Of course. Here is the complete "Complete Kubernetes Troubleshooting Guide" consolidated into a single response for easy copying.

The Complete Kubernetes Troubleshooting Guide

By Shakthi Vinayak

Table of Contents

Foreword: The Zen of Kubernetes Troubleshooting

Kubernetes is a beast. A powerful, complex, and often unforgiving one. As a DevOps engineer who has spent countless nights in the trenches, I can tell you that every single person who works with Kubernetes will, at some point, face a cryptic error message that sends them down a rabbit hole of `kubectl` commands and GitHub issues.

This book is born from those late nights. It's a compilation of real-world scenarios, hard-won lessons, and the systematic approaches I've developed over years of maintaining and healing Kubernetes clusters. The goal is not just to give you a list of commands to copy and paste, but to instill a troubleshooting mindset. It's about learning to ask the right questions, to understand the flow of information within the cluster, and to methodically peel back the layers until the root cause is exposed.

Troubleshooting Kubernetes is like being a detective. Your cluster is the crime scene, the logs are your witnesses (some more reliable than others), and the metrics are your forensic evidence. This guide will be your partner in that investigation. We will start with the basics, your detective's toolkit, and then move through every level of the Kubernetes stack, from the humble Pod to the mighty control plane.

So, grab a coffee, open your terminal, and let's begin the journey to becoming a Kubernetes troubleshooting master.

Part 2: Application-Level Troubleshooting

Chapter 3: Pod-Level Issues: The Building Blocks of Failure

If the Pod is the atom of Kubernetes, then its failures are the elementary particles of cluster chaos. Understanding why a Pod fails to run is the most fundamental skill in a troubleshooter's arsenal. Most application-level problems start here.

Scenario 1: The Dreaded `CrashLoopBackOff`

This means your application is starting, crashing, and Kubernetes is restarting it.

1. **Probe:** `kubectl describe pod <pod-name>` to see the exit code and reason.
2. **Probe:** `kubectl logs <pod-name> --previous` to see the application error that caused the crash.
3. **Fix:** The fix depends on the logs. It's usually an application bug, a missing ConfigMap/Secret (leading to a missing environment variable), or a misconfigured command.

Scenario 2: My Pod is `Pending` Forever

The pod has been accepted by the cluster but cannot be scheduled onto a node.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Events` section is key.
2. **Common Causes & Fixes:**
    - **`Insufficient cpu/memory`:** The cluster nodes don't have enough available resources to meet the pod's `requests`. **Fix:** Lower the pod's requests, add more nodes, or remove other pods.
    - **`didn't match node selector` / `node(s) had taints that the pod didn't tolerate`:** The pod's scheduling constraints (selectors, affinity, tolerations) prevent it from being placed on any available node. **Fix:** Adjust the pod's scheduling constraints or the node's labels/taints.

- **`PersistentVolumeClaim is not bound`:** The pod is waiting for a storage volume that is not yet available. **Fix:** Troubleshoot the PVC (See Chapter 6).

Scenario 3: `ImagePullBackOff` and `ErrImagePull`

Kubelet cannot pull the container image.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Events` section will show the specific pull error.
2. **Common Causes & Fixes:**
    - **`not found`:** The image name or tag is spelled incorrectly, or the image does not exist in the registry. **Fix:** Correct the image name in your Deployment YAML.
    - **`unauthorized: authentication required`:** The cluster doesn't have credentials to access a private registry. **Fix:** Create a `docker-registry` secret and reference it using `imagePullSecrets` in your pod spec.
    - **Timeout / `no such host`:** The node cannot reach the image registry due to a network issue (DNS, firewall, proxy). **Fix:** SSH to the node and debug its network connectivity to the registry.

