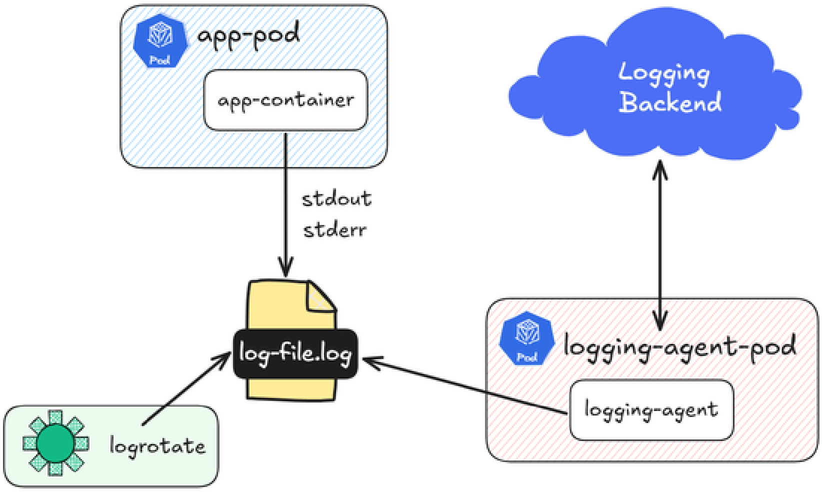
Cluster level logging architectures

Kubernetes natively doesn't provide log storage, so cluster level logging requires independent dedicated backend storage and lifecycle for logs, separate from nodes, pods, and containers.

Here are a few options:

- Run a logging agent on each node to collect logs.
- Add a sidecar container in the application pod specifically for logging.
- Send logs directly from the application to a logging backend.

1. Node logging Agent



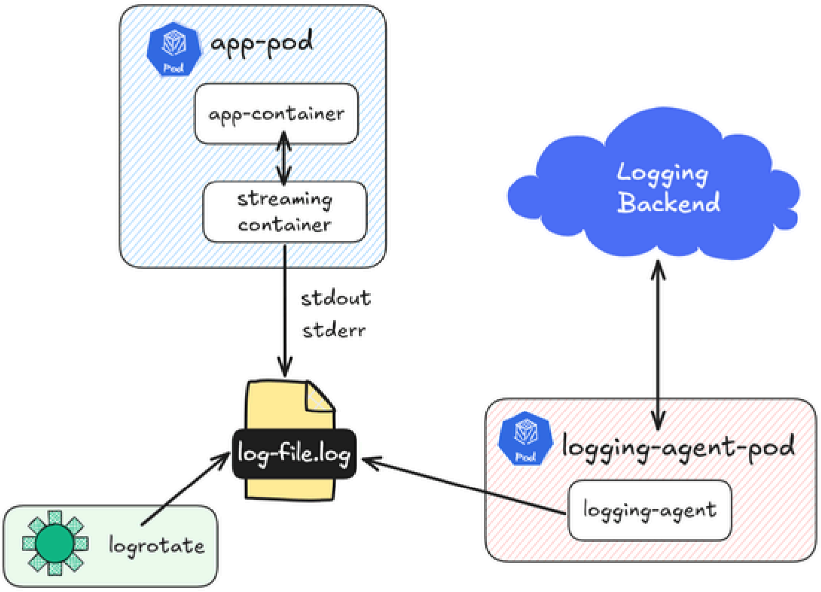
To implement cluster- evel logging, deploy a node-level logging agent on each node, typically as a DaemonSet, to collect and forward logs to a backend.

This agent, running as a container, accesses log directories from all application containers on the node.

Node level logging creates one agent per node without needing changes to applications.

Containers log to *stdout* and *stderr* in varied formats, which the node level agent gathers for aggregation.

2. Streaming Sidecar Container



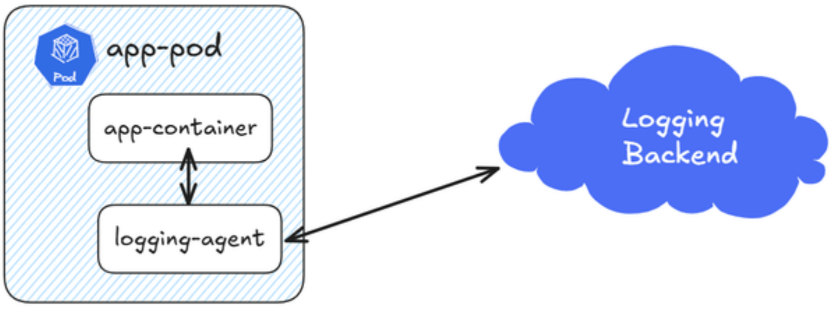
By configuring sidecar containers to write to their own stdout and stderr streams, you can leverage the kubelet and node-level logging agent already running on each node.

Sidecars read logs from sources like files, sockets, or journald and output to separate streams, enabling multiple log streams for different application parts.

This setup supports components that don't natively log to stdout or stderr, with minimal redirection overhead.

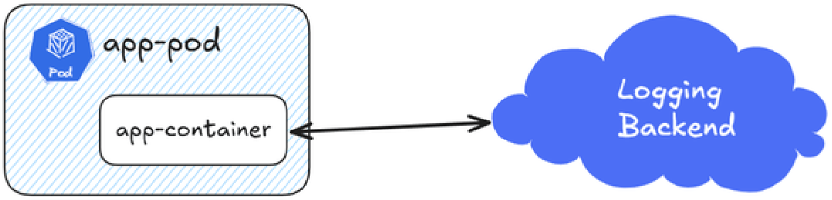
Since kubelet manages *stdout* and *stderr*, you can easily access logs using tools like *kubectl logs.*

3. Sidecar Container With a Logging Agent



If the node level logging agent doesn't meet your needs, you can add a sidecar container with a logging agent set up specifically for your application.

4. Pushing Logs Directly From The Application



Logging that involves directly exposing or pushing logs from each application is outside the scope of Kubernetes, though it can be an option for specific cases, like when an application requires direct integration with an external logging service.

