Hey there, fellow DevOps adventurers!

Are you as excited as I am about the ever-evolving world of **Cloud, Kubernetes, Infrastructure as Code, CI/CD pipelines**, and beyond?

I'm sharing hands-on tips, real-world solutions, and game-changing strategies to help you navigate the DevOps universe like a pro!

Whether you're a beginner, a seasoned engineer, or just curious, there's always something new to learn here.

Let's make this a two-way conversation!

💬 Got questions? Drop them in the comments!
📢 Found something insightful? **Repost and share with your network**--let's grow the DevOps community together!

Together, we're not just talking DevOps, we're living it.

Don't forget to **hit follow** to stay updated with every new post!

in **Follow** Medium

Kubernetes Scenario-Based Interview Questions with Answers

Scenario 1: Performance Degradation Investigation

Situation: Your production Kubernetes cluster is experiencing severe API server slowdowns. Users report that `kubectl` commands are taking 30+ seconds to respond, and some are timing out completely. The cluster was performing normally until yesterday.

Question 1: What would be your initial approach to diagnose this performance issue?

Answer:

- Check cluster resource utilization (CPU, memory, disk I/O) on all nodes
- Examine API server metrics using Prometheus or monitoring tools
- Review API server logs for error patterns, warnings, or authentication issues
- Check etcd health and performance metrics
- Monitor network latency between control plane components
- Verify if the issue affects all API operations or specific resource types
- Check for any recent changes or deployments that might correlate with the issue

Question 2: Which metrics and logs would you examine first?

Answer:

- **API Server Metrics:**
  - `apiserver_request_duration_seconds` (request latency)
  - `apiserver_request_total` (request rate and error codes)
  - `apiserver_current_inflight_requests` (concurrent requests)
- **etcd Metrics:**
  - `etcd_disk_wal_fsync_duration_seconds` (disk performance)
  - `etcd_request_duration_seconds` (etcd request latency)
- **Event Metrics:**
  - `apiserver_audit_events_total` (event generation rate)
- **Logs to examine:**
  - API server logs: `/var/log/pods/kube-apiserver-*`
  - etcd logs for performance warnings
  - kubelet logs for authentication issues

Question 3: If you discovered that the etcd datastore is overwhelmed with excessive write operations, what could be the potential causes?

Answer:

- **Event flooding:** Controllers or applications generating excessive events (50+ events/second)
- **Frequent configuration changes:** Rapid ConfigMap or Secret updates triggering reconciliation loops
- **Inefficient controllers:** Custom controllers with poor logic causing continuous reconciliation
- **Large-scale operations:** Bulk resource creation/deletion operations
- **Lack of rate limiting:** No throttling on client API requests
- **Watch operations:** Too many clients watching large resource sets
- **Debugging left enabled:** Verbose logging or debug flags increasing write volume

Question 4: How would you identify which component or application might be generating excessive API calls?

Answer:

- **Event analysis:** `kubectl get events --sort-by=.metadata.creationTimestamp | head -50`
- **Audit logging:** Enable API server audit logging to track request sources
- **Prometheus queries:** Check client-specific request rates by user-agent
- **Controller logs:** Examine controller-manager and custom controller logs for reconciliation patterns
- **Resource monitoring:** Use `kubectl top` to identify resource-intensive operations
- **Network analysis:** Monitor API server connection patterns and source IPs

Question 5: What preventive measures would you implement to avoid similar issues in the future?

Answer:

- **Rate limiting:** Configure `--event-rate-limit` and `--event-burst-limit` on API server
- **Event management:** Set `--event-ttl=1h` to limit event retention
- **Controller improvements:** Implement event deduplication and rate limiting in custom controllers
- **Monitoring:** Set up alerts for API server latency and event generation rates
- **Resource policies:** Implement ResourceQuotas and LimitRanges
- **Regular maintenance:** Schedule etcd defragmentation and cleanup procedures
- **Testing:** Load test API server before deploying changes

Scenario 2: DNS Resolution Failure

Situation: Multiple applications in your cluster suddenly cannot resolve internal service names. Pods are failing with DNS lookup errors, and inter-service communication has broken down. External DNS resolution appears to be working fine.

Question 1: How would you systematically troubleshoot DNS resolution problems in Kubernetes?

Answer:

1. **Verify CoreDNS status:** `kubectl get pods -n kube-system -l k8s-app=kube-dns`
2. **Check DNS service:** `kubectl get svc -n kube-system kube-dns`
3. **Test from debug pod:**

```
kubectl run debug --image=busybox --rm -it -- sh
nslookup kubernetes.default.svc.cluster.local
```

4. **Examine CoreDNS logs:** `kubectl logs -n kube-system -l k8s-app=kube-dns`
5. **Check network policies:** Verify DNS traffic (port 53) isn't blocked
6. **Validate kubelet DNS config:** Check `/etc/resolv.conf` in pods

Question 2: What are the critical components involved in Kubernetes DNS resolution?

Answer:

- **CoreDNS pods:** Handle DNS queries and service discovery
- **kube-dns service:** ClusterIP service exposing CoreDNS (usually 10.96.0.10)
- **CoreDNS ConfigMap:** Contains Corefile with DNS configuration
- **kubelet configuration:** Points pod DNS to cluster DNS service
- **CNI plugin:** Ensures DNS traffic routing between pods and CoreDNS
- **Service/Endpoint controllers:** Maintain DNS records for services
- **Search domains:** Allow short name resolution (e.g., `my-service` → `my-service.default.svc.cluster.local`)

Question 3: If you suspect a configuration issue, where would you look for DNS configuration problems?

