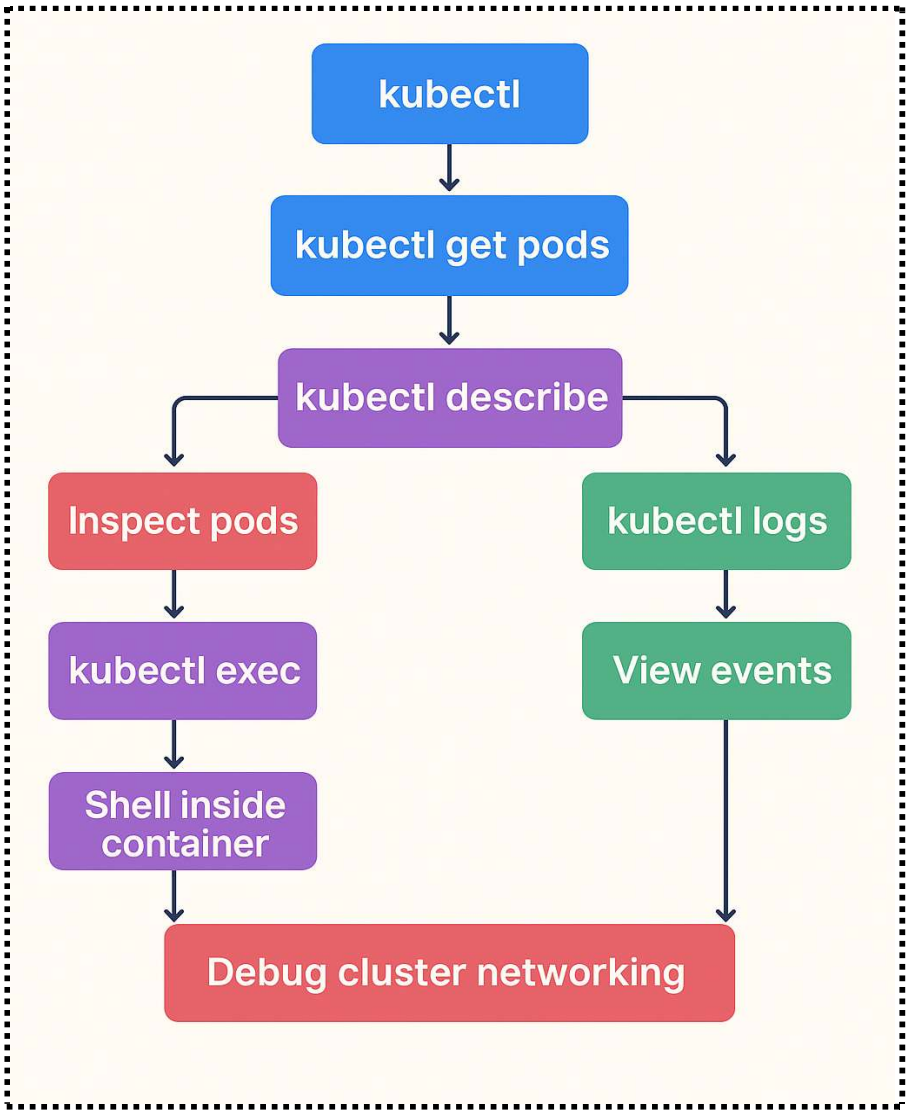
Debugging and Logging Kubernetes Applications

This project focuses on understanding how to **debug, troubleshoot, and analyze** applications running inside a Kubernetes cluster.



Since Kubernetes apps run in multiple Pods and containers, troubleshooting requires checking:

- Pod health
- Logs and events
- Container status
- Service connectivity
- DNS issues
- Network Policies

2. Project Objective

The primary goals of this project were to:

- Debug Kubernetes workloads using kubectl
- Check pod lifecycle, events, and health
- View and analyze container logs
- Perform in-cluster debugging using exec
- Troubleshoot service and connectivity issues
- Verify DNS functionality inside the cluster
- Review NetworkPolicies for traffic restrictions

---

3. Prerequisites

Before starting the project, the following prerequisites were required:

- A running Kubernetes cluster
- kubectl installed and configured
- A sample app deployed for testing
- Basic knowledge of YAML, Pods, Services, and Deployments

---

4. Implementation Steps

Step 1: Create a Sample Deployment

A deployment was created to run a simple guestbook application.