6. How To Handle Most Common Errors

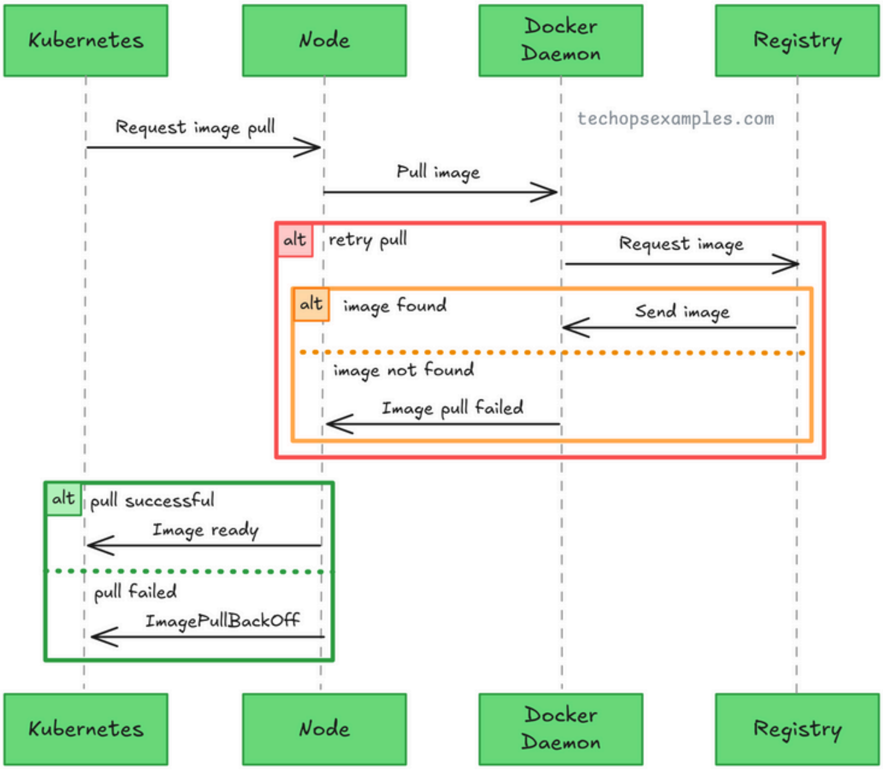
ImagePullBackOff

For Kubernetes practitioners, encountering the ImagePullBackOff error is a common challenge that is inevitable.

The Kubernetes ImagePullBackOff error happens when the system fails to retrieve the required container image. When this occurs, the container remains in a Waiting state, unable to proceed with deployment.

What exactly happens during an ImagePullBackOff?

- Kubernetes requests the node to pull the image.
- Node sends a request to Docker Daemon to pull the image.
- Docker Daemon requests the image from the registry.
- If the image is found, the registry sends the image back to Docker Daemon.
- Docker Daemon pulls the image to the node.
- If the image is successfully pulled, Kubernetes marks the image as ready.
- If the image is not found, Docker Daemon reports the failure.
- Kubernetes retries pulling the image.
- If the **retry fails**, Kubernetes enters **ImagePullBackOff** state, preventing further retries.



How to detect ImagePullBackOff?

1. View Pods Status

Check if the pod is experiencing issues by viewing its status.

```
$ kubectl get pods
NAME          READY    STATUS          RESTARTS    AGE
api-pod       0/1      ErrImagePull    0           25s
```

Wait a moment, then run the get pods command again: Confirm if the status has changed to ***ImagePullBackOff*** after a few seconds.

```
$ kubectl get pods
NAME         READY    STATUS             RESTARTS    AGE
api-pod      0/1      ImagePullBackOff   0           1m5s
```

2. Inspect the Pod

Get detailed information on why the image pull is failing.

```
$ kubectl describe pod api-pod
Events:
Type     Reason     Age                 From                 Message
----     ------     ----                ----                 -------
Normal   Scheduled  50s                 default-scheduler
Successfully assigned default/api-pod to minikube
Normal   Pulling    25s (x2 over 45s)   kubelet              Pulling
image "techopsexamples.com/api-service:v2.5"
Warning  Failed     15s (x2 over 40s)   kubelet              Failed to
pull image "techopsexamples.com/api-service:v2.5": Error response
from daemon: pull access denied for example.com/api-service,
repository does not exist or may require 'docker login': denied:
requested access to the resource is denied
Warning  Failed     15s (x2 over 40s)   kubelet              Error:
ErrImagePull
Normal   BackOff    5s (x2 over 40s)    kubelet              Back-off
pulling image "techopsexamples.com/api-service:v2.5"
Warning  Failed     5s (x2 over 40s)    kubelet              Error:
ImagePullBackOff
```

Fixing ImagePullBackOff

1. Fix Manifest or Image Name

Correct typos or incorrect image names in the pod manifest.

```
apiVersion: v1
kind: Pod
metadata:
  name: api-pod
  labels:
    app: api
spec:
  containers:
  - name: api-container
    image: techopsexamples.com/api-service:v2.5
    ports:
    - containerPort: 8080
```

Apply this manifest again: Apply the corrected manifest to update the pod.

```
$ kubectl apply -f pod.yaml
pod/api-pod configured
```

Check pod status: Verify that the pod is now running successfully.

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
api-pod     1/1     Running   0          1m
```

2. Non existent Image Validation

Ensure the image is available in the registry before pulling.

```
$ docker push techopsexamples.com/api-service:v2.5
```

3. Image Registry Authorization

Create a secret with credentials to access the private registry.

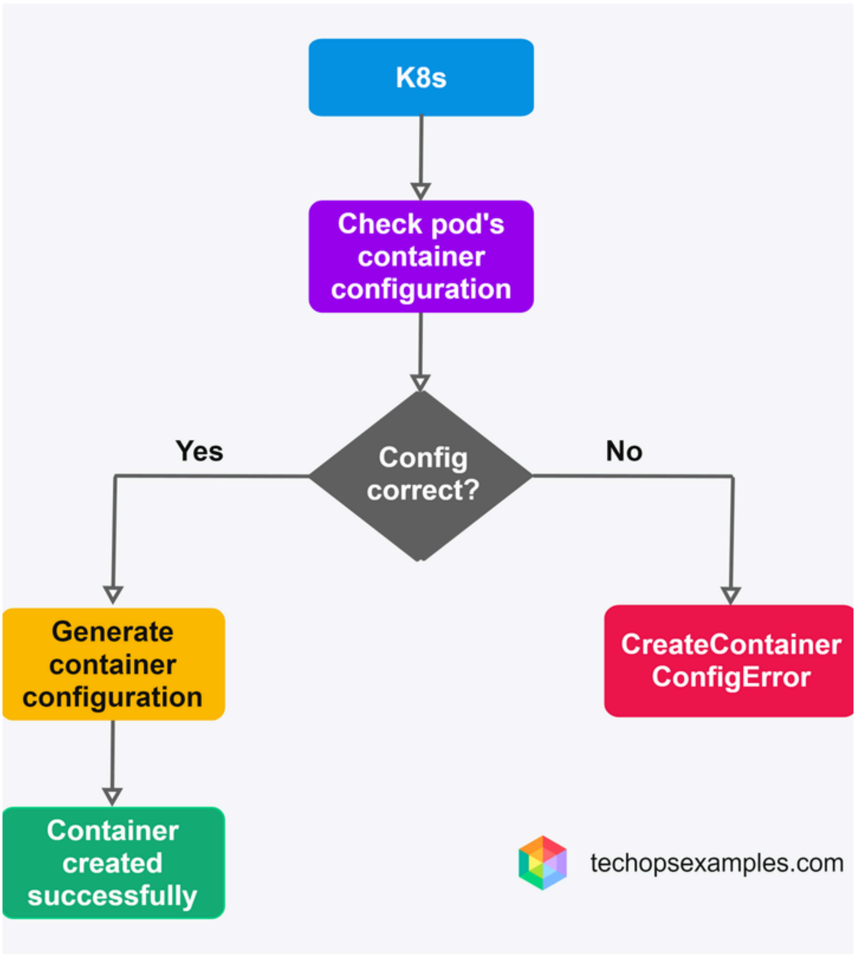
```
$ kubectl create secret docker-registry reg-secret \--
docker-server=private-repo.com \--docker-
username=myuser \--docker-password=mysecurepassword \-
-docker-email=user@techopsexamples.com
```

Link the secret to the pod's manifest to allow access.