Answer:

- **CoreDNS ConfigMap:** `kubectl get configmap coredns -n kube-system -o yaml`
- **Check for syntax errors:** Invalid directives, missing plugins, wrong formatting
- **Verify upstream DNS:** Ensure external DNS servers are configured correctly
- **Validate rewrite rules:** Check if custom rewrite rules have correct syntax
- **Plugin configuration:** Verify kubernetes plugin configuration matches cluster setup
- **Zone configuration:** Ensure cluster.local zone is properly configured

Question 4: How would you test DNS resolution from within a pod to isolate the issue?

Answer:

```
# Create debug pod
kubectl run dnstest --image=busybox --rm -it -- sh

# Test internal service resolution
nslookup kubernetes.default.svc.cluster.local
nslookup my-service.my-namespace.svc.cluster.local

# Test external resolution
```

```
nslookup google.com

# Check DNS configuration in pod
cat /etc/resolv.conf

# Test specific DNS server
nslookup kubernetes.default.svc.cluster.local 10.96.0.10

# Use dig for detailed analysis
dig @10.96.0.10 kubernetes.default.svc.cluster.local
```

Question 5: What backup strategies would you recommend for DNS configurations?

Answer:

- **Version control:** Store Corefile configurations in Git with proper versioning
- **ConfigMap backups:** Regular automated backups of CoreDNS ConfigMap
- **Validation pipeline:** Implement CoreDNS configuration validation before deployment
- **Staged deployment:** Test DNS changes in development/staging environments first
- **Rollback procedures:** Document steps to quickly revert to working configuration
- **Monitoring:** Set up alerts for DNS resolution failures and response times

---

Scenario 3: Network Connectivity Crisis

Situation: After adding new worker nodes to your on-premises cluster, you notice that some pods cannot communicate with the control plane, and kubectl commands from certain nodes are failing. Pod-to-pod communication is also intermittently broken.

Question 1: What networking components would you investigate when facing cross-node communication issues?

Answer:

- **CNI plugin configuration:** Verify consistency across all nodes
- **Pod CIDR allocation:** Check if new nodes have correct CIDR ranges
- **Network routes:** Validate routing tables on all nodes
- **iptables rules:** Check for conflicts or missing rules
- **Network interfaces:** Verify CNI-created interfaces are functioning
- **Bridge configuration:** Ensure network bridges are properly configured
- **Security groups/firewalls:** Check if traffic is being blocked at network level

Question 2: How would you verify that the CNI configuration is consistent across all nodes?

Answer:

```
# Check CNI configuration files on each node
ls /etc/cni/net.d/
cat /etc/cni/net.d/10-flannel.conflist
```

```
# Verify kubelet CNI configuration
systemctl status kubelet
grep -i cni /var/lib/kubelet/config.yaml

# Check pod CIDR allocation
kubectl get nodes -o custom-columns=NAME:.metadata.name,PODCIDR:.spec.podCIDR

# Verify CNI plugin binary
ls /opt/cni/bin/
/opt/cni/bin/bridge --version

# Check CNI network namespace setup
ip netns list
```

Question 3: What role does CIDR allocation play in this scenario, and how would you validate it?

Answer: Role of CIDR allocation:

- Each node gets a unique subnet from the cluster's pod CIDR range
- Prevents IP address conflicts between pods on different nodes
- Enables proper routing between pod networks
- Required for CNI plugins to configure networking correctly

Validation steps:

```
# Check cluster CIDR configuration
kubectl cluster-info dump | grep -i cidr
kubectl get nodes -o yaml | grep podCIDR

# Verify no CIDR overlaps
kubectl get nodes -o custom-columns=NAME:.metadata.name,PODCIDR:.spec.podCIDR

# Check kubelet configuration
grep pod-cidr /var/lib/kubelet/config.yaml

# Validate CNI CIDR matches kubelet
cat /etc/cni/net.d/*.conf | grep -i cidr
```

Question 4: If you suspect IP routing problems, what diagnostic commands would you use?

Answer:

```
# Check routing tables
ip route show
route -n

# Verify pod-to-pod connectivity
```

```
kubectl exec -it pod1 -- ping <pod2-ip>

# Trace network path
kubectl exec -it pod1 -- traceroute <destination-ip>

# Check iptables rules
iptables -L -n -v
iptables -t nat -L -n -v

# Verify network interfaces
ip addr show
ip link show

# Check for dropped packets
netstat -i
cat /proc/net/dev

# Test connectivity to control plane
kubectl exec -it pod1 -- telnet kubernetes.default.svc.cluster.local 443
```

Question 5: How would you prevent similar networking issues when scaling the cluster?

Answer:

- **Automated validation:** Create scripts to validate network configuration before adding nodes
- **CIDR management:** Use admission controllers to enforce CIDR allocation policies
- **Configuration management:** Use tools like Ansible/Terraform for consistent node setup
- **Network testing:** Implement automated network connectivity tests for new nodes
- **Documentation:** Maintain clear procedures for node addition with network validation steps
- **Monitoring:** Set up alerts for network connectivity issues and routing problems

---

Scenario 4: Service Accessibility Problems

Situation: Your applications are deployed successfully, but clients cannot reach certain services. Some services work intermittently, while others are completely unreachable. The pods backing these services appear healthy and running.

Question 1: What layers of the Kubernetes networking stack would you examine?

