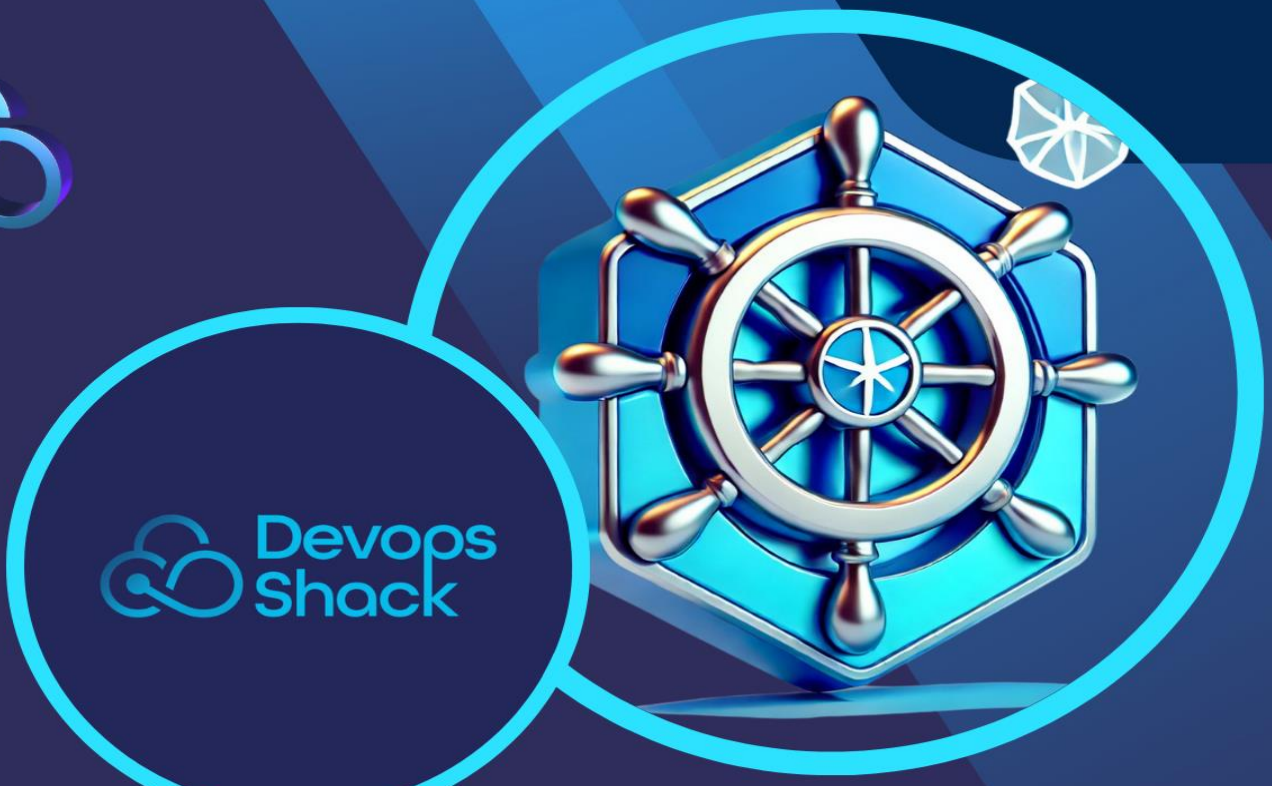

Devops Shack

KUBERNETES

REAL-TIMS SCENARIOS WITH SOLUTIONS

2025

www.devopsshackcom

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Real-Time Kubernetes Scenarios with Solutions