Scenario 4: Readiness and Liveness Probes Fail

The pod is running, but Kubernetes thinks it's unhealthy.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Events` will show `Unhealthy` messages for Readiness or Liveness probes.
2. **Probe:** `kubectl exec -it <pod-name> -- /bin/sh` and manually run the probe command (e.g., `curl http://localhost:8080/healthz`) to see why it's failing.
3. **Common Causes & Fixes:**
    - **`initialDelaySeconds` is too short:** The application hasn't finished starting before the probe begins. **Fix:** Increase the delay.
    - **Wrong port, path, or command:** The probe is checking the wrong thing. **Fix:** Correct the probe definition.
    - **The application is truly unhealthy:** The probe is working correctly and has detected a problem. **Fix:** Debug the application code.

Scenario 5: `OOMKilled`: The Memory Monster

The container used more memory than its `limit` and was terminated by the kernel.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Last State` section will show `Reason: OOMKilled`. The exit code is often `137`.
2. **Probe:** Use Prometheus/Grafana to view the historical memory usage of the pod. You will see it spike up and hit its limit right before termination.
3. **Fix:**
    - Increase the `resources.limits.memory` in your Deployment YAML.
    - Optimize the application code to use less memory or fix a memory leak.

Scenario 6: Init Container Failures

An init container fails, preventing the main application containers from starting.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Init Containers` section will show the failing container and its exit code.
2. **Probe:** `kubectl logs <pod-name> -c <init-container-name>` to see the logs from the failed init container.
3. **Fix:** Debug the issue based on the logs. It's often a script error, a permissions problem, or an inability to connect to another service (like a database) during the setup phase.

Chapter 4: Deployment & ReplicaSet Issues

Scenario 1: Stuck Rollouts

You run `kubectl rollout status` and it hangs, waiting for new replicas to become available.

1. **Probe:** Find the new pods created by the new ReplicaSet: `kubectl get pods --selector=app=<your-app>`.
2. **Analysis:** The new pods are failing. They are in `CrashLoopBackOff`, `ImagePullBackOff`, or are not becoming `Ready`.
3. **Fix:** This is a Pod-level issue. Use the techniques from **Chapter 3** to diagnose and fix the new pods. To restore service quickly, run `kubectl rollout undo deployment <deployment-name>`.

Scenario 2: The Mismatched Selector

You create a Deployment, but no pods are created.

1. **Probe:** `kubectl get deployment <name> -o yaml`.
2. **Analysis:** Compare `spec.selector.matchLabels` with `spec.template.metadata.labels`. They **must** be identical. If they are not, the ReplicaSet created by the Deployment will not "adopt" the pods it's supposed to create.
3. **Fix:** You cannot change a selector on an existing Deployment. Delete the broken Deployment (`kubectl delete deployment <name>`), correct the YAML, and apply it again.

Scenario 3: ConfigMap and Secret Updates Not Propagating

You updated a ConfigMap or Secret, but the application is still using the old values.

1. **Analysis:** Pods consuming these as **environment variables** only get the values at startup. Pods consuming them as **mounted volumes** get updated eventually, but the application must know how to reload the file from disk.
2. **Fix (Best Practice):** Trigger a graceful rolling restart of your application. This forces the creation of new pods, which will pick up the new configuration.

```
kubectl rollout restart deployment <deployment-name>
```

3. **Fix (GitOps Method):** Add a checksum annotation of the ConfigMap/Secret to your Deployment's pod template. When the config changes, the checksum changes, which automatically triggers a rolling update.

Scenario 4: Horizontal Pod Autoscaler (HPA) Not Scaling

You've configured an HPA, but it's not adding or removing pods when the load changes.

1. **Probe:** `kubectl describe hpa <hpa-name>`. The `Events` and `Metrics` sections are critical.
2. **Common Causes & Fixes:**
    - **Metrics show `<unknown> / target`:** The Kubernetes Metrics Server is not installed or is unhealthy. **Fix:** Install or debug the `metrics-server` in the `kube-system` namespace. You can verify this with `kubectl top pods`.
    - **Event: `missing request for cpu`:** The HPA calculates utilization as a percentage of the pod's `requests`. If you haven't set `resources.requests.cpu` in your container spec, the HPA cannot function. **Fix:** Add CPU requests to your Deployment's containers.
    - **No errors, but no scaling:** The actual metric (e.g., 35% CPU) has not crossed the target threshold you set (e.g., 60% CPU). The HPA is working correctly; the load is just not high enough.