Answer:

1. **Service layer:** Check service configuration and endpoints
2. **kube-proxy layer:** Verify proxy rules and load balancing
3. **iptables/IPVS layer:** Check NAT rules and traffic routing
4. **CNI layer:** Verify pod networking and connectivity
5. **Node networking layer:** Check host networking and firewall rules
6. **External networking:** Load balancers, ingress controllers, network policies

Question 2: How would you differentiate between service configuration issues and underlying network problems?

Answer:

```
# Test direct pod connectivity (bypasses service)
kubectl exec -it client-pod -- curl <pod-ip>:8080

# Test service connectivity
kubectl exec -it client-pod -- curl <service-name>:80

# Check service configuration
kubectl describe service <service-name>
kubectl get endpoints <service-name>

# Verify service DNS resolution
kubectl exec -it client-pod -- nslookup <service-name>

# Check if service has healthy endpoints
kubectl get endpoints <service-name> -o yaml

# Test from different namespaces
kubectl exec -it pod-in-different-ns -- curl <service-name>.
<namespace>.svc.cluster.local
```

Question 3: If you discovered conflicting network rules, how would you identify and resolve them?

Answer: Identification:

```
# Check iptables rules for conflicts
iptables-save | grep -A10 -B10 KUBE-SERVICES
iptables -L -n -v --line-numbers

# Look for custom rules interfering with kube-proxy
iptables -t nat -L PREROUTING -n --line-numbers
iptables -t nat -L OUTPUT -n --line-numbers

# Check for rule precedence issues
iptables -t nat -L KUBE-SERVICES -n --line-numbers
```

Resolution:

```
# Backup current rules
iptables-save > /tmp/iptables-backup.txt

# Remove conflicting custom rules
iptables -t nat -D PREROUTING <rule-number>
```

```
# Restart kube-proxy to regenerate rules
systemctl restart kube-proxy

# Or delete kube-proxy pod if running as DaemonSet
kubectl delete pod -n kube-system -l k8s-app=kube-proxy
```

Question 4: What tools would you use to trace network packets and identify where they're being dropped?

Answer:

```
# Use tcpdump to capture packets
tcpdump -i any -nn port 80 and host <service-ip>

# Trace packets with specific filters
tcpdump -i cni0 -nn 'dst <pod-ip> and port 8080'

# Use netstat to check listening ports
netstat -tlpn | grep :80

# Check connection tracking
conntrack -L -p tcp --dport 80

# Use ss for socket statistics
ss -tlpn | grep :80

# Monitor iptables packet counters
watch -n1 'iptables -L -n -v | grep <service-ip>'

# Use kubectl port-forward for testing
kubectl port-forward svc/<service-name> 8080:80
```

Question 5: How would you establish networking guidelines to prevent conflicts?

Answer:

- **Documentation:** Create clear guidelines for custom iptables rules and network configuration
- **Separation:** Use dedicated chains for custom rules, avoid modifying KUBE-* chains
- **Validation:** Implement pre-deployment network connectivity tests
- **Monitoring:** Set up alerts for service connectivity failures
- **Network policies:** Use Kubernetes NetworkPolicies instead of manual iptables rules
- **Review process:** Require peer review for any network configuration changes
- **Testing:** Maintain automated tests for critical service connectivity paths

---

Scenario 5: Node Management Challenges

Situation: You're trying to add new nodes to your cluster, but they remain in a "NotReady" state indefinitely. The join process appears to hang, and you notice a large number of certificate-related warnings in the logs.

Question 1: What are the key steps in the node joining process, and where might this process fail?

Answer: Node joining process:

1. **Bootstrap token validation:** Node authenticates with cluster using bootstrap token
2. **CSR generation:** kubelet generates Certificate Signing Request
3. **CSR approval:** CSR must be approved (manually or automatically)
4. **Certificate issuance:** Node receives signed certificate
5. **kubelet startup:** kubelet starts with proper certificates
6. **Node registration:** Node registers with API server
7. **CNI setup:** Network plugin configures pod networking

Common failure points:

- Invalid or expired bootstrap tokens
- CSR approval controller not running
- Network connectivity issues
- Incorrect cluster CA certificate
- kubelet configuration errors
- CNI plugin failures

Question 2: How would you investigate certificate-related issues preventing node registration?

Answer:

```
# Check pending CSRs
kubectl get csr

# Examine specific CSR details
kubectl describe csr <csr-name>

# Check kubelet logs on the node
journalctl -u kubelet -f

# Verify bootstrap token
kubectl get secrets -n kube-system | grep bootstrap-token

# Check if CSR approver is running
kubectl get pods -n kube-system | grep controller-manager

# Verify cluster CA certificate
openssl x509 -in /etc/kubernetes/pki/ca.crt -text -noout

# Check kubelet configuration
cat /var/lib/kubelet/config.yaml
```

Question 3: What role do Certificate Signing Requests (CSRs) play in node management?

Answer: CSR functions:

- **Authentication:** Establishes trust between kubelet and API server
- **Secure communication:** Enables encrypted communication channels
- **Identity verification:** Proves node identity to cluster
- **Authorization:** Allows kubelet to perform necessary operations

CSR lifecycle:

1. kubelet generates private key and CSR
2. CSR submitted to API server
3. CSR approval (automatic via controller or manual)
4. Certificate issued and stored by kubelet
5. Certificate used for ongoing authentication

CSR types:

- `kubernetes.io/kube-apiserver-client-kubelet` - kubelet client certificate
- `kubernetes.io/kubelet-serving` - kubelet serving certificate

Question 4: If you found a backlog of unapproved certificates, how would you resolve this systematically?