Table of Contents

16.Service Not Accessible

17.Pods Not Terminating Gracefully

18.Cluster Autoscaler Not Scaling Nodes

19.ConfigMap Changes Not Applied

20.Pod Security Violations

21.Deployment Not Using Updated ConfigMap or Secret

22.Failed Horizontal Pod Autoscaler Due to Missing Metrics

23.Pods Scheduled on Unsuitable Nodes

24.Service Discovery Fails in StatefulSets

25.Unauthenticated API Access

26.Data Loss in Stateful Applications

27.Cluster Network Performance Issues

28.API Server High Latency

29.Deployment Not Recovering After Node Failure

30.Securing Kubernetes Secrets

31.Application Downtime During Deployment

32.Cross-Cluster Communication Issues

33.Certificate Expiration in Ingress

34.Namespace Resource Quota Issues

35.External Service Integration Fails

36.Overprovisioning Nodes

37.PersistentVolume Bound to Wrong Claim

38.Services Unable to Communicate Across Namespaces

39.Horizontal Pod Autoscaler Not Scaling Pods

40.Pods Stuck in Terminating State

41.Service Port Conflicts

42. Environment Variable Misconfiguration
43. Pods Failing Readiness Probes
44. Network Plugin Misconfiguration
45. Rolling Back a Faulty Deployment
46. Monitoring Pods with Prometheus and Grafana
47. Pod Affinity and Anti-Affinity Issues
48. Pod Security Policies (PSP) Misconfiguration
49. Kubernetes API Auditing and Security
50. Leveraging External Secret Management Tools

Conclusion

1. Summary of Key Takeaways
2. Best Practices for Kubernetes Operations
3. Encouragement to Explore More Advanced Scenarios

Introduction

Kubernetes has become the cornerstone of modern container orchestration, empowering organizations to deploy, scale, and manage containerized applications effortlessly. As a platform designed to handle complex workloads across diverse environments, Kubernetes simplifies the deployment process while providing tools for automation, scalability, and resilience.

However, working with Kubernetes in real-world scenarios comes with its own set of challenges. From managing resources effectively to ensuring high availability, and from debugging failing pods to configuring network policies, administrators and developers often face a myriad of issues during their day-to-day operations.

This guide aims to address these challenges by presenting **50 real-time Kubernetes scenarios** commonly encountered in production environments. Each scenario is accompanied by a detailed explanation and practical solutions, including configuration examples and troubleshooting steps.

Whether you're an experienced DevOps engineer or a beginner in Kubernetes, this guide is designed to:

- **Identify** the root causes of common Kubernetes issues.
- **Provide** actionable solutions with clear, step-by-step instructions.
- **Equip** you with the knowledge to handle complex Kubernetes environments confidently.

By mastering these scenarios, you'll gain insights into best practices and strategies that not only solve immediate problems but also help build a robust and reliable Kubernetes infrastructure.

Scenario 1: Application Crash Due to Insufficient Resources

Problem: Your application pods frequently crash or restart due to insufficient CPU or memory allocation.

Solution: To prevent this, define **resource requests and limits** in your pod or deployment YAML file. These settings ensure the pod gets guaranteed resources and limits its usage to avoid overconsumption.

Code Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: resource-limited-app
spec:
replicas: 2
selector:
matchLabels:
app: resource-limited-app
template:
metadata:
labels:
app: resource-limited-app
spec:
containers:
- name: app-container
image: nginx:1.21
resources:
requests:
memory: "256Mi"
cpu: "500m"
limits:
memory: "512Mi"
cpu: "1"
```

Explanation:

- requests: The minimum amount of resources (CPU/Memory) Kubernetes guarantees for the pod.
- limits: The maximum resources the pod can use.
- Here, the container requests 256Mi of memory and 500m (0.5 cores) of CPU and cannot use more than 512Mi memory and 1 CPU core.

Scenario 2: High Pod Latency Due to Load Balancer Configuration

Problem: Your application experiences high latency because the default load balancer isn't optimized for traffic.

Solution:
You can customize the load balancer settings to handle traffic more efficiently by enabling session affinity (stickiness). This ensures subsequent requests from the same client are routed to the same pod.

Code Example:

```
apiVersion: v1
kind: Service
metadata:
name: app-service
spec:
selector:
app: my-app
ports:
    protocol: TCP
    port: 80
    targetPort: 8080
    type: LoadBalancer
    sessionAffinity: ClientIP
```

Explanation:

- sessionAffinity: ClientIP ensures requests from the same client IP are routed to the same pod, reducing latency caused by unnecessary re-routing.

- You can also tune the external load balancer (e.g., AWS ELB or GCP) using annotations specific to your cloud provider.

Scenario 3: Pods Stuck in Pending State

Problem: Pods remain in the Pending state because there aren't enough nodes to schedule them.

Solution:

1. Check the reason using the kubectl describe pod command.
2. If the issue is due to resource constraints, you can add more nodes to the cluster or adjust the pod's resource requests.

Code Example:

```
kubectl describe pod <pod-name>
kubectl scale --replicas=3 deployment/my-deployment
```

Explanation:

- Inspect the event log with kubectl describe pod to identify the issue.
- Add nodes by scaling the cluster or increasing resource capacity on existing nodes.

Scenario 4: Inconsistent DNS Resolution

Problem: Applications fail to resolve internal services due to DNS misconfiguration.

Solution:
Ensure the kube-dns or CoreDNS service is running and correctly configured. You may need to increase the DNS cache size or adjust stub domains.

Code Example:

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
kubectl edit configmap coredns -n kube-system
```

Add the following line to the ConfigMap:
cache { success 1000 negative 100 }

Explanation:

- Use kubectl get to check DNS service health.
- Editing the CoreDNS ConfigMap can increase cache size, reducing DNS query latency.

Scenario 5: Pod Fails Due to Liveness Probe Misconfiguration

Problem: Pods are restarting because the liveness probe is not properly configured.

Solution:
Set appropriate liveness and readiness probes based on your application's health endpoint.

Code Example:

livenessProbe:
httpGet:
path: /healthz
port: 8080
initialDelaySeconds: 10
periodSeconds: 5

Explanation:

- httpGet: Kubernetes checks the /healthz endpoint.
- initialDelaySeconds: Waits 10 seconds before starting the probe.
- periodSeconds: Checks every 5 seconds.

Scenario 6: Cluster High Availability Issues

Problem: The control plane fails, leading to downtime.

Solution:
Deploy a multi-master cluster for High Availability (HA) with an external etcd cluster and load balancer.

Code Example:

- Configure multiple master nodes during kubeadm initialization.
- Use an external load balancer (e.g., HAProxy) to distribute traffic across masters.

Scenario 7: Outdated Application Image in Pods

Problem: Updated application code is not reflected in running pods.

Solution:
Use kubectl rollout to trigger a rolling update or specify an image pull policy.

Code Example:

```
kubectl set image deployment/my-deployment my-container=nginx:latest
kubectl rollout status deployment/my-deployment
```

Scenario 8: Excessive Logs Filling Disk Space

Problem: Pods generate excessive logs, consuming disk space.

Solution:
Implement log rotation and centralize logs using tools like Fluentd or ELK.

Code Example:

```
logging: options: max-size: "200m"
max-file: "3"
```

Scenario 9: Node Fails Health Check

Problem: Nodes fail health checks due to disk pressure or memory usage.

Solution:
Monitor nodes using kubectl describe node. Use taints and tolerations to manage workloads effectively.

Code Example:

```
kubectl taint nodes <node-name> key=value:NoSchedule
```

Scenario 10: Application Deployment Fails

Problem: Deployment fails due to YAML misconfiguration or invalid image.

Solution:
Validate YAML files before applying them and check image pull secrets if using private registries.

Code Example:

```
Use tools like kubeval to validate your YAML.
kubectl logs pod/<pod-name>
```

Scenario 11: CrashLoopBackOff Issue

Problem: Pods keep restarting, and the status shows CrashLoopBackOff.

Solution:

1. Use the kubectl logs command to check the pod logs for errors.
2. Ensure the application is not exiting unexpectedly due to misconfigurations or missing dependencies.

Code Example:

```
kubectl logs <pod-name>
kubectl describe pod <pod-name>
```

Explanation:

- Check the logs to identify the root cause (e.g., database connection errors, missing environment variables).
- Correct the issue and redeploy the pod. For example, if an environment variable is missing, add it to the deployment YAML.

Scenario 12: PersistentVolumeClaim Stuck in Pending State

Problem: Your PersistentVolumeClaim (PVC) remains in Pending because no matching PersistentVolume (PV) is available.