```
apiVersion: v1
kind: Pod
metadata:
  name: private-api-pod
  labels:
    app: private-api
spec:
  containers:
  - name: private-api-container
    image: private-repo.com/internal-app:v3.0
  imagePullSecrets:
  - name: reg-secret
```

CreateContainerConfigError

CreateContainerConfigError is an error that occurs when Kubernetes fails to create a container due to an incorrect or incomplete configuration in the Pod's container settings, preventing it from generating the required setup for the container.


K8s
Check pod's container configuration
Config correct?
Yes
No
Generate container configuration
Container created successfully
CreateContainer ConfigError
techopsexamples.com

You can detect the error by running *kubectl get pods*:

```
$ kubectl get pods
NAME       READY   STATUS                       RESTARTS   AGE
teg-pod    0/1     CreateContainerConfigError   0           11m
```

Common Causes for CreateContainerConfigError:

Cause	Description
Missing ConfigMap	The ConfigMap referred to by the Pod doesn't exist or can't be accessed.
Missing secret	The secret mentioned in the Pod's config is either unavailable or inaccessible.
Container name already in use	A container with the same name is already running, leading to a naming conflict.

When starting a container, Kubernetes uses the **generateContainerConfig** method to read configuration data, including commands, ConfigMaps, Secrets, and storage.

If Kubernetes can't find these resources, it triggers a **CreateContainerConfigError**.

How to Troubleshoot CreateContainerError:

1. Inspect Pod Details

Using *kubectl inspect pod*, you can gather detailed insights into the problematic Pod and its containers:

Containers:

```
web-service:
    Container ID:
    Image:           apache
    Image ID:
    Port:            8080/TCP
    Host Port:       8080/TCP
    State:           Waiting
      Reason:        ContainerConfigError
    Ready:           False
    Restart Count:   0
    Limits:
      memory:   256Mi
      cpu:      250m
```

Volumes:

```
app-config:
    Type:      ConfigMap (populated from a ConfigMap)
    Name:      web-config
    Optional:  false
```

2. Retrieve Logs

Utilize *kubectl get-logs* to access logs from the containers within the Pod.

If the Pod has multiple containers, ensure to use *--all-containers*:

```
Error from server (BadRequest): container "web-
service" in pod "teg-pod" is in a waiting state:
ContainerConfigError
```

3. Analyze Recent Pod Events

Run *kubectl events-list* to check all recent events related to the Pod.

You can also get this information at the end of the *kubectl inspect pod* output:

```
LAST SEEN   TYPE      REASON         OBJECT        MESSAGE
45s         Warning   FailedCreate   pod/teg-pod   Container
failed to start: ContainerConfigError
1m          Normal    Scheduled      pod/teg-pod
Successfully assigned teg-pod to node3
```

4. Validate Access and Scope

Verify permissions and namespaces by running kubectl auth can-i and checking the current namespace with *kubectl config view:*

```
$ kubectl auth can-i get pods --
namespace=stagingkubectl config view --minify | grep
namespace
```

Fixing CreateContainerConfigError:

1. Create Missing ConfigMaps and Secrets

If ConfigMaps or Secrets are missing, create them in the correct namespace:

```
$ kubectl create configmap web-config --from-
file=config.yaml

$ kubectl create secret generic db-secret --from-
literal=password=*****
```

2. Check Permissions

Verify that pod has access to the necessary resources:

```
$ kubectl auth can-i get configmap/web-config --
namespace=staging

$ kubectl auth can-i get secret/db-secret --
namespace=staging
```

3. Review Configuration

Check the pod's ConfigMap and Secret references for correctness:

```
$ kubectl describe pod teg-pod --namespace=staging
```

Remember, resolving CreateContainerConfigError is all about verifying your **resources, permissions, and configurations**.

CreateContainerError

CreateContainerConfigError vs. CreateContainerError

While these two errors may sound alike, they happen at different stages in the container lifecycle:

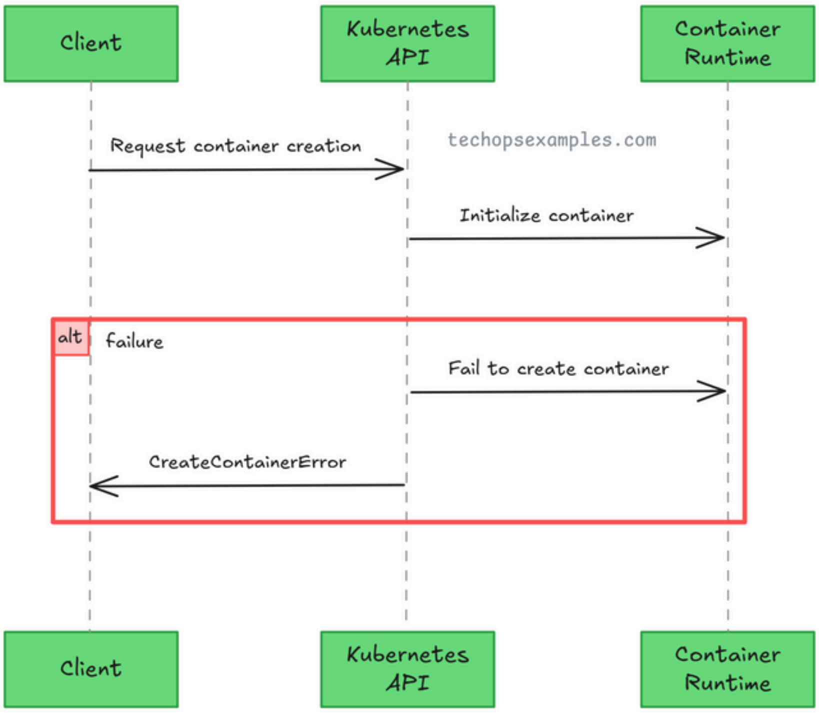
- **CreateContainerConfigError:** This happens when something is wrong with your Pod's configuration. Think of it as a setup issue that stops the container from being created.

- **CreateContainerError:** This occurs later, during the actual creation of the container. The setup might be correct, but the container fails to start for other reasons.

You can detect the error by running the **kubectl get pods** command:

```
NAME                READY   STATUS                  RESTARTS   AGE
techops-examples    0/1     CreateContainerError    0          5m
```

When starting a container, Kubernetes goes through the initialization process, where it pulls the image, allocates resources, and mounts volumes. If any of these steps fail, Kubernetes triggers a **CreateContainerError**.



Common Causes for CreateContainerError:

Cause	Description
Incorrect Image	The image specified in the Pod manifest doesn’t exist, is unavailable, or lacks a command.
Resource Constraints	If the requested resources (CPU or memory) exceed the available capacity on the node.
Volume Mount Issues	The Pod is trying to mount a volume that doesn’t exist or is misconfigured.
Container Runtime Error	An issue with the container runtime (e.g., Docker, containerd) on the node.

How to Troubleshoot CreateContainerError:

1. Inspect Pod Details

Start by inspecting the Pod to get detailed insights into the container creation failure:

```
$ kubectl describe pod techops-examples
```

you can see that the container is in the “**Waiting**” state with the reason listed as **CreateContainerError**

```
Containers:
  techops-examples:
    Container ID:
    Image:          ubuntu:latest
    Image ID:
    State:          Waiting
      Reason:       CreateContainerError
    Ready:          False
    Restart Count:  0
    Resources:
      limits:
        memory: "4Gi"
        cpu: "2000m"
Volumes:
  data-volume:
    Type:       PersistentVolumeClaim (created from a PVC)
    Name:       data-pvc
    Optional:   false
```

2. Retrieve Logs

Use the following command to check the logs of the Pod’s container and find more details on why the container failed to start:

```
$ kubectl logs techops-examples
```

If the container has not yet been created, you may see an error indicating that there are no logs available.

3. Analyze Recent Pod Events

Use the **kubectl get events** command to view recent events related to the Pod and identify any specific reasons for the **CreateContainerError**.

```
$ kubectl get events --field-selector
involvedObject.name=techops-examples
```

Example Output:

```
LAST SEEN   TYPE      REASON                  OBJECT
MESSAGE
30s         Warning   FailedCreate            pod/techops-
examples Failed to create container: CreateContainerError
```

4. Validate Resource Availability

Check if there are enough resources available (CPU and memory) on the node to run the container. Use:

```
$ kubectl top nodes
```

Fixing CreateContainerError:

1. Fix Image Issues

If the image specified is incorrect, ensure the correct image is pulled by providing a valid image name and tag:

```
containers:
  - name: techops-examples
    image: nginx:1.21.3
    command:
      - "/bin/bash"
      - "-c"
      - "echo Application is up"
```

If the error is caused by a missing command, add a valid entrypoint to the image.

2. Adjust Resource Requests

If the container is failing due to insufficient resources, adjust the resource requests and limits in the Pod configuration to fit the available resources on the node:

```
resources:
  requests:
    memory: "2Gi"
    cpu: "1000m"
```

3. Correct Volume Mounts

If the issue is with missing or misconfigured volumes, verify that the correct PersistentVolumeClaim (PVC) is available and referenced properly:

```
volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: data-pvc
```

in this example, ensure that the data-pvc exists in the namespace and that it is correctly configured.

4. Check Container Runtime

If there is a problem with the container runtime, review the kubelet logs on the node where the Pod is scheduled.

Run the following command on the node:

```
sudo journalctl -u kubelet
```

Look for errors related to the container runtime and restart the kubelet or runtime service if needed.

Running into a CreateContainerError can be frustrating, but it usually comes down to checking your image, resources, volume mounts, and runtime environment.

RunContainerError

The **RunContainerError** indicates that the container couldn’t initiate.

When you see this error, it means the application inside hasn’t started because the container itself encountered a failure before it could begin loading the application.

```
$ kubectl get pods

NAME         READY   STATUS             RESTARTS   AGE
techops-app  0/1     RunContainerError  0          6m12s
```

If your pod status shows RunContainerError, it’s often due to:

- Missing or incorrect volume mounts (e.g., ConfigMap or Secret).
- Attempting to write to a read-only volume.
- Invalid commands or missing executables.
- Permissions or security context issues.

How to Fix RunContainerError

1. Check Pod Events for Detailed Errors

Start by describing the Pod to check for specific error events.

Look for any error messages in the "**Message**" field, such as *permission denied, file not found, or invalid command*.

In the below example, the error permission denied indicates a file permission issue with */app/start.sh*.

```
$ kubectl describe pod techops-app

Events:

Type     Reason     Age   From                  Message
----     ------     ----  ----                  -------
Normal   Scheduled  6m    default-scheduler     Successfully
assigned default/techops-app to node-1
Normal   Pulling    6m    kubelet               Pulling image
"techops-image:latest"
Normal   Pulled     6m    kubelet               Successfully
pulled image "techops-image:latest"
Warning  Failed     6m    kubelet               Error: failed
to start container "techops-container":
                                                Error response
from daemon: oci runtime error:

container_linux.go:345: starting container process caused
                                                "exec:
\"/app/start.sh\": permission denied"
Warning  BackOff    5m    (x3 over 6m) kubelet  Back-off
restarting failed container
```

2. Inspect Container Logs

If the container started briefly before failing, check logs for any application-specific error messages.

If you see errors like *Permission denied or Command not found*, it points to issues with permissions or command configurations in the container.

