

# CKAD Exam 2025 - Questions and Answers

---

A comprehensive guide containing all 20 practice questions with detailed solutions for the Certified Kubernetes Application Developer (CKAD) exam.

---

## Table of Contents

1. [Question 1: Ingress Creation](#)
  2. [Question 2: Fix Broken Ingress \(404 Error\)](#)
  3. [Question 3: Fix Broken Deployment \(Incorrect Secret\)](#)
  4. [Question 4: NetworkPolicy - Adjust Pod Labels for Communication](#)
  5. [Question 5: ResourceQuota and LimitRange Compliance](#)
  6. [Question 6: CronJob Configuration](#)
  7. [Question 7: RBAC - ServiceAccount, Role, and RoleBinding](#)
  8. [Question 8: Canary Deployment](#)
  9. [Question 9: Multi-Container Sidecar Pod](#)
  10. [Question 10: Fix Deprecated API Version](#)
  11. [Question 11: Modify Deployment Twice, Then Rollback](#)
  12. [Question 12: Job with Failure Policy](#)
  13. [Question 13: Docker Build + OCI Export](#)
  14. [Question 14: SecurityContext - Edit Existing Deployment](#)
  15. [Question 15: Find Existing ServiceAccount and Apply to Deployment](#)
  16. [Question 16: Secret with Multiple Keys](#)
  17. [Question 17: Expose Deployment with NodePort](#)
  18. [Question 18: Fix Deployment - Container Name & Image + Resume Rollout](#)
  19. [Question 19: Service Selector Fix](#)
  20. [Question 20: Pod with Command](#)
- 

## Question 1: Ingress Creation

**Weight:** ~7% | **Domain:** Services & Networking

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

You need to expose an existing web application to external traffic using an Ingress. The Service 'webapp' has already been created and exposes port 8080.

### Tasks

1. Create an Ingress resource named 'ingress-name' in the 'external' namespace that exposes the application to the URL 'external.app.local'
2. Ensure the Ingress routes all traffic from the root path '/' to the backend service 'webapp' on port 8080
3. Use the ingress class 'nginx-exam'

## Namespace

external

## Solution

### Method 1: Imperative Command (FASTEST for exam!)

```
kubectl create ingress ingress-name \
  --rule="external.app.local/*=webapp:8080" \
  --class=nginx-exam \
  -n external
```

### Method 2: Using YAML Manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-name
  namespace: external
spec:
  ingressClassName: nginx-exam
  rules:
  - host: external.app.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: webapp
            port:
              number: 8080
```

## Key Points

- apiVersion: networking.k8s.io/v1 (not v1beta1)
- ingressClassName: nginx-exam (not annotation)
- pathType: Prefix is required for v1
- Backend uses service.name and service.port.number structure

### ⚡ Fastest Exam Approach (< 30 seconds)

```
# ONE COMMAND - using imperative method:
kubectl create ingress ingress-name \
  --rule="external.app.local/*=webapp:8080" \
  --class=nginx-exam \
  -n external

# Verify:
kubectl get ingress -n external
```

**TIP:** The `/*` automatically sets `pathType: Prefix`!

---

## Question 2: Fix Broken Ingress (404 Error)

**Weight:** ~5% | **Domain:** Services & Networking

### Context

```
kubectl config use-context k8s-cluster2
```

### Scenario

An Ingress resource named 'web-ingress' in namespace 'production' is returning 404 errors when users try to access the application.

### Tasks

1. Troubleshoot and fix the Ingress configuration so that requests to 'app.example.com' are properly routed to the backend service 'web-service' on port 80
2. Do NOT create any new resources. Only fix the existing Ingress.

### Namespace

production

### Solution

#### Step 1: Check Current Ingress Configuration

```
kubectl describe ingress web-ingress -n production
```

#### Step 2: Verify Backend Service Exists

```
kubectl get svc web-service -n production  
kubectl get endpoints web-service -n production
```

#### Step 3: Fix the Ingress

##### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit ingress web-ingress -n production
```

Find and fix the following issues:

```
spec:
  rules:
    - host: app.example.com      # FIX typo: app.exmple.com →
    app.example.com
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: web-service   # FIX: web-svc → web-service
              port:
                number: 80       # FIX: 8080 → 80
```

Save and exit (:wq in vim)

### Method B: Using kubectl patch

```
kubectl patch ingress web-ingress -n production --type='json' \
-p='[
  {"op": "replace", "path": "/spec/rules/0/host", "value":
"app.example.com"},
  {"op": "replace", "path":
"/spec/rules/0/http/paths/0/backend/service/name", "value": "web-
service"},
  {"op": "replace", "path":
"/spec/rules/0/http/paths/0/backend/service/port/number", "value": 80}
]'
```

### Common Issues to Check

- Host: app.exmple.com → app.example.com (typo fix)
- Service name: web-svc → web-service
- Port: 8080 → 80

### ⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: Quick inspection (20 sec)
kubectl describe ingress web-ingress -n production
kubectl get svc web-service -n production

# Step 2: Fix with kubectl edit (40 sec)
kubectl edit ingress web-ingress -n production
# Fix: host typo, service name, port number
# Save (:wq)

# Step 3: Verify
kubectl describe ingress web-ingress -n production
```

**TIP:** kubectl edit is FASTEST for multi-field fixes in existing resources!

---

## Question 3: Fix Broken Deployment (Incorrect Secret)

**Weight:** ~5% | **Domain:** Application Deployment

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment named 'backend-deployment' in namespace 'staging' is failing to start. The pods are in CrashLoopBackOff state. The issue is caused by an incorrect Secret reference.

### Tasks

1. Investigate and determine the exact issue causing the pod failures
2. Fix the Deployment so that it references the correct Secret 