Solution:

1. Ensure a PV exists that satisfies the PVC's storage class, size, and access mode.
2. If not, create a new PV.

Code Example:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: my-pv
spec:
capacity:
storage: 5Gi
accessModes:
- ReadWriteOnce
persistentVolumeReclaimPolicy: Retain
storageClassName: standard
```

Explanation:

- Define a PV that matches the PVC's specifications.
- PVCs are automatically bound to matching PVs when available.

Scenario 13: Failed to Pull Image

Problem: Pods fail to start because the image can't be pulled from the registry.

Solution:

1. Verify the image name and tag are correct.
2. If using a private registry, create a Kubernetes secret and reference it in your deployment.

Code Example:

```
kubectl create secret docker-registry my-registry-secret --docker-server=<registry> --docker-username=<username> --docker-password=<password>
```

Explanation:

- Use the secret in your deployment under the imagePullSecrets section to authenticate with the private registry.

Scenario 14: NetworkPolicy Blocking Traffic

Problem: Pods can't communicate with other pods or external services due to NetworkPolicy restrictions.

Solution:
Ensure you have an appropriate NetworkPolicy allowing the required traffic.

Code Example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: allow-app-traffic
spec:
podSelector:
matchLabels:
app: my-app
policyTypes:
- Ingress
ingress:
- from:
- podSelector:
matchLabels:
app: web
```

Explanation:

- This policy allows traffic from pods labeled app: web to pods labeled app: my-app.

Scenario 15: Unable to Scale Deployment

Problem: Running kubectl scale doesn't increase pod replicas.

Solution:

1. Verify if the cluster has enough resources to schedule new pods.
2. Check for custom Horizontal Pod Autoscaler (HPA) configurations that might override manual scaling.

Code Example:

```
kubectl scale deployment my-deployment --replicas=5
```

Explanation:

- Ensure nodes have sufficient capacity.
- Disable or update the HPA if it's limiting replica scaling.

Scenario 16: Service Not Accessible

Problem: The application service is not accessible externally.

Solution:

1. Ensure the service is of type LoadBalancer or NodePort.
2. Check for firewall rules or cloud provider-specific configurations.

Code Example:

```
apiVersion: v1
kind: Service
metadata:
name: my-service
spec:
selector:
app: my-app
ports:
- protocol: TCP
port: 80
targetPort: 8080
type: LoadBalancer
```

Explanation:

- Using type: LoadBalancer exposes the service externally.
- Ensure the cloud provider provisions a public IP.

Scenario 17: Pods Not Terminating Gracefully

Problem: Pods fail to terminate gracefully, causing disruptions during updates.

Solution: Configure termination grace periods and preStop hooks to handle cleanup tasks before the pod shuts down.

Code Example:

```
lifecycle:
preStop:
exec:
command: ["/bin/sh", "-c", "cleanup.sh"]
terminationGracePeriodSeconds: 30
```

Explanation:

- preStop: Executes a script or command to perform cleanup (e.g., closing connections).
- terminationGracePeriodSeconds: Ensures the container has enough time to complete cleanup tasks.

Scenario 18: Cluster Autoscaler Not Scaling Nodes

Problem: The cluster autoscaler doesn't add nodes when pods are pending due to insufficient resources.

Solution:

1. Verify the autoscaler configuration.
2. Ensure the pod's resource requests fit within node limits.

Code Example:

```
kubectl get configmap cluster-autoscaler-status -n kube-system
```

Explanation:

- Adjust the cluster autoscaler parameters, such as max and min node counts.
- Ensure resource requests are not too large for available nodes.

Scenario 19: ConfigMap Changes Not Applied

Problem: Changes to a ConfigMap are not reflected in running pods.

Solution:
Restart the pods or update the deployment to pick up the changes.

Code Example:

```
kubectl rollout restart deployment/my-deployment
```

Explanation:

- ConfigMap changes don’t automatically propagate to running pods.
- Rolling out a restart ensures new pods pick up the updated ConfigMap.

Scenario 20: Pod Security Violations

Problem: Pods fail to start due to security policies, such as running as root.

Solution:
Create a PodSecurityPolicy that permits specific actions or adjust the pod's security context.