Chapter 5: Networking Nightmares

Scenario 1: Pod-to-Pod Communication Failures

A `frontend` pod cannot connect to a `backend` pod's IP address.

1. **Probe:** `exec` into the source pod (`kubectl exec -it <source-pod> -- /bin/sh`) and try to connect to the destination pod's IP and port (`curl -v <dest-pod-ip>:<port>`).
2. **Suspect #1: Network Policies.** This is the most common cause of silent failures.
    - **Probe:** `kubectl get networkpolicy -n <namespace>`.
    - **Analysis:** Check if any policy with a `podSelector` matching your destination pod has an `ingress` rule that allows traffic from your source pod's labels. If any policy applies, it's deny-by-default.
    - **Fix:** Adjust the Network Policy to allow the connection.
3. **Suspect #2: CNI Plugin.** If no policies are blocking traffic, the CNI daemonset itself may be unhealthy.
    - **Probe:** Check the logs of the CNI pods (e.g., Calico, Flannel) in the `kube-system` namespace on both the source and destination nodes.

Scenario 2: "Could not resolve host": DNS Debugging

Your pod cannot resolve service names like `my-db-service`.

1. **Probe:** `exec` into the pod. Run `nslookup my-db-service`. Then try `nslookup kubernetes.default`. Then `nslookup www.google.com`. This helps isolate the problem.
2. **Probe:** Check the logs of the CoreDNS pods: `kubectl logs -l k8s-app=kube-dns -n kube-system`.
3. **Common Causes & Fixes:**
    - **CoreDNS pods are unhealthy:** Restart them (`kubectl rollout restart deployment coredns -n kube-system`).

- **Upstream DNS failure:** CoreDNS can't reach the external DNS servers. Debug the node's DNS.
- **Network Policy:** A network policy is blocking your pod from reaching the CoreDNS pods on port 53.

Scenario 3: Ingress Errors (502/503)

External traffic is failing to reach your service.

1. **Trace the Path:** Ingress Controller -> Service -> Endpoints -> Pod. A failure at any point breaks the chain.
2. **Probe:** Check the logs of your Ingress Controller pods. This is the most important step. It will often tell you exactly what's wrong.
    - `"could not find any endpoints for service..."` -> **This is the smoking gun.**
3. **Probe:** If you see that message, check the Service's Endpoints: `kubectl describe service <service-name>`.
4. **Analysis:** If the `Endpoints` list is `<none>`, it means the Service's `selector` does not match the labels of any **Ready** pods.
5. **Fix:**
    - Correct the `selector` on the Service or the `labels` on the pods.
    - Ensure the pods are `Ready` (passing their readiness probes).
    - Verify the `targetPort` on the Service matches the `containerPort` in the Pod.

Chapter 6: Storage Woes

Scenario 1: My PersistentVolumeClaim is `Pending`

A PVC is stuck and never gets `Bound`.

1. **Probe:** `kubectl describe pvc <pvc-name>`. The `Events` will tell you why.
2. **Common Causes & Fixes:**
    - **`no volume plugin matched` / invalid `storageClassName`:** The StorageClass name in your PVC doesn't exist. **Fix:** Use `kubectl get sc` to see available classes and correct the name.
    - **`Provisioning failed` with a cloud error:** The CSI driver is failing to create the disk in your cloud account. **Fix:** Check the logs of the CSI provisioner pod (in `kube-system`) for detailed errors from the cloud provider (e.g., permissions, API limits).
    - **No default StorageClass:** You didn't specify a StorageClass, and no default is configured in the cluster. **Fix:** Specify a StorageClass or create a static PV for it to bind to.

Scenario 2: Pod stuck with `FailedMount` Error

The PVC is `Bound`, but the pod can't mount it and is stuck in `ContainerCreating`.