```
$ kubectl logs techops-app -c techops-container
/bin/sh: 1: /app/start.sh: Permission denied
```

3. Verify Volume Mounts

Check that any referenced ConfigMaps, Secrets, or PersistentVolumes are correctly defined and available.

If you receive *Error from server (NotFound)*, the ConfigMap or Secret is missing, causing *RunContainerError*.

```
$ kubectl get configmap techops-config
Error from server (NotFound): configmaps
"techops-config" not found
```

```
$ kubectl get secret techops-secret
Error from server (NotFound): secrets "techops-
secret" not found
```

4. Check Commands and Entrypoints

Verify the command and args fields in your Pod’s specification. If these commands or paths are incorrect, the container won’t start.

Ensure the command path (*/app/start.sh* in this example) exists within the container and has the correct permissions.

```yaml
containers:
  - name: techops-container
    image: techops-image:v1.2.3
    command: ["/app/start.sh"]
    args: ["--env", "production", "--debug", "false"]
```

5. Inspect Security Contexts

Check if the security context is properly configured in pod.yaml, as incorrect user/group settings can lead to permissions errors.

Verify that runAsUser and runAsGroup match the requirements of your container image.

```yaml
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
```

6. Validate Image Permissions

Some images require specific permissions to start. If needed, adjust the Pod's serviceAccount or security settings to meet these requirements.

7. Check File and Directory Permissions

If the error message includes “permission denied,” ensure that the relevant files and directories have the correct permissions.

The file should have executable permissions (e.g., -rwxr-xr-x).

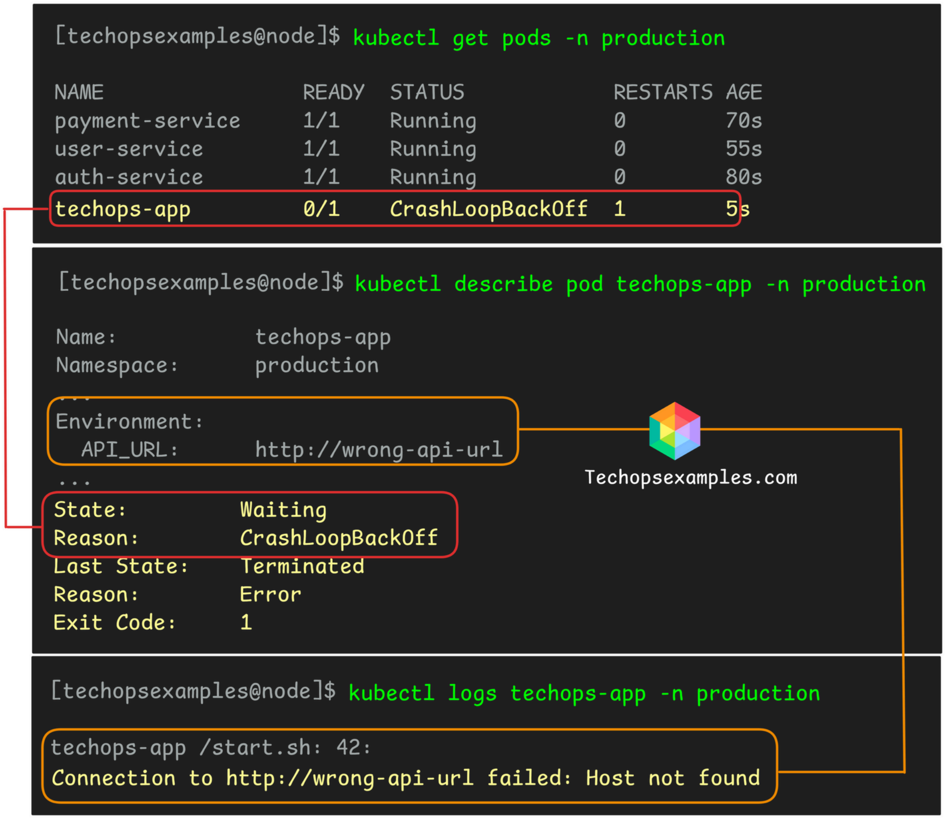
```
$ kubectl exec -it techops-app -- ls -l /app/start.sh

-rwxr-xr-- 1 appuser appgroup 4096 Nov 5 07:00
/app/start.sh
```

In crux, proper file permissions, valid volume mounts, accurate command paths, and appropriate user contexts help to prevent **RunContainerError**.

CrashLoopBackOff

Ever had one of those days when everything seems fine, but there's that one irritating pod that just won't stay up? The most common Kubernetes issue that can really test your patience: the CrashLoopBackOff.


```
[techopsexamples@node]$ kubectl get pods -n production

NAME                 READY   STATUS             RESTARTS  AGE
payment-service      1/1     Running            0         70s
user-service         1/1     Running            0         55s
auth-service         1/1     Running            0         80s
techops-app          0/1     CrashLoopBackOff   1         5s
```

```
[techopsexamples@node]$ kubectl describe pod techops-app -n production

Name:             techops-app
Namespace:        production
...
Environment:
  API_URL:        http://wrong-api-url
...
State:          Waiting
Reason:         CrashLoopBackOff
Last State:     Terminated
Reason:         Error
Exit Code:      1
```

```
[techopsexamples@node]$ kubectl logs techops-app -n production

techops-app /start.sh: 42:
Connection to http://wrong-api-url failed: Host not found
```

What is it?

Simply put, CrashLoopBackOff is a status message indicating that a pod is failing to start repeatedly. It's Kubernetes' way of telling you, "Hey, something's wrong, and I'm giving it a break before I try again."

What factors cause it?

A variety of issues can trigger a CrashLoopBackOff, like:

- **Application bugs**, like unhandled exceptions or critical logic failures, prevent proper startup.
- **Misconfigured volume mounts** result in the application not finding necessary files or directories.
- **Incorrect environment variables** that lead to startup failures, such as specifying a wrong API URL.
- **Dependencies that are unavailable** due to network issues or incorrect DNS settings can cause crashes.
- **Resource constraints**, where insufficient CPU or memory allocation hinders the pod's ability to start.
- **Missing config maps or secrets** can prevent the application from accessing required configuration or credentials.

The problem in the illustrated example lies in the environment variable configuration:

```
Environment:
  API_URL: "http://wrong-api-url"
```

The API_URL is set incorrectly, causing the application to fail at startup.

How to detect it?

You'll notice the **CrashLoopBackOff** pod status:

```
[techopsexamples@node]$ kubectl get pods -n production

NAME              READY   STATUS             RESTARTS   AGE
payment-service   1/1     Running            0          70s
user-service      1/1     Running            0          55s
auth-service      1/1     Running            0          80s
techops-app       0/1     CrashLoopBackOff   1          5s
```

Describe the pod to get more details:

```
[techopsexamples@node]$ kubectl describe pod techops-app -n
production

Name:            techops-app
Namespace:       production
...
Environment:
  API_URL:       http://wrong-api-url
...
State:           Waiting
Reason:          CrashLoopBackOff
Last State:      Terminated
Reason:          Error
Exit Code:       1
Warning          BackOff  2m15s (x100 over 30m)
kubelet          Back-off restarting failed container
```

Check the logs for specific errors:

```
[techopsexamples@node]$ kubectl logs techops-app -n
production
techops-app /start.sh: 42:
Connection to http://wrong-api-url failed: Host not
found
```

now you see, the root cause is ‘incorrect API URL’

How to rollout the fix ?

Set the environment variable to point to the correct API URL:

```
name: API_URLvalue: http://correct-api-url
```

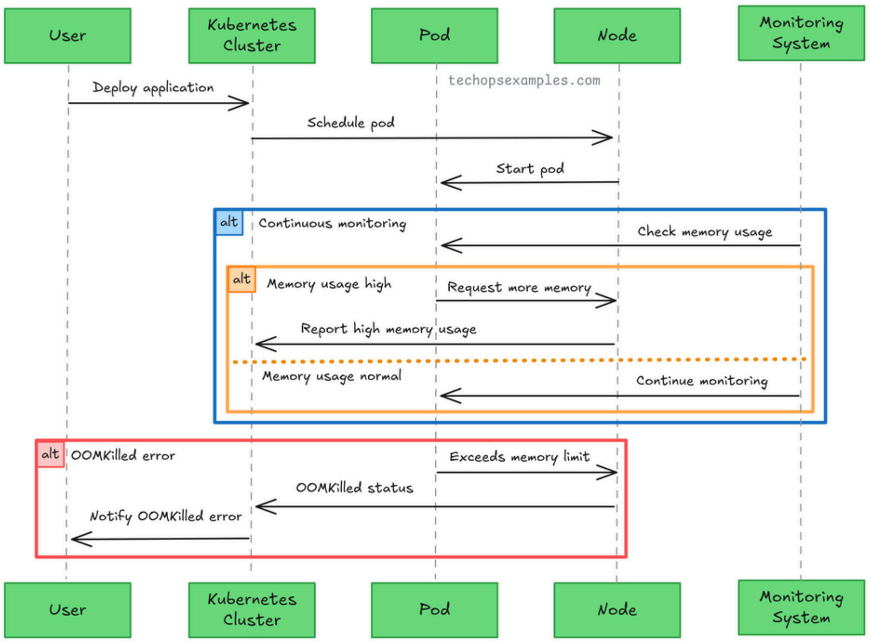
Redeploy the application:

```
[techopsexamples@node]$ kubectl apply -f
deployment.yaml
```

And the pod should be up and running !

OOMKilled

OOMKilled occurs in Kubernetes when a container exceeds its memory limit or tries to access unavailable resources on a node, flagged by exit code 137.



Typical OOMKilled looks like

```
NAME                    READY      STATUS        RESTARTS      AGE
app-pod-1               0/1        OOMKilled     0             4m7s
```

Pods must use less memory than the total available on the node; if they exceed this, Kubernetes will kill some pods to restore balance.

How to Fix OOMKilled Kubernetes Error (Exit Code 137)

1. Identify OOMKilled Event: Run *kubectl get pods* and check if the pod status shows *OOMKilled*.

2. Gather Pod Details: Use *kubectl describe pod [pod-name]* and review the Events section for the OOMKilled reason.

Check the Events section of the describe pod, and look for the following message:

```
State:          Running
       Started:      Mon, 11 Aug 2024 19:15:00 +0200
       Last State:   Terminated
       Reason:       OOMKilled
       Exit Code:    137
       ...
```

3. Analyze Memory Usage: Check memory usage patterns to identify if the limit was exceeded due to a spike or consistent high usage.

4. Adjust Memory Settings: Increase memory limits in pod specs if necessary, or debug and fix any memory leaks in the application.

5. Prevent Overcommitment: Ensure memory requests do not exceed node capacity by adjusting pod resource requests and limits.

Point worth noting:

If a pod is terminate due to a memory issue. it doesn’t necessarily mean it will be removed from the node. If the node’s restart policy is set to ‘**Always**’, the pod will attempt to restart

To check the QoS class of a pod, run this command:

kubectl get pod -o jsonpath='{.status.qosClass}'

To inspect the oom_score of a pod:

1. Run *kubectl exec -it /bin/bash*

2. To see the *oom_score, run cat/proc//oom_score*

3. To see the *oom_score_adj, run cat/proc//oom_score_adj*

The pod with the **lowest oom_score is the first to be terminated** when the node experiences memory exhaustion.

Node Disk Pressure

Imagine you deploy an application, but after a few days, it starts throwing warnings like this:

```
Warning NodePressure [timestamp] kubelet Node
[node-name] status is now: NodeHasDiskPressure
```

Your application slows, pods get evicted, and new ones fail to schedule. This common error in Kubernetes is known as Node Disk Pressure, and if left unchecked, it can severely impact application performance.

What is Kubernetes Node Disk Pressure?

Node Disk Pressure occurs when a node’s filesystem is under strain due to low available disk space or inodes.

Kubernetes automatically detects these low resource conditions and sets a **NodeHasDiskPressure** status.

This status signals that the node has insufficient disk resources for further scheduling, evicting non critical pods to prevent critical system disruptions.

How to Check Kubernetes Node Disk Pressure ?

To verify if your nodes are experiencing Disk Pressure, you can use:

```
kubectl describe node [node-name]
```

Look for any nodes with the condition type DiskPressure and status True.

In the output, focus on the Conditions section. Here’s an example where techops-node2 is experiencing disk pressure:

techops-node2 *Ready worker 14d v1.28.1* ***DiskPressure=True****,MemoryPressure=False,PIDPressure =False,Ready=True*

Additionally, use this command to monitor disk usage:

```
kubectl top nodes
```

This shows overall CPU, memory, and disk usage for each node, helping you pinpoint where Disk Pressure is affecting your nodes.

Why Should You Care About Node Disk Pressure?

Ignoring Disk Pressure can lead to various issues:

Pod Eviction: Kubernetes evicts lower-priority pods to free up disk space, which can cause disruptions in non critical workloads.

Scheduling Failures: New workloads may not deploy if nodes are in a Disk Pressure state.

Performance Degradation: Insufficient disk space impacts node performance and can lead to application latency.

How to Fix Kubernetes Node Disk Pressure ?

Here are some strategies to address Disk Pressure:

Clean Up Disk Space: Clear out unused images and containers, which can take up significant space.

Increase Node Disk Size: If your nodes are in a cloud environment, consider resizing disks. In AWS, for instance - Increase the EBS volume size.

Move Logs and Data to Separate Disks: If your node frequently generates large logs, consider mounting a separate disk for log storage to keep system space free.

Implement Resource Quotas and Limits:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-quota
  namespace: [namespace]
spec:
  hard:
    requests.storage: 10Gi
```

Monitor with Alerts:

Use Prometheus or another monitoring tool to set up alerts when disk usage exceeds a threshold. This proactive approach helps you intervene before Disk Pressure arises.