Code Example:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
name: allow-non-root
spec:
runAsUser:
rule: MustRunAsNonRoot
```

Explanation:

- This policy enforces that pods must not run as root.

- Update pod definitions to include a securityContext section if necessary.

Scenario 21: PersistentVolume Bound to Wrong Claim

Problem: A PersistentVolume (PV) is accidentally bound to the wrong PersistentVolumeClaim (PVC).

Solution:

1. Delete the PVC and recreate it with correct specifications.
2. Ensure the PV is properly configured with a unique claimRef.

Code Example:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: my-correct-pv
spec:
capacity:
storage: 10Gi
accessModes:
- ReadWriteOnce
persistentVolumeReclaimPolicy: Retain
claimRef:
namespace: default
name: my-correct-pvc
```

Explanation:

- claimRef links a PV to a PVC. Verify this before binding.
- Use persistentVolumeReclaimPolicy: Retain to prevent automatic deletion when the PVC is deleted.

Scenario 22: Services Unable to Communicate Across Namespaces

Problem: Applications in different namespaces can't access each other's services.

Solution:
Use the service's fully qualified domain name (FQDN) in the format <service-name>.<namespace>.svc.cluster.local.

Code Example:

```
apiVersion: v1
kind: Service
metadata:
name: my-service
namespace: namespace-a
spec:
selector:
app: my-app
ports:
- port: 80
```

Explanation:

- Use the service's FQDN (my-service.namespace-a.svc.cluster.local) to access it from other namespaces.
- Verify network policies don’t block inter-namespace traffic.

Scenario 23: Horizontal Pod Autoscaler Not Scaling Pods

Problem: The Horizontal Pod Autoscaler (HPA) isn't increasing the number of pods under high load.

Solution:

1. Ensure metrics-server is running and properly configured.
2. Verify resource requests/limits are defined in the deployment.

Code Example:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
name: my-hpa
spec:
scaleTargetRef:
```

```
apiVersion: apps/v1
kind: Deployment
name: my-deployment
minReplicas: 2
maxReplicas: 10
metrics:
- type: Resource
resource:
name: cpu
target:
type: Utilization
averageUtilization: 50
```

Explanation:

- metrics-server collects and provides resource usage data.
- Ensure the target CPU utilization is appropriate for your application.

Scenario 24: Ingress Controller Not Routing Traffic

Problem: The ingress controller is not routing requests to backend services.

Solution:
Verify the ingress resource configuration and ensure the ingress controller is running correctly.

Code Example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: my-ingress
spec:
rules:
- host: example.com
http:
paths:
- path: /
pathType: Prefix
backend:
```

```
service:
name: my-service
port:
number: 80
```

Explanation:

- Ensure the DNS resolves to the ingress controller's IP.
- Use the correct annotations for your ingress controller (e.g., NGINX, Traefik).

Scenario 25: Pod Image Pull BackOff

Problem: Pods fail to start with an ImagePullBackOff error.

Solution:

1. Check if the image name or tag is incorrect.
2. If using a private registry, create a docker-registry secret and reference it in your pod.

Code Example:

```
kubectl create secret docker-registry my-secret --docker-server=<registry> --docker-username=<username> --docker-password=<password>
```

Explanation:

- Add the secret to the deployment under imagePullSecrets.
- Ensure the registry is reachable from the cluster.

Scenario 26: Node Becomes Unschedulable

Problem: A node is marked Unschedulable, and pods can't be scheduled on it.

Solution:
Uncordon the node or troubleshoot the taints that are preventing scheduling.

Code Example:

```
kubectl uncordon <node-name>
```

Explanation:

- Use kubectl describe node to check for taints.
- Remove the taint if it's unnecessary or adjust pod tolerations to handle it.

Scenario 27: Application Fails After Node Restart

Problem: Application data is lost or corrupted after a node restarts.

Solution:
Use PersistentVolumes backed by reliable storage (e.g., NFS, cloud storage) to ensure data persistence.

Code Example:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: app-data-pv
spec:
capacity:
storage: 20Gi
accessModes:
- ReadWriteOnce
persistentVolumeReclaimPolicy: Retain
```

Explanation:

- Volumes ensure data persists beyond pod lifecycles or node restarts.
- Use a ReclaimPolicy of Retain for manual recovery.

Scenario 28: Pod Security Context Issues

Problem: Pods are unable to run due to security context restrictions.

Solution:
Define appropriate security contexts for pods or containers.

Code Example:

```
securityContext:
runAsUser: 1000
runAsGroup: 3000
fsGroup: 2000
```

Explanation:

- runAsUser specifies the user ID to run the container.
- fsGroup sets the file system group ID, ensuring proper permissions for mounted volumes.

Scenario 29: Configuring Readiness Probe

Problem: The application is marked Unready by the readiness probe, affecting traffic routing.

Solution:
Configure the readiness probe with an appropriate endpoint and delay.

Code Example:

```
readinessProbe:
httpGet:
path: /readiness
port: 8080
initialDelaySeconds: 5
periodSeconds: 10
```

Explanation:

- The readiness probe prevents the pod from receiving traffic until it's ready to serve.
- Adjust initialDelaySeconds and periodSeconds based on application startup time.

Scenario 30: High CPU/Memory Usage on Nodes

Problem: Nodes experience high CPU or memory usage, causing pod evictions.

Solution:

1. Monitor node resource usage using tools like kubectl top.
2. Tweak resource requests/limits for pods and add new nodes if required.

Code Example:

```
kubectl top nodes
kubectl set resources deployment my-deployment --
limits=cpu=500m,memory=512Mi
```

Explanation:

- Limit resource usage to prevent overloading nodes.
- Use autoscalers to dynamically manage cluster resources.

Scenario 31: Unable to Roll Back a Deployment

Problem: A deployment is updated with a faulty configuration, and you need to roll back to the previous version.

Solution:
Use Kubernetes' built-in rollout history to revert to the last stable state.

Code Example:

```
kubectl rollout undo deployment/my-deployment
```

Explanation:

- Kubernetes maintains a history of deployment revisions.
- You can roll back to the most recent stable version or a specific revision using --to-revision=<number>.

Scenario 32: Environment Variable Misconfiguration

Problem: Pods fail to start due to missing or misconfigured environment variables.

Solution:
Define environment variables correctly in the deployment YAML or use a ConfigMap.

Code Example:

```
env:
    name: DB_HOST
    valueFrom:
    configMapKeyRef:
    name: app-config
    key: database-host
```

Explanation:

- Use valueFrom to fetch environment variables from ConfigMaps or Secrets.
- Ensure the ConfigMap or Secret exists before deploying the pods.

Scenario 33: Service Port Conflicts

Problem: Two services attempt to use the same port, causing traffic routing issues.

Solution:
Use unique port and targetPort values for each service.

Code Example:

```
apiVersion: v1
kind: Service
metadata:
name: service-a
spec:
ports:
- port: 80
targetPort: 8080
```

```
apiVersion: v1
kind: Service
metadata:
name: service-b
spec:
ports:
- port: 81
targetPort: 8081
```

Explanation:

- Ensure each service uses a distinct port for external access.
- targetPort maps to the container port internally.

Scenario 34: Pods Stuck in Terminating State

Problem: Pods fail to terminate and remain stuck in Terminating state.

Solution:
Force delete the pod or identify the blocking finalizers.

Code Example:

```
kubectl delete pod <pod-name> --force --grace-period=0
```

Explanation:

- Force deletion removes the pod immediately, bypassing termination grace periods.
- Use kubectl describe pod to check for finalizers that may block termination.

Scenario 35: Application Downtime During Deployment

Problem: Users experience downtime during application updates.

Solution:
Enable rolling updates with proper health checks in your deployment.

Code Example:

```
strategy:
type: RollingUpdate
rollingUpdate:
maxUnavailable: 1
maxSurge: 1
```

Explanation:

- Rolling updates replace pods incrementally, minimizing downtime.

- Use maxUnavailable and maxSurge to control the number of pods updated at a time.

Scenario 36: Cross-Cluster Communication Issues

Problem: Applications in one Kubernetes cluster can't communicate with another cluster.

Solution:
Implement a multi-cluster network solution like Istio or Linkerd to enable service discovery and communication.

Code Example:

```
kubectl apply -f istio-multi-cluster.yaml
```

Explanation:

- Multi-cluster mesh solutions provide secure cross-cluster communication.
- Ensure the networking setup includes proper service discovery configurations.

Scenario 37: Certificate Expiration in Ingress

Problem: TLS certificates used by the ingress controller expire, causing HTTPS connections to fail.

Solution:
Automate certificate management using Cert-Manager with Let's Encrypt.

Code Example:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
name: tls-cert
namespace: default
spec:
secretName: tls-secret
issuerRef:
```

```
name: letsencrypt-prod
kind: ClusterIssuer
dnsNames:
- example.com
```

Explanation:

- Cert-Manager automatically renews TLS certificates.
- Use a ClusterIssuer like Let's Encrypt for automated certificate issuance.