Answer:

```
# List all pending CSRs
kubectl get csr --sort-by=.metadata.creationTimestamp

# Check if auto-approval is enabled
kubectl get clusterrolebinding | grep csrapproving

# Verify controller-manager CSR flags
kubectl describe pod -n kube-system kube-controller-manager-*

# Approve legitimate CSRs individually
kubectl certificate approve <csr-name>

# Batch approve kubelet client CSRs (carefully!)
kubectl get csr -o name | xargs kubectl certificate approve

# Clean up old/rejected CSRs
kubectl get csr | grep Denied | awk '{print $1}' | xargs kubectl delete csr

# Re-enable auto-approval if disabled
# Check controller-manager flags: --cluster-signing-cert-file, --cluster-signing-
key-file
```

Question 5: What monitoring would you implement to catch certificate management issues early?

Answer:

- **CSR queue monitoring:** Alert on pending CSR count > threshold
- **Certificate expiry alerts:** Monitor certificate expiration dates
- **Node readiness tracking:** Alert when nodes remain NotReady for extended periods
- **kubelet health checks:** Monitor kubelet startup and authentication errors
- **Bootstrap token lifecycle:** Track token creation and expiration
- **Controller manager health:** Ensure CSR approval controller is running
- **Certificate rotation monitoring:** Track automatic certificate renewal

---

Scenario 6: Upgrade Complications

Situation: During a cluster upgrade, the process has stalled. The control plane components are not starting correctly, and you cannot access the cluster through kubectl. The upgrade was progressing normally until it reached the etcd component.

Question 1: What would be your immediate steps to assess the situation during a failed upgrade?

Answer:

1. **Check static pod status:**

```
crictl ps -a | grep etcd
crictl logs <etcd-container-id>
```

2. **Examine static pod manifests:**

```
ls /etc/kubernetes/manifests/
cat /etc/kubernetes/manifests/etcd.yaml
```

3. **Check kubelet status:**

```
systemctl status kubelet
journalctl -u kubelet -f
```

4. **Verify etcd data integrity:**

```
etcdctl endpoint health
etcdctl endpoint status
```

5. **Check available backups:**

```
ls /var/lib/etcd/backup/
```

Question 2: How would you investigate static pod startup failures?

Answer:

```
# Check static pod manifests for syntax errors
yamllint /etc/kubernetes/manifests/*.yaml

# Verify volume mounts exist
ls -la /var/lib/etcd
ls -la /etc/kubernetes/pki/etcd/

# Check container runtime logs
crictl logs <container-id>

# Verify image availability
crictl images | grep etcd

# Check pod events (if API server accessible)
kubectl describe pod etcd-<node-name> -n kube-system

# Validate manifest against schema
kubectl apply --dry-run=client -f /etc/kubernetes/manifests/etcd.yaml

# Check for permission issues
ls -la /etc/kubernetes/manifests/
```

Question 3: What are the critical files and configurations that could prevent control plane components from starting?

Answer: Critical files:

- `/etc/kubernetes/manifests/` - Static pod manifests
- `/etc/kubernetes/pki/` - Certificate files
- `/var/lib/etcd/` - etcd data directory
- `/etc/kubernetes/kubelet.conf` - kubelet configuration
- `/etc/kubernetes/admin.conf` - kubectl configuration

Common configuration issues:

- **Invalid volume mounts:** Incorrect paths in static pod manifests
- **Certificate problems:** Missing or corrupted certificate files
- **Version mismatches:** Incompatible component versions
- **Resource constraints:** Insufficient memory/CPU requests

- **Network configuration:** Incorrect bind addresses or ports
- **Feature gates:** Deprecated or invalid feature flags

Question 4: How would you safely recover from a partially completed upgrade?

Answer:

1. **Create backup of current state:**

```
cp -r /etc/kubernetes /etc/kubernetes.backup
cp -r /var/lib/etcd /var/lib/etcd.backup
```

2. **Attempt component restart:**

```
systemctl restart kubelet
```

3. **Restore from backup if necessary:**

```
# Stop etcd
mv /etc/kubernetes/manifests/etcd.yaml /tmp/

# Restore etcd data
rm -rf /var/lib/etcd/member
etcdctl snapshot restore /path/to/backup.db --data-dir /var/lib/etcd

# Restart etcd
mv /tmp/etcd.yaml /etc/kubernetes/manifests/
```

4. **Rollback upgrade:**

```
kubeadm upgrade node --kubelet-version=v1.x.x
```

Question 5: What pre-upgrade validation steps would you recommend for future upgrades?

Answer:

- **Backup verification:** Ensure recent etcd and configuration backups exist
- **Health checks:** Verify all components are healthy before starting
- **Version compatibility:** Check upgrade paths and component compatibility
- **Resource validation:** Ensure sufficient resources for upgrade process
- **Network connectivity:** Verify all nodes can communicate with control plane
- **Testing in staging:** Perform upgrade in non-production environment first
- **Rollback plan:** Document and test rollback procedures
- **Maintenance window:** Schedule adequate time for upgrade and potential rollback

Scenario 7: Resource Exhaustion Crisis

Situation: Your cluster nodes are experiencing disk space issues, with /var/log directories reaching 100% capacity. Some nodes have become unresponsive, and new pods are failing to schedule.

Question 1: How would you quickly identify which components are consuming excessive disk space?