Node Not Ready

It is very familiar to see a mix of node statuses in a Kubernetes cluster, especially when troubleshooting. Sometimes, nodes might be marked as NotReady due to various issues.

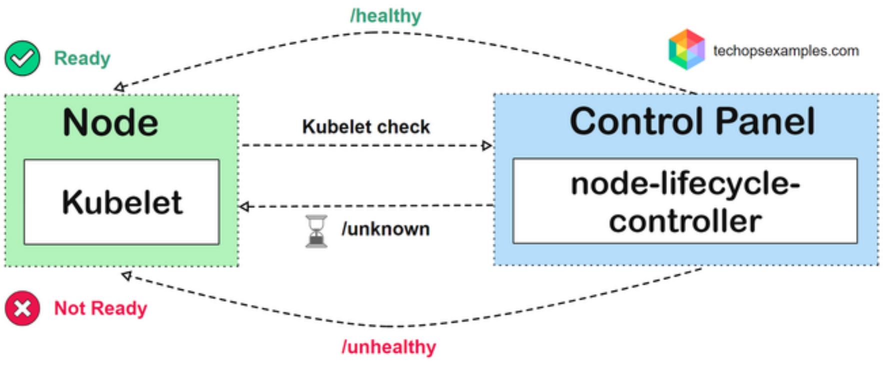
Typically it looks like:

```
techops_examples@master:~$ kubectl get nodes
NAME                  STATUS      ROLES     AGE    VERSION
master                Ready       master    51m    v1.31.0
node-worker-1         NotReady    worker    49m    v1.31.0
node-worker-2         Ready       worker    47m    v1.31.0
```

Behind the scenes:

The kubelet on each node is responsible for reporting the node's status to the control plane, specifically to the **node-lifecycle-controller**. The control plane then assesses this data (or the absence of it) to determine the node’s state.

This is what happens in the background:



The node’s kubelet sends information about various checks it performs, including:

1. Whether the network for the container runtime is functional.
2. If the CSI (Container Storage Interface) provider on the node is fully initialized.
3. The completeness of the container runtime status checks.
4. The operational state of the container runtime itself.
5. The functionality of the pod lifecycle event generator.

6. Whether the node is in the process of shutting down.
7. The availability of sufficient CPU, memory, or pod capacity on the node.

This information is then relayed to the node-lifecycle-controller, which uses it to assign the node one of the following statuses:

- **True**: All checks have passed, indicating the node is operational and healthy.

- **False**: One or more checks have failed, showing the node has issues and isn't functioning correctly.

- **Unknown**: The kubelet hasn't communicated with the control plane within the expected timeframe, leaving the node's status unclear.

When the status is marked as **Unknown**, it usually indicates that the node has lost contact with the control plane, possibly due to network problems, kubelet crashes, or other communication failures.

Diagnosis:

1. Node Status Check:

Run → **Kubectl get nodes** and watch out for the status '**NotReady**'

```
techops_examples@master:~$ kubectl get nodes
NAME                  STATUS      ROLES     AGE    VERSION
master                Ready       master    51m    v1.31.0
node-worker-1         NotReady    worker    49m    v1.31.0
node-worker-2         Ready       worker    47m    v1.31.0
```

2. Node Details and Conditions Check:

To dive deeper into why a node might be **NotReady**, use the kubectl describe command to get detailed information on the node's condition:

MemoryPressure: Node is low on memory.

DiskPressure: Node is running out of disk space.

PIDPressure: Node has too many processes running.

3. Network Misconfiguration Check:

Run → **ping <node-IP>** to check connectivity between the nodes.

If there's packet loss, it indicates a possible network issue that might be causing the node's NotReady status.

```
techops_examples@master:~$ ping 10.0.0.67
PING 10.0.0.67 (10.0.0.67) 56(84) bytes of data.
--- 10.0.0.67 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss,
time 3054ms
```

4. Kubelet Issue Check:

Run → **systemctl status kubelet** on the node to verify if the kubelet service is running properly.

If the kubelet is down, it may be the reason for the node's NotReady status.

```
techops_examples@node-worker-1:~$ systemctl status
kubelet

kubelet.service - Kubernetes Kubelet
   Loaded: loaded
(/etc/systemd/system/kubelet.service; enabled; vendor
preset: enabled)
   Active: active (running) since Tue 2024-08-28
09:25:10 UTC; 1h 29min ago
 Main PID: 2345 (kubelet)
    Tasks: 13 (limit: 4915)
   Memory: 150.1M
      CPU: 8min 27.345s
   CGroup: /system.slice/kubelet.service
           └─2345 /usr/bin/kubelet
```

5. Kube-proxy Issue Check:

Run → **kubectl get pods -n kube-system -o wide | grep kube-proxy** to check the status of the kube-proxy pods on the node.

If the **kube-proxy** pod is in a crash loop or not running, it could cause network issues leading to the **NotReady** status.

```
techops_examples@master:~$ kubectl get pods -n
kube-system -o wide | grep kube-proxy

kube-proxy-5b7c8dfd9f-lk1bp   1/1   Running   0
1h   10.0.0.67   node-worker-1
```

How To Fix:

1. Resolve Lack of Resources:

- **Increase Resources:** Scale up the node or optimize pod resource requests and limits.
- **Monitor & Clean:** Use top or htop to monitor usage, stop non Kubernetes processes, and check for hardware issues.

2. Resolve Kubelet Issues:

Check Status: Run systemctl status kubelet.

- **active (running):** Kubelet is fine; the issue might be elsewhere.
- **active (exited):** Restart with sudo systemctl restart kubelet.
- **inactive (dead):** Check logs with sudo cat /var/log/kubelet.log to diagnose.