Scenario 38: Namespace Resource Quota Issues

Problem: A namespace exceeds its resource quota, preventing new pods from being scheduled.

Solution:
Check and update the namespace's resource quota or allocate a larger quota.

Code Example:

```
apiVersion: v1
kind: ResourceQuota
metadata:
name: namespace-quota
spec:
hard:
pods: "50"
requests.cpu: "10"
requests.memory: "10Gi"
limits.cpu: "20"
limits.memory: "20Gi"
```

Explanation:

- ResourceQuota limits resource usage within a namespace.
- Adjust quota values to meet application needs.

Scenario 39: External Service Integration Fails

Problem: An application cannot connect to an external database or API due to firewall rules.

Solution:
Configure an egress policy or use a NAT gateway for outgoing traffic.

Code Example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: allow-egress
spec:
podSelector: {}
policyTypes:
- Egress
egress:
- to:
- ipBlock:
cidr: 192.168.1.0/24
```

Explanation:

- Use a NetworkPolicy to allow egress traffic to specific external IPs.
- Verify firewall rules on the external service.

Scenario 40: Overprovisioning Nodes

Problem: Nodes are underutilized due to overprovisioning of resources.

Solution:
Use Kubernetes resource requests and cluster autoscalers to optimize resource allocation.

Code Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: optimized-app
spec:
```

```
replicas: 3
template:
spec:
containers:
- name: app-container
resources:
requests:
memory: "512Mi"
cpu: "500m"
limits:
memory: "1Gi"
cpu: "1"
```

Explanation:

- Properly defined resource requests and limits prevent overprovisioning.
- Autoscalers adjust the number of nodes and pods based on actual usage.

Scenario 41: Deployment Not Using Updated ConfigMap or Secret

Problem: ConfigMap or Secret changes are not reflected in the application after an update.

Solution:
Pods must be restarted to reflect updated ConfigMap or Secret values. Automate this using annotations.

Code Example:

```
annotations:
configmap.reloader.stakater.com/reload: "true"
```

Explanation:

- Use tools like reloader (e.g., Stakater's Reloader) to detect ConfigMap/Secret changes and restart affected pods automatically.

Scenario 42: Failed Horizontal Pod Autoscaler Due to Missing Metrics

Problem: HPA doesn't work because the metrics-server is missing or misconfigured.

Solution:
Install and configure the metrics-server in your cluster.

Code Example:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Explanation:

- Metrics-server provides CPU/memory usage data for HPA.
- Ensure the metrics-server pod is running in the kube-system namespace.

Scenario 43: Pods Scheduled on Unsuitable Nodes

Problem: Pods are scheduled on nodes that don't meet resource or hardware requirements.

Solution:
Use node selectors or affinity rules to specify node requirements.

Code Example:

```
nodeSelector:
disktype: ssd
```

```
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: disktype
operator: In
values:
- ssd
```

Explanation:

- nodeSelector allows you to specify a simple key-value requirement.

- nodeAffinity provides advanced filtering with operators like In, NotIn, etc.

Scenario 44: Service Discovery Fails in StatefulSets

Problem: Pods in a StatefulSet cannot discover each other due to DNS issues.

Solution:
Use the stable network identity provided by StatefulSets.

Code Example:

```
statefulset-name-<ordinal>.statefulset-name.<namespace>.svc.cluster.local
```

Explanation:

- Pods in a StatefulSet have predictable names and DNS.
- Use this naming convention for internal service discovery.

Scenario 45: Unauthenticated API Access

Problem: Unauthenticated users access sensitive Kubernetes API endpoints.

Solution:
Implement Role-Based Access Control (RBAC) to restrict API access.

Code Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
namespace: default
name: pod-reader
rules:
```

- ```
  apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
  ```

Explanation:

- Use RBAC roles to define resource access permissions.
  ```

- Bind roles to specific users or groups with RoleBinding.

Scenario 46: Data Loss in Stateful Applications

Problem: Data is lost when a pod in a StatefulSet is deleted or rescheduled.