Answer:

```
# Check disk usage by directory
du -sh /var/log/* | sort -hr
du -sh /var/lib/docker/* | sort -hr
du -sh /var/lib/containerd/* | sort -hr

# Find largest log files
find /var/log -type f -exec ls -lh {} \; | sort -k5 -hr | head -20

# Check container logs specifically
du -sh /var/log/containers/* | sort -hr | head -10
du -sh /var/log/pods/* | sort -hr | head -10

# Identify specific pods with large logs
kubectl get pods --all-namespaces | while read ns pod rest; do
  if [[ "$ns" != "NAMESPACE" ]]; then
    size=$(kubectl logs $pod -n $ns --tail=1 2>/dev/null | wc -c)
    echo "$ns/$pod: $size bytes"
  fi
done | sort -k2 -nr | head -10
```

Question 2: What immediate actions would you take to restore cluster functionality?

Answer:

1. **Emergency log cleanup:**

```
# Truncate largest log files (don't delete, just truncate)
> /var/log/containers/large-log-file.log

# Clean up old rotated logs
find /var/log -name "*.log.*" -mtime +7 -delete

# Clean up journal logs
journalctl --vacuum-time=1d
```

2. **Restart container runtime:**

```
systemctl restart containerd
# or
systemctl restart docker
```

3. **Force pod eviction from unhealthy nodes:**

```
kubectl cordon <node-name>
kubectl drain <node-name> --ignore-daemonsets --force
```

4. **Scale down problematic pods:**

```
kubectl scale deployment <deployment-name> --replicas=0
```

Question 3: How would you implement log rotation and retention policies?

Answer: Container runtime configuration:

```
# /etc/docker/daemon.json
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}

# /etc/containerd/config.toml
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
  max_container_log_line_size = 16384
```

Logrotate configuration:

```
# /etc/logrotate.d/containers
/var/log/containers/*.log {
    daily
    missingok
    rotate 7
    compress
    delaycompress
    copytruncate
    create 0644 root root
}
```

Pod-level log configuration:

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp
    env:
    - name: LOG_LEVEL
      value: "WARN"  # Reduce log verbosity
```

Question 4: What monitoring would you establish to prevent future disk space issues?

Answer: Prometheus metrics to monitor:

- `node_filesystem_avail_bytes` - Available disk space
- `node_filesystem_size_bytes` - Total disk space
- `container_fs_usage_bytes` - Container filesystem usage

Alert rules:

```
groups:
- name: disk-space
  rules:
  - alert: NodeDiskSpaceHigh
    expr: (node_filesystem_avail_bytes / node_filesystem_size_bytes) < 0.1
    for: 5m
    annotations:
      summary: "Node {{ $labels.instance }} disk space is {{ $value }}% full"

  - alert: ContainerLogSizeHigh
    expr: 
container_fs_usage_bytes{container_label_io_kubernetes_container_name!=""} > 
1000000000
    annotations:
      summary: "Container {{ $labels.container_label_io_kubernetes_container_name 
}} logs are using {{ $value }} bytes"
```

Question 5: How would you balance log retention requirements with resource constraints?

Answer:

- **Centralized logging:** Use tools like Fluentd/Fluent Bit to ship logs to external systems
- **Log sampling:** Implement sampling for debug logs while retaining error logs
- **Tiered storage:** Keep recent logs locally, archive older logs to object storage
- **Application-level controls:** Implement configurable log levels and structured logging
- **Resource quotas:** Set limits on log volume per namespace/application

- **Regular cleanup:** Automated scripts to clean up old logs and unused images
- **Monitoring integration:** Alert before disk space becomes critical

---

Scenario 8: Maintenance Window Complications

Situation: You need to perform maintenance on a worker node, but the drain operation has been running for over an hour without completing. Several pods appear to be stuck and won't evict despite repeated attempts.

Question 1: What mechanisms in Kubernetes control pod eviction during node maintenance?

Answer: Key mechanisms:

- **PodDisruptionBudgets (PDB):** Define minimum available replicas during disruptions
- **Graceful termination:** Pods receive SIGTERM and have grace period to shut down
- **finalizers:** Custom logic that can prevent pod deletion
- **Node conditions:** Taints and tolerations affect pod scheduling and eviction
- **Resource constraints:** QoS classes influence eviction priority

Eviction process:

1. Node is cordoned (no new pods scheduled)
2. Pods are selected for eviction
3. PDB constraints are checked
4. SIGTERM sent to pod containers
5. Grace period countdown begins
6. SIGKILL sent if grace period expires

Question 2: How would you identify which pods are preventing the drain operation from completing?

Answer:

```
# Check drain status and stuck pods
kubectl get pods --all-namespaces --field-selector spec.nodeName=<node-name>

# Look for pods in Terminating state
kubectl get pods --all-namespaces | grep Terminating

# Check PodDisruptionBudgets
kubectl get pdb --all-namespaces
kubectl describe pdb <pdb-name> -n <namespace>

# Examine specific pod details
kubectl describe pod <stuck-pod> -n <namespace>

# Check for finalizers on stuck pods
kubectl get pod <stuck-pod> -n <namespace> -o yaml | grep finalizers

# View events related to eviction
kubectl get events --field-selector involvedObject.name=<pod-name>
```

Question 3: What configuration conflicts could cause eviction deadlocks?

Answer: Common deadlock scenarios:

- **PDB misconfiguration:** `minAvailable: 2` with only 2 replicas running
- **Single replica with PDB:** Deployment has 1 replica but PDB requires 1 available
- **Stuck finalizers:** Custom controllers not responding to pod deletion
- **Resource constraints:** No other nodes have capacity for evicted pods
- **Affinity rules:** Pod anti-affinity preventing scheduling on other nodes
- **Persistent volumes:** Pods can't be rescheduled due to volume constraints

Example problematic configurations:

```yaml
# Problematic PDB
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: problematic-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: myapp
---
# With only 2 replicas - creates deadlock
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 2  # Same as minAvailable!
```

Question 4: How would you safely force completion of the maintenance while minimizing service disruption?

Answer:

1. **Temporary PDB adjustment:**

   ```
   # Scale up replicas first
   kubectl scale deployment <app> --replicas=3

   # Or temporarily modify PDB
   kubectl patch pdb <pdb-name> --type='merge' -p='{"spec":{"minAvailable":1}}'
   ```

2. **Force delete stuck pods (carefully):**

```
# Remove finalizers if safe
kubectl patch pod <pod-name> -p '{"metadata":{"finalizers":null}}'

# Force delete as last resort
kubectl delete pod <pod-name> --force --grace-period=0
```

3. **Use --force flag with drain:**

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data --force
```

4. **Manual eviction with verification:**

```
# Evict pods one by one
kubectl delete pod <pod-name> --grace-period=30
```

Question 5: What policies would you establish to prevent similar maintenance issues?

Answer:

- **PDB validation:** Ensure PDBs allow at least one pod to be disrupted
- **Replica requirements:** Mandate minimum 3 replicas for applications with PDBs
- **Testing procedures:** Test drain operations in staging environments
- **Maintenance automation:** Use tools like kops or cluster-autoscaler for managed maintenance
- **Communication protocols:** Coordinate with application teams before maintenance
- **Monitoring setup:** Alert on stuck evictions or long-running drain operations
- **Documentation:** Maintain runbooks for handling different types of stuck pods

---

Scenario 9: Control Plane Instability

Situation: One of your control plane components keeps crashing immediately after startup. The cluster is partially functional, but certain operations are failing. Recent configuration changes were made to enhance security.

Question 1: How would you systematically diagnose control plane component crashes?

Answer:

1. **Identify which component is crashing:**

```
kubectl get pods -n kube-system
crictl ps -a | grep -E "(apiserver|controller|scheduler)"
```

2. **Examine crash logs:**

```
kubectl logs -n kube-system kube-controller-manager-<node> --previous
crictl logs <container-id>
journalctl -u kubelet | grep -i error
```

3. **Check static pod manifests:**

```
ls /etc/kubernetes/manifests/
cat /etc/kubernetes/manifests/kube-controller-manager.yaml
```

4. **Validate configuration syntax:**

```
yamllint /etc/kubernetes/manifests/*.yaml
kubectl apply --dry-run=client -f /etc/kubernetes/manifests/
```

Question 2: What configuration areas would you examine when components fail to start?

Answer: Common configuration areas:

- **Command line flags:** Invalid or deprecated flags
- **Admission controllers:** Unknown or removed admission plugins
- **Feature gates:** Deprecated or invalid feature gates
- **Certificate paths:** Incorrect or missing certificate file paths
- **API versions:** Deprecated API versions in configuration
- **Resource requests:** Insufficient memory/CPU allocation
- **Volume mounts:** Missing or incorrect volume mount paths

Specific checks:

```
# Check for deprecated flags
grep -E "(enable-admission-plugins|feature-gates)"
/etc/kubernetes/manifests/*.yaml

# Validate certificate paths
ls -la /etc/kubernetes/pki/

# Check resource allocations
grep -A5 resources /etc/kubernetes/manifests/*.yaml
```

Question 3: How would you identify deprecated or incompatible configurations after an upgrade?

Answer:

1. **Review Kubernetes changelog:**

- Check deprecated APIs for your version
- Identify removed feature gates
- Review admission controller changes

2. **Use kubectl convert:**

```
kubectl convert -f old-manifest.yaml --output-version=apps/v1
```

3. **API deprecation tools:**

```
# Use kubent (Kubernetes No Trouble)
kubent --cluster

# Check API versions
kubectl api-versions | grep -v v1
```

4. **Validate configurations:**

```
# Test manifest application
kubectl apply --dry-run=server -f /etc/kubernetes/manifests/

# Check for warnings
kubectl apply --dry-run=client -f manifest.yaml
```

Question 4: What would be your approach to safely testing configuration changes before applying them?

Answer:

1. **Staging environment testing:**
   - Test all changes in non-production cluster first
   - Use identical Kubernetes versions and configurations

2. **Backup and validation:**

```
# Backup current configurations
cp -r /etc/kubernetes /etc/kubernetes.backup

# Validate new configurations
yamllint /etc/kubernetes/manifests/*.yaml
kubectl apply --dry-run=server -f new-manifest.yaml
```

3. **Progressive rollout:**

- Apply changes to one control plane node first
- Verify component health before proceeding
- Use rolling updates for multi-master clusters

4. **Monitoring and rollback:**

```
# Monitor component startup
watch kubectl get pods -n kube-system

# Quick rollback if needed
mv /etc/kubernetes.backup/* /etc/kubernetes/
```

Question 5: How would you implement configuration validation in your deployment pipeline?