- ***kubectl create deployment guestbook --image=ibmcom/guestbook:v1***

```
controlplane:~$ kubectl create deployment guestbook --image=ibmcom/guestbook:v1
deployment.apps/guestbook created
```

This deploys the guestbook container inside a Kubernetes Pod, allowing further debugging actions.

Step 2: Inspecting and Debugging Pods

When a deployment fails or a pod enters an error state, the following inspection methods were used:

a) Retrieve Basic Pod Information

- *kubectl get pods*

This outputs pod name, status, restart count, and age.

b) Describe a Pod for Detailed Diagnostics

- *kubectl describe pod <pod-name>*

```
controlplane:~$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
guestbook-86cf798756-nm4p9   1/1     Running   0          18s
controlplane:~$ kubectl describe pod guestbook-86cf798756-nm4p9
Name:             guestbook-86cf798756-nm4p9
Namespace:        default
Priority:         0
Service Account:  default
Node:             node01/172.30.2.2
Start Time:       Sat, 06 Dec 2025 06:11:51 +0000
Labels:           app=guestbook
                  pod-template-hash=86cf798756
Annotations:      cni.projectcalico.org/containerID: fd8b0d20e7ac5c7f0bc3574f7ed72585d9841675ca0aa1ab3e4022e7e4b55cf0
                  cni.projectcalico.org/podIP: 192.168.1.4/32
                  cni.projectcalico.org/podIPs: 192.168.1.4/32
```

This command provides detailed information including:

- Container states
- Restarts and failure reasons
- Events (Scheduling, ImagePull errors, Probe failures)
- Resource requirements
- Node assignment

c) Check Cluster Events

- *kubectl get events -n <namespace>*

```
controlplane:~$ kubectl get events
LAST SEEN   TYPE      REASON                   OBJECT                  MESSAGE
18d         Normal    Starting                 node/controlplane       Starting kubelet.
18d         Normal    NodeHasSufficientMemory  node/controlplane       Node controlplane status is now: NodeHasSufficientMemory
18d         Normal    NodeHasNoDiskPressure    node/controlplane       Node controlplane status is now: NodeHasNoDiskPressure
18d         Normal    NodeHasSufficientPID     node/controlplane       Node controlplane status is now: NodeHasSufficientPID
18d         Normal    NodeAllocatableEnforced  node/controlplane       Updated Node Allocatable limit across pods
18d         Normal    Starting                 node/controlplane       Starting kubelet.
18d         Normal    NodeAllocatableEnforced  node/controlplane       Updated Node Allocatable limit across pods
18d         Normal    NodeHasSufficientMemory  node/controlplane       Node controlplane status is now: NodeHasSufficientMemory
18d         Normal    NodeHasNoDiskPressure    node/controlplane       Node controlplane status is now: NodeHasNoDiskPressure
18d         Normal    NodeHasSufficientPID     node/controlplane       Node controlplane status is now: NodeHasSufficientPID
18d         Normal    RegisteredNode           node/controlplane       Node controlplane event: Registered Node controlplane in Cont
oller
```

Events helped in identifying:

- Image pull issues
- Pod scheduling failures
- Readiness or liveness probe failures
- Networking and configuration errors

Step 3: Viewing Application Logs

Logs were used to analyze runtime behavior inside containers.

a) View Pod Logs

- *kubectl logs <pod-name>*

b) View Logs from a Specific Container

- *kubectl logs <pod-name> -c <container-name>*

c) View Logs from a Previous Container Instance

- *kubectl logs <pod-name> --previous*

This is useful when a container crashed and restarted.

```
controlplane:~$ kubectl logs guestbook-86cf798756-nm4p9
[negroni] listening on :3000
controlplane:~$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
guestbook-86cf798756-nm4p9   1/1     Running   0          3m3s
controlplane:~$ kubectl logs guestbook-86cf798756-nm4p9 -c guestbook
[negroni] listening on :3000
controlplane:~$ kubectl logs  guestbook-86cf798756-nm4p9 --previous
Error from server (BadRequest): previous terminated container "guestbook" in pod "guestbook-86cf798756-nm4p9" n
```

Step 4: Debugging Through Shell Access

When logs were insufficient to diagnose issues inside a container, shell access was used.

a) Create a Pod with Shell Support

- *kubectl create -f https://kubernetes.io/examples/application/shell-demo.yaml*

b) Open Shell Inside the Container

- *kubectl exec -it shell-demo -- /bin/bash*

From here, commands like ls, cat, ps, etc., were used for real-time inspection.

```
controlplane:~$ kubectl create -f https://kubernetes.io/examples/application/shell-demo.yaml
pod/shell-demo created
controlplane:~$ kubectl exec -it shell-demo -- /bin/bash
root@node01:/# ls
bin   dev                docker-entrypoint.sh  home  lib64  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint.d  etc                   lib   media  opt  root  sbin  sys  usr
```

Step 5: Debugging Services

Service-related issues are common in Kubernetes. The following checks were performed:

a) Create a Service if Missing

- *kubectl expose deployment <deployment-name> --type=NodePort --port=3000*

b) List All Services

- *kubectl get svc*

c) Verify Service Configuration

- *kubectl describe svc <service-name>*

```
controlplane:~$ k get deployments.apps
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
guestbook   1/1     1            1           9m40s
controlplane:~$ kubectl expose deployment guestbook --type=NodePort --port=3000
service/guestbook exposed
controlplane:~$ kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
guestbook    NodePort    10.104.10.242   <none>        3000:30560/TCP   9s
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          18d
controlplane:~$ kubectl describe svc guestbook
Name:                     guestbook
Namespace:                default
Labels:                   app=guestbook
Annotations:              <none>
Selector:                 app=guestbook
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.104.10.242
IPs:                      10.104.10.242
Port:                     <unset>  3000/TCP
TargetPort:               3000/TCP
NodePort:                 <unset>  30560/TCP
Endpoints:                192.168.1.4:3000
Session Affinity:         None
```

Important aspects checked:

- TargetPort matching container port
- Correct Service type
- Endpoint availability
- Protocol consistency

Common Issue:

If no endpoints are shown, the service cannot forward traffic to pods.

Step 6: Validating DNS Functionality

DNS issues often lead to service communication problems.

a) Test DNS Resolution

- *kubectl exec -ti <pod-name> -- nslookup kubernetes.default*

b) Inspect /etc/resolv.conf

- *kubectl exec -ti <pod-name> -- cat /etc/resolv.conf*

```
controlplane:~$ kubectl exec -ti guestbook-86cf798756-nm4p9 -- nslookup kubernetes.default
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes.default
Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local
controlplane:~$ kubectl exec -ti guestbook-86cf798756-nm4p9 -- cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
controlplane:~$
```

This ensured that search domains and nameservers were correctly configured.

Step 7: Inspecting NetworkPolicies

NetworkPolicies control inbound and outbound traffic between pods.

Key checks included:

- Confirming if a policy blocks traffic
- Ensuring pods have required labels
- Validating ingress/egress rules
- Testing connectivity using curl / ping

By default, pods accept all traffic unless restricted by NetworkPolicies.

---

5. Results and Observations

This systematic approach helped in identifying common Kubernetes issues such as:

- Missing services
- Misconfigured ports
- Image pull errors
- CrashLoopBackOff conditions
- DNS lookup failures
- Blocked traffic due to NetworkPolicies

---

6. Conclusion

This project provided a comprehensive understanding of Kubernetes debugging and logging techniques. By using various kubectl commands, it was possible to inspect pods, analyze logs, review events, troubleshoot services, and validate networking.