**You have an application deployed on Kubernetes that is experiencing increased traffic. How would you scale the application to handle the increased load?**

Identify the bottleneck: Analyze resource utilization, including CPU, memory, and network, to determine the limiting factor. Horizontal scaling: If the bottleneck is CPU or memory, I would scale the application horizontally by increasing the number of replicas using a Horizontal Pod Autoscaler HPA. Vertical scaling: If the bottleneck is resource-specific, I would vertically scale the application by upgrading the resources allocated to each pod. Monitor and validate: Monitor the applications performance after scaling to ensure the desired scalability is achieved without impacting stability.

**While troubleshooting a networking issue in the cluster, you noticed kube-proxy in the logs. What is the role of kube-proxy in Cluster?**

Kube-proxy is a component in Kubernetes that runs on each node. It handles TCP/UDP packet forwarding between backend network services. It's crucial for reliable communication between pods and services within the cluster by routing traffic to the right destination.

**Your team is planning a high-availability Kubernetes cluster. Describe the Process and Considerations for Designing a High-Availability Kubernetes Cluster.**

To create a high-availability Kubernetes cluster, we followed these steps and considerations in our previous project : Multi-Master Setup We deployed multiple master nodes across three availability zones to ensure redundancy and fault tolerance. This setup prevents a single point of failure at the master level. etcd Distribution We distributed etcd members across the availability zones in a similar fashion as the master nodes. This ensures data redundancy and resilience against zone failures. Load Balancing We configured a TCP load balancer to evenly distribute API requests among the API servers. This setup eliminates the risk of a single API server becoming a bottleneck or point of failure. Node Auto-Repair We enabled node auto-repair feature provided by Kubernetes Engine. This feature automatically detects and replaces unhealthy nodes, maintaining the desired cluster size and health without manual intervention.

**Real time Kubernetes problem solution:**

[**https://hervekhg.medium.com/debugging-kubernetes-issues-in-production-a-technical-guide-9e3d26e27180**](https://hervekhg.medium.com/debugging-kubernetes-issues-in-production-a-technical-guide-9e3d26e27180)

**1. Understanding cluster health**

The first step in debugging an issue is to check the overall health of the cluster. This overview helps identify failing components or resources under pressure.

**Checking node status**

Nodes are the backbone of your cluster. If a node encounters issues, the pods scheduled on it may also be affected.

# Get list of nodes  
kubectl get nodes

If all nodes are ok get the detail information:

# Get detailled information about nodes  
kubectl describe node <node-name>

This command provides details on why a node might be NotReady, such as memory issues, disk pressure, or taints not tolerated by the pods, pods running on the nodes, label... It’s a powerful command that will help you to get detailed informations about nodes.

**b. Checking core components**

Critical components like kube-apiserver, kube-scheduler, and kube-controller-manager must function correctly to ensure cluster health.

# The command below is depreciated up to kubernetes 1.9  
kubectl get componentstatuses  
  
# You can use this command instead  
kubectl get --raw='/readyz?verbose'

This command checks the status of the core components. If any component is marked as Unhealthy, it requires immediate attention.

**2. Diagnosing pod issues**

Pods are the units of execution in Kubernetes, and most production issues manifest as failing pods.

**a. Get all pods that are not in the running state**

This command will scan all the pods of all the namespace and list those that are not in a running state. It’s a good command to quickly identified pod bad status

# Get all the pods of all the namespace that are not in Running state  
kubect get pod -A | grep -v "Running"

**b. Inspecting pod status**

Instead of scanning all the namespace, you can checking the overall status of the pods in the relevant namespace:

kubectl get pods -n <namespace-name>

For additional details on a specific pod, use:

kubectl describe pod <pod-name> -n <namespace-name>

This command shows recent events, errors, and status messages.

**b. Exploring container logs**

If a pod fails or behaves unexpectedly, container logs often provide valuable clues:

kubectl logs <pod-name> -n <namespace-name> --previous

The --previous flag is particularly useful for viewing logs from a container that crashed and was restarted.

You can also tail the last x line of your pod logs using this command

# Get 200 last line of your pod logs  
kubectl logs <pod-name> -n <namespace-name> --tail=200

The --tail flag is particularly useful for viewing the previous x logs of the container. If you want to see the logs in live, you can replace them with the option -f without any number

kubectl logs <pod-name> -n <namespace-name> -f

**c. Accessing a live container**

For deeper exploration, it might be necessary to run commands inside a live container:

kubectl exec -it <pod-name> -n <namespace-name> -- /bin/bash

This opens an interactive shell, allowing you to inspect files, configuration, or run diagnostics directly in the container

**3. Resolving network issues**

Network issues are common in distributed environments like Kubernetes. Here are some steps to diagnose connectivity problems.

**Checking services and endpoints**

Kubernetes services abstract pods and provide a stable network interface. If a service is not functioning as expected, start by checking if it is correctly configured:

kubectl get svc -n <namespace-name>

Next, check the associated endpoints to ensure the backend pods are correctly linked:

kubectl get endpoints <service-name> -n <namespace-name>

A service with no endpoints may indicate that the backend pods are failing or unavailable.

**Debugging ingress and load balancers**

kubectl describe ingress <ingress-name> -n <namespace-name>

Kubernetes Ingress is needed to manage and control external access to services within a cluster, providing a centralized and flexible way to route traffic and secure access. It simplifies the process of exposing multiple services under a single IP address, allowing for load balancing, SSL termination, and routing based on rules defined by the Kubernetes API.

**Testing intra cluster connectivity**

kubectl exec -it <source-pod-name> -n <namespace-name> -- curl <destination-service>:<port>

If connectivity fails, it could indicate a deeper network problem, such as a firewall rule blocking traffic or a misconfigured network policy.

**4. Using advanced debugging tools**

**kubectl-debug**

The kubectl-debug tool allows you to launch a debug container (like busybox or alpine) attached to an existing pod, giving you more flexibility to inspect the runtime environment:

kubectl debug <pod-name> -n <namespace-name> --image=busybox --target=<container-name>