Answer:

- **CI/CD integration:** Add Kubernetes manifest validation to pipeline
- **Schema validation:** Use tools like kubeval or kustomize for validation
- **Policy enforcement:** Implement OPA/Gatekeeper for configuration policies
- **Automated testing:** Include cluster component health checks in deployment tests
- **Version compatibility:** Automate checking of API version compatibility
- **Change approval:** Require peer review for control plane configuration changes
- **Documentation:** Maintain change logs and configuration baselines

---

Scenario 10: Disaster Recovery Situation

Situation: Your cluster has experienced a catastrophic failure, and you need to restore from backups. However, after restoring the etcd snapshot, many applications are failing with missing volume and secret errors.

Question 1: What components should be included in a comprehensive Kubernetes backup strategy?

Answer: Essential backup components:

- **etcd data:** Core cluster state and configuration
- **Persistent volumes:** Application data and stateful workloads
- **Secrets and ConfigMaps:** Application configuration and credentials
- **Custom resources:** CRDs and custom resource instances
- **RBAC configurations:** Service accounts, roles, and bindings
- **Network policies:** Security and traffic rules
- **Certificates:** PKI infrastructure and TLS certificates

Backup tools and methods:

- **Velero:** Comprehensive backup including volumes and resources
- **etcd snapshots:** `etcdctl snapshot save`
- **Volume snapshots:** Cloud provider native snapshot tools

- **GitOps:** Configuration stored in version control
- **External secrets:** Backup external secret management systems

Question 2: How would you verify the completeness of your backup before a restore operation?

Answer:

```
# Verify etcd snapshot integrity
etcdctl snapshot status backup.db --write-out=table

# List backed up resources
velero backup describe <backup-name> --details

# Check volume snapshots
kubectl get volumesnapshot
aws ec2 describe-snapshots --owner-ids <account-id>

# Verify backup metadata
velero backup logs <backup-name>

# Test restore in isolated environment
velero restore create test-restore --from-backup <backup-name>

# Compare resource counts
kubectl get all,secrets,pv,pvc --all-namespaces | wc -l
```

Question 3: What dependencies exist between different Kubernetes resources that could cause restore issues?

Answer: Resource dependencies:

- **Pods → Secrets/ConfigMaps:** Pods cannot start without referenced secrets
- **Pods → PVCs:** Stateful pods need persistent volume claims
- **PVCs → PVs:** Claims must bind to available persistent volumes
- **Services → Endpoints:** Services need backing pods for endpoints
- **Ingress → Services:** Ingress rules reference services
- **RBAC → ServiceAccounts:** Pods need service accounts for API access
- **NetworkPolicies → Pods:** Policies reference pod selectors

Common dependency issues:

- PVCs restored but underlying storage volumes missing
- Secrets restored but encryption keys changed
- Service accounts restored but RBAC bindings missing
- Custom resources restored but CRDs not installed

Question 4: How would you test your disaster recovery procedures without impacting production?

Answer:

1. **Isolated test environment:**

```
# Create separate cluster for testing
kind create cluster --name disaster-test

# Restore backup to test cluster
velero restore create test-restore --from-backup prod-backup
```

2. **Validation testing:**

```
# Test application functionality
kubectl run test-pod --image=busybox --rm -it -- sh

# Verify data integrity
kubectl exec -it app-pod -- cat /data/important-file.txt

# Check all resources are present
kubectl get all,secrets,pv,pvc --all-namespaces
```

3. **Performance testing:**
   - Run load tests against restored applications
   - Verify database integrity and consistency
   - Test inter-service communication

4. **Documentation and automation:**
   - Script the entire restore process
   - Document any manual steps required
   - Measure restore time objectives (RTO)

Question 5: What backup validation and testing schedule would you recommend?

Answer: Regular testing schedule:

- **Daily:** Automated backup verification and integrity checks
- **Weekly:** Partial restore testing of critical applications
- **Monthly:** Full disaster recovery drill in isolated environment
- **Quarterly:** Complete end-to-end recovery testing with stakeholders

Automated validation:

```
# Daily backup verification script
#!/bin/bash
etcdctl snapshot status /backup/etcd-$(date +%Y%m%d).db
velero backup get --completed
```

```
# Weekly restore test
velero restore create weekly-test-$(date +%Y%m%d) --from-backup latest-backup
```

Metrics to track:

- Backup completion time and size
- Restore time objective (RTO)
- Recovery point objective (RPO)
- Success rate of backup operations
- Time to complete full recovery

Scenario 11: Enterprise Network Integration

Situation: Your organization uses corporate proxies for all outbound internet traffic. After deploying a new cluster in this environment, nodes cannot pull container images, and some internal service communications are failing.

Question 1: How do corporate proxies affect different aspects of Kubernetes operations?

Answer: Affected operations:

- **Image pulls:** Container runtime needs proxy for registry access
- **Package downloads:** Node setup and kubelet installation
- **External services:** Applications accessing external APIs
- **Cluster networking:** DNS resolution and external connectivity
- **Monitoring/logging:** Metrics and logs shipping to external systems
- **Certificate management:** OCSP validation and certificate downloads

Components requiring proxy configuration:

- kubelet (for image pulls and API communication)
- Container runtime (Docker/containerd)
- kube-proxy (for external service access)
- CNI plugins (for external registry pulls)
- Applications (for external API calls)

Question 2: What configuration is required to make Kubernetes work correctly in a proxied environment?