Solution:
Attach PersistentVolumeClaims to each pod in the StatefulSet.

Code Example:

volumeClaimTemplates:

```
metadata:
name: data-volume
spec:
accessModes: ["ReadWriteOnce"]
resources:
requests:
storage: 10Gi
```

Explanation:

- Each pod gets its own PersistentVolume, ensuring data persistence across reschedules.
- Use volumeClaimTemplates in StatefulSets for dynamic volume provisioning.

Scenario 47: Cluster Network Performance Issues

Problem: Network latency or packet loss affects application performance.

Solution:
Use a network plugin (CNI) like Calico or Flannel and optimize its configuration.

Code Example:

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Explanation:

- Network plugins manage pod-to-pod and pod-to-service communication.

- Calico allows additional network policies for fine-grained control.

Scenario 48: API Server High Latency

Problem: The Kubernetes API server is slow due to excessive requests.

Solution:
Rate-limit API calls and use caching tools like kube-apiserver audit logs.

Code Example:

```
auditPolicy:
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: RequestResponse
verbs: ["create", "update", "patch"]
```

Explanation:

- Audit logs monitor high-frequency API calls.
- Optimize applications making frequent API requests.

Scenario 49: Deployment Not Recovering After Node Failure

Problem: Pods on a failed node are not rescheduled to other nodes.

Solution:
Use pod anti-affinity rules and node taints/tolerations for better fault tolerance.

Code Example:

```
tolerations:
    key: "node.kubernetes.io/unreachable"
    operator: "Exists"
    effect: "NoExecute"
    tolerationSeconds: 300
```

Explanation:

- Tolerations allow pods to remain scheduled even if nodes are temporarily unreachable.
- Use anti-affinity rules to spread pods across multiple nodes.

Scenario 50: Securing Kubernetes Secrets

Problem: Secrets are stored in plain text and vulnerable to unauthorized access.

Solution:
Use external secret management tools like HashiCorp Vault or AWS Secrets Manager and integrate them with Kubernetes.

Code Example:

```
kubectl create secret generic db-secret --from-literal=username=admin --from-literal=password=securepass
```

Explanation:

- Store sensitive data as Kubernetes Secrets.
- Use tools like Vault for encryption and dynamic secret management.

Conclusion

Kubernetes is a powerful and flexible tool for managing containerized applications, but its complexity often presents challenges in real-world scenarios. From addressing resource limitations and network policies to managing high availability and ensuring secure configurations, mastering Kubernetes requires both practical experience and a strong understanding of its core principles.

Through this guide, we've explored **50 real-time Kubernetes scenarios**, covering a wide range of challenges that administrators, developers, and DevOps engineers frequently encounter. Each scenario was paired with clear explanations and actionable solutions to help you navigate the complexities of Kubernetes with confidence.

Key Takeaways:

1. **Resource Management**: Properly configuring resource requests, limits, and scaling mechanisms ensures efficient utilization of cluster resources.
2. **Networking**: Mastering NetworkPolicies, DNS configurations, and cross-cluster communication is critical for seamless operations.
3. **Persistence and State Management**: Using PersistentVolumes and StatefulSets effectively can safeguard data integrity and application stability.
4. **Security**: Implementing Role-Based Access Control (RBAC), Pod Security Policies (PSP), and secret management tools enhances cluster security.
5. **Automation and Monitoring**: Leveraging tools like Horizontal Pod Autoscalers, metrics-server, and monitoring solutions such as Prometheus and Grafana simplifies cluster management and debugging.
6. **Best Practices**: Following Kubernetes best practices—such as rolling updates, readiness/liveness probes, and proper use of namespaces—ensures high availability and minimizes downtime.

Kubernetes is a continuously evolving ecosystem, and its vast range of features allows for tremendous flexibility in application deployment and management. By understanding and applying the solutions presented in these scenarios, you will not only resolve specific issues but also develop a deeper comprehension of how Kubernetes operates.

As you continue your journey with Kubernetes, remember that the key to success lies in consistent learning, experimentation, and staying updated with the latest advancements in the Kubernetes ecosystem. Embrace the challenges, and you'll find Kubernetes to be a powerful ally in building modern, scalable, and resilient applications.

Happy orchestrating!