3. Resolve Kube-proxy Issues:

- **Check Logs:** Use kubectl logs <kube-proxy-pod-name> -n kube-system to review logs.
- **DaemonSet:** Ensure the kube-proxy DaemonSet is configured correctly. If needed, delete the kube-proxy pod to force a restart.

4. Checking Connectivity:

- **Network Setup:** Verify network configuration, ensure necessary ports are open.
- **Test Connections:** Use ping <node-IP> and traceroute <node-IP> to check network connectivity.

7. POD Troubleshooting Tactics

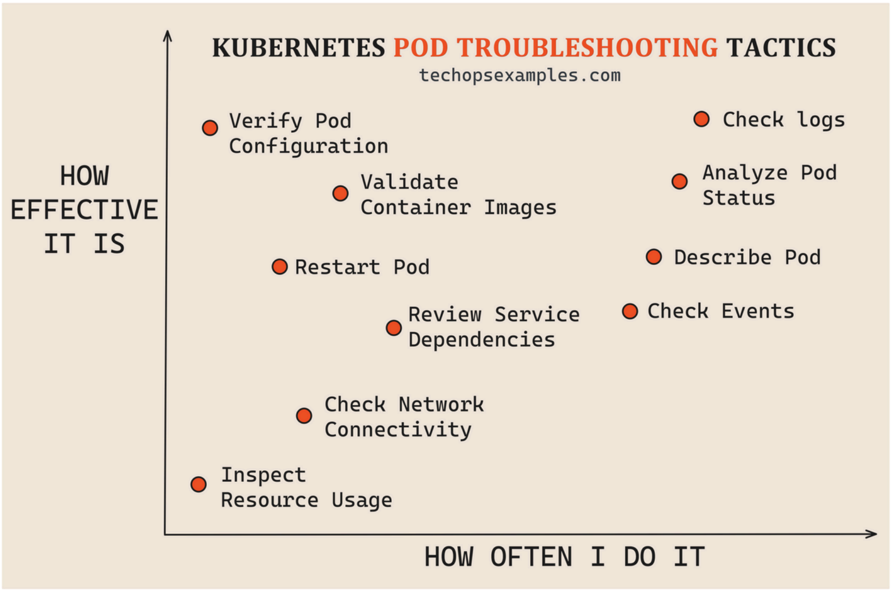
There's a joke in the industry:

Debugged failed pods for 8 hours - No luck.

A random restart the next morning - all set!

If you've been there, you know the frustration.

But instead of hoping for a miraculous restart, here's a structured way to troubleshoot Kubernetes pods effectively.



1. Check Logs

```
kubectl logs <pod_name>
```

If your pod has multiple containers, specify one:

```
kubectl logs <pod_name> -c <container_name>
```

2. Analyze Pod Status

```
kubectl get pod <pod_name>
```

Look at the STATUS column.

If it shows CrashLoopBackOff, ImagePullBackOff, or ErrImagePull, you have clear hints on what to check next.

3. Describe Pod

```
kubectl describe pod <pod_name>
```

Look for warning events, scheduling failures, and container state details.

4. Verify Pod Configuration

A misconfigured pod can cause all sorts of issues. Review its YAML configuration.

```
kubectl get pod <pod_name> -o yaml
```

Check environment variables, resource limits, image versions, and volumes.

5. Check Events

Kubernetes events provide historical context on failures.

```
kubectl get events --sort-by=.metadata.creationTimestamp
```

Pay attention to events like FailedScheduling, ImagePullBackOff, or OOMKilled

6. Validate Container Images

Ensure your container images are correct and available:

Check if the image tag exists.

```
kubectl get pod <pod_name> -o
jsonpath='{.spec.containers[*].image}'
```

Try pulling the image manually.

```
docker pull <image_name>
```

7. Restart Pod

Sometimes, instead of deleting the pod, restarting the deployment helps.

```
kubectl rollout restart deployment/<deployment_name>
```

8. Review Service Dependencies

Pods may fail if dependent services are unavailable. Check the relevant services.

```
kubectl get svc
```

Ensure services are resolving correctly.

```
nslookup <service_name>
```

9. Check Network Connectivity

If your pod can’t communicate with another service, test connectivity.

```
kubectl exec -it <pod_name> -- sh
ping <target_host>
curl <target_url>
```

10. Inspect Resource Usage

If your pod is OOMKilled or throttled, check resource usage.

```
kubectl top pod <pod_name>
```

Compare with defined limits.

Following this structured approach, you save time, avoid frustration, and debug with confidence!

8. Must Know K8s Troubleshooting Commands

Command	Description
kubectl get pods	Lists all pods in the current namespace with their statuses.
kubectl describe pod <pod-name>	Provides detailed information about a specific pod, including events.
kubectl logs <pod-name>	Fetches logs from a running pod's main container.
kubectl logs -f <pod-name>	Streams real-time logs from a pod's main container.
kubectl logs <pod-name> -c <container-name>	Retrieves logs from a specific container inside a pod.

8. Must Know K8s Troubleshooting Commands (continued..)

Command	Description
kubectl exec -it <pod-name> -- /bin/sh	Opens a shell inside a running container for debugging.
kubectl get events --sort-by=.metadata.creationTimestamp	Displays Kubernetes events sorted by time for diagnosing issues.
kubectl get nodes -o wide	Lists nodes with detailed information such as IPs and roles.
kubectl get all -n <namespace>	Retrieves all resources in a specific namespace.
kubectl describe node <node-name>	Shows detailed information about a node, including conditions and taints.
kubectl get deployments -o wide	Lists deployments with additional details like image versions.
kubectl get svc -o wide	Lists all services with their cluster IPs and external endpoints.

8. Must Know K8s Troubleshooting Commands (continued..)

Command	Description
kubectl top pod	Displays resource usage (CPU, memory) of pods if metrics-server is enabled.
kubectl top node	Shows resource usage of nodes.
kubectl get pvc	Lists persistent volume claims and their statuses.
kubectl get pv	Displays persistent volumes and their bindings.
kubectl describe ingress <ingress-name>	Provides detailed information about an ingress resource for debugging.
kubectl debug node/<node-name> --image=img_v1.0	Creates a debugging pod on a node using img_v1.0.
kubectl cordon <node-name>	Creates a debugging pod on a node using BusyBox.

Hope this guide was helpful - build on it, stay patient through challenges, and everything will fall into place.

— Govardhana Miriyala Kannaiah

Give a shout out on how this guide helped you on my LinkedIn and Twitter (X), I'm listening.

LinkedIn link

Twitter (X) link