Answer: kubelet configuration:

```
# /etc/systemd/system/kubelet.service.d/10-proxy.conf
[Service]
Environment="HTTP_PROXY=http://proxy.company.com:8080"
Environment="HTTPS_PROXY=http://proxy.company.com:8080"
Environment="NO_PROXY=10.0.0.0/8,172.16.0.0/12,192.168.0.0/16,localhost,127.0.0.1,
kubernetes.default.svc,cluster.local"
```

Container runtime configuration:

```
# /etc/systemd/system/docker.service.d/http-proxy.conf
[Service]
Environment="HTTP_PROXY=http://proxy.company.com:8080"
Environment="HTTPS_PROXY=http://proxy.company.com:8080"
Environment="NO_PROXY=localhost,127.0.0.1,docker-registry.local"

# For containerd: /etc/systemd/system/containerd.service.d/http-proxy.conf
```

Pod-level proxy configuration:

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    env:
    - name: HTTP_PROXY
      value: "http://proxy.company.com:8080"
    - name: HTTPS_PROXY
      value: "http://proxy.company.com:8080"
    - name: NO_PROXY
      value: "kubernetes.default.svc,cluster.local"
```

Question 3: How would you troubleshoot image pull failures in a proxy environment?

Answer:

```
# Test proxy connectivity from node
curl -x http://proxy.company.com:8080 https://registry-1.docker.io

# Check container runtime proxy configuration
systemctl show docker | grep -i proxy
systemctl show containerd | grep -i proxy

# Test image pull manually
docker pull --debug ubuntu:latest
crictl pull ubuntu:latest

# Check kubelet logs for proxy-related errors
journalctl -u kubelet | grep -i proxy

# Verify NO_PROXY settings
echo $NO_PROXY
```

```
# Test DNS resolution through proxy
nslookup registry-1.docker.io
```

Question 4: What internal traffic should bypass the proxy, and how would you configure this?

Answer: Traffic that should bypass proxy:

- Cluster IP ranges (service CIDR)
- Pod IP ranges (pod CIDR)
- Node IP ranges
- Cluster DNS (kubernetes.default.svc)
- Local services (.cluster.local)
- Load balancer IPs
- Localhost/loopback addresses

NO_PROXY configuration:

```
NO_PROXY="10.0.0.0/8,172.16.0.0/12,192.168.0.0/16,localhost,127.0.0.1,.local,.clus
ter.local,kubernetes.default.svc"

# For specific cluster
NO_PROXY="10.96.0.0/12,10.244.0.0/16,kubernetes.default.svc,cluster.local,localhos
t,127.0.0.1"
```

Validation:

```
# Test internal service access
kubectl run test --image=busybox --rm -it -- sh
nslookup kubernetes.default.svc.cluster.local

# Verify external access goes through proxy
curl -v https://google.com  # Should show proxy in output
```

Question 5: How would you validate proxy configuration across all cluster components?

Answer: Automated validation script:

```
#!/bin/bash
# check-proxy-config.sh

echo "Checking kubelet proxy configuration..."
systemctl show kubelet | grep -i proxy

echo "Checking container runtime proxy..."
systemctl show docker | grep -i proxy
```

```
echo "Testing external connectivity..."
curl -x $HTTP_PROXY -I https://registry-1.docker.io

echo "Testing internal connectivity..."
kubectl run proxy-test --image=busybox --rm -it -- nslookup kubernetes.default.svc

echo "Validating NO_PROXY settings..."
echo "NO_PROXY: $NO_PROXY"
```

Monitoring and validation:

- **Regular connectivity tests:** Automated tests for image pulls and external access
- **Proxy health monitoring:** Monitor proxy server availability and response times
- **Configuration drift detection:** Ensure proxy settings remain consistent across nodes
- **Network policy validation:** Verify internal traffic bypasses proxy correctly
- **Documentation maintenance:** Keep proxy configuration documentation updated

---

Follow-up Questions for All Scenarios:

Prevention: What monitoring and alerting would you implement to detect this issue earlier?

Answer:

- **Proactive monitoring:** Set up alerts for early warning indicators
- **Health checks:** Implement comprehensive health monitoring for all components
- **Baseline metrics:** Establish normal operating parameters and alert on deviations
- **Trend analysis:** Monitor metrics over time to predict potential issues
- **Synthetic testing:** Regular automated tests to validate functionality

Documentation: How would you document this incident for future reference?

Answer:

- **Incident timeline:** Detailed chronology of events and actions taken
- **Root cause analysis:** Technical analysis of why the issue occurred
- **Resolution steps:** Step-by-step instructions for fixing similar issues
- **Lessons learned:** Key insights and process improvements identified
- **Prevention measures:** Actions taken to prevent recurrence

Process Improvement: What changes to procedures or automation would prevent recurrence?

Answer:

- **Automation:** Script manual processes prone to human error
- **Validation pipelines:** Add checks to prevent problematic configurations
- **Change management:** Improve review and testing processes for changes
- **Monitoring enhancements:** Fill gaps in observability and alerting
- **Training:** Ensure team has knowledge to handle similar situations

Knowledge Sharing: How would you ensure your team learns from this experience?

Answer:

- **Post-mortem meetings:** Team discussion of incident and lessons learned
- **Documentation sharing:** Make incident reports accessible to all team members
- **Training sessions:** Conduct technical training on the specific issue area
- **Runbook updates:** Improve operational procedures based on lessons learned
- **Cross-training:** Ensure multiple team members can handle similar issues

Tooling: What additional tools or scripts would help with faster diagnosis and resolution?

Answer:

- **Diagnostic scripts:** Automate common troubleshooting steps
- **Monitoring dashboards:** Create focused views for specific problem types
- **Automation tools:** Implement self-healing where appropriate
- **Testing frameworks:** Develop automated tests for critical functionality
- **Integration tools:** Connect monitoring, alerting, and response systems

---