1. **Probe:** `kubectl describe pod <pod-name>`. The `Events` will show a `FailedMount` warning with a detailed message.
2. **Common Causes & Fixes:**
    - **`Multi-Attach error`:** A `ReadWriteOnce` (RWO) volume is still attached to a dead node and cannot be attached to a new one. **Fix:** Manually detach the disk from the old node in your cloud provider's

console.

- **`PermissionDenied`:** The node's IAM role doesn't have permission to attach disks. **Fix:** Add the required permissions to the node's instance profile.
- **Timeout:** The CSI node plugin (a DaemonSet) is not running on the node where the pod is scheduled. **Fix:** Debug the CSI node daemonset.

Chapter 7: Security & RBAC

Scenario 1: "Forbidden: user cannot list resource..."

A human user gets a `Forbidden` error from `kubectl`.

1. **Probe (as admin):** Use the `can-i` command to impersonate the user and verify the lack of permission.

```
kubectl auth can-i list pods --as jane.doe@example.com -n team-alpha
```

2. **Probe:** Find the `RoleBindings` or `ClusterRoleBindings` that apply to the user.
3. **Analysis:** Inspect the `Role` or `ClusterRole` that the binding points to. You will find that the required permission (`list` on `pods`) is missing.
4. **Fix:** Edit the `Role`/`ClusterRole` to add the missing permission.

Scenario 2: Debugging Service Account Permissions

A pod trying to talk to the Kubernetes API gets a `Forbidden` error.

1. **Probe:** Identify the `ServiceAccount` the pod is using (`kubectl describe pod <name>`).
2. **Probe (as admin):** Use `can-i` to impersonate the ServiceAccount.

```
kubectl auth can-i get secrets --as system:serviceaccount:my-ns:my-sa -n my-ns
```

3. **Fix:** Create the necessary `Role`/`ClusterRole` and `RoleBinding`/`ClusterRoleBinding` to grant the ServiceAccount the permissions it needs.

Scenario 3: Pod Security Admission / Policy Issues

`kubectl apply` fails with a `Forbidden` error, saying the pod `violates PodSecurity`.

1. **Probe:** Check the labels on the namespace: `kubectl get ns <name> --show-labels`. Look for the `pod-security.kubernetes.io/enforce` label (e.g., `restricted`).
2. **Analysis:** The pod spec violates the enforced security policy. The error message will often tell you exactly what's wrong (e.g., `runAsNonRoot (container must set securityContext.runAsNonRoot=true)`).
3. **Fix:** Add a compliant `securityContext` to your pod and container spec in your Deployment YAML. For `restricted`, this usually means setting `runAsNonRoot: true`, `allowPrivilegeEscalation: false`, dropping all `capabilities`, and setting a `seccompProfile`.

Chapter 8: Node-Level Issues

Scenario 1: The `NotReady` Node

A node's status is `NotReady`.

1. **Probe (control plane):** `kubectl describe node <node-name>`. The `Conditions` will show `NodeStatusUnknown` because the Kubelet has stopped reporting in.
2. **Probe (on the node):** SSH into the affected node.
3. **Check Services (on the node):**
    - `sudo systemctl status containerd` (or `docker`)
    - `sudo systemctl status kubelet`
4. **Fix:**
    - If a service is dead, try restarting it (`sudo systemctl restart ...`).
    - Check its logs (`sudo journalctl -u <service-name> -f`) for the root cause.
    - Common causes are a crashed container runtime, a misconfigured Kubelet, resource exhaustion (100% CPU/disk) on the node, or a network partition between the node and the control plane.

Scenario 2: Disk, Memory, and PID Pressure

The node is `Ready` but has a condition like `MemoryPressure` or `DiskPressure`, causing pod evictions.

1. **Probe:** SSH into the node.
2. **For `DiskPressure`:** Use `df -h` to find which filesystem is full (`/` or `/var/lib/containerd`). **Fix:** Clean up unused container images (`crictl rmi --prune`) and large log files.
3. **For `MemoryPressure`:** Use `top` to find the processes consuming the most memory. **Fix:** This is usually a pod-level issue. Find the pod and debug its memory usage (see Chapter 3).

Chapter 9: Control Plane Catastrophes (Self-Managed)

Scenario 1: API Server Unavailability

`kubectl` commands fail to connect.

1. **Probe:** SSH to a master node. Check the status and logs of the `kube-apiserver` pod: `kubectl get pods -n kube-system` and `kubectl logs <apiserver-pod> -n kube-system`.
2. **Common Causes & Fixes:**
    - **Cannot connect to `etcd`:** The API server's primary dependency is down. Troubleshoot `etcd`.
    - **Expired certificates:** The API server's TLS certificates have expired. **Fix:** Use `kubeadm certs renew all` to regenerate them.
    - **Resource exhaustion on the master node.**

Scenario 2: `etcd` Cluster Health and Quorum Loss

The cluster is down or read-only.

1. **Probe:** SSH to a master node. Execute an `etcdctl endpoint health --cluster` command from within an `etcd` pod to check member status.
2. **Fix (Disaster Recovery):** Losing quorum (e.g., losing 2 of 3 masters) is a critical failure. **Do not improvise.** You must follow the official Kubernetes documentation to restore the `etcd` cluster from a snapshot. The real fix is prevention: run a 3- or 5-node `etcd` cluster across different availability zones and have automated backups.

Chapter 10: Performance Tuning and Bottlenecks

Scenario 1: CPU Throttling

Your application is slow and has latency spikes, but isn't crashing.

1. **Probe:** Use Prometheus to query the `container_cpu_cfs_throttled_seconds_total` metric. If the rate is greater than 0, your container is being throttled.
2. **Analysis:** This means the container is trying to use more CPU than its configured `limit`.
3. **Fix:**
    - Increase the `resources.limits.cpu`.
    - Optimize the application code to be more efficient.
    - Consider removing the CPU limit (but keeping the request) for latency-sensitive services.

Scenario 2: API Server and Scheduling Delays

`kubectl` is slow, and pods take a long time to get scheduled.

1. **Probe:** Use Prometheus to check key control plane metrics:
    - `apiserver_request_duration_seconds_bucket`: Shows API server latency.
    - `etcd_request_duration_seconds_bucket`: Shows latency of the underlying database.
    - `etcd_disk_wal_fsync_duration_seconds`: `etcd`'s disk performance. This is critical.
2. **Analysis:** If `etcd` disk latency is high (>10ms), your master node disks are too slow. This is the most common cause of control plane performance issues.
3. **Fix:** You **must** run master nodes on high-performance SSDs.

Chapter 11: Kubernetes Component Failures

Scenario 1: When CoreDNS Fails

All in-cluster DNS resolution fails.

1. **Probe:** Check the status and logs of the `coredns` pods in the `kube-system` namespace.
2. **Fix:** Look for errors in the logs. Common causes are a bad configuration in the `coredns` ConfigMap, inability to reach upstream DNS servers from the nodes, or insufficient CPU/memory resources for the CoreDNS pods themselves.

Scenario 2: Troubleshooting the CNI Plugin

New pods are stuck in `ContainerCreating` with CNI errors.

1. **Probe:** Check the health of the CNI DaemonSet (e.g., `calico-node`). Check the logs of the CNI pod on the node where the new pod is failing to start.
2. **Common Causes & Fixes:**
    - **IP Address Exhaustion:** The node has run out of available IP addresses from its assigned Pod CIDR block. **Fix:** This is a design issue requiring you to re-architect with larger CIDRs.
    - **Connectivity to `etcd` or datastore:** Some CNIs need to talk to a central store to coordinate. **Fix:** Debug the network path between the nodes and that datastore.

Conclusion: The Troubleshooter's Mindset

1. **Start Broad, Then Go Deep:** Work from Deployment -> Pod -> Node.
2. **`describe` is Your Best Friend:** Always start with `kubectl describe`.
3. **Logs Tell the Story:** When `describe` tells you *what* failed, `kubectl logs` tells you *why*.
4. **Trust but Verify with Metrics:** Use Prometheus to find performance bottlenecks and "unknown unknowns."
5. **Understand the Data Flow:** Trace the path of a request through the system to build a mental model of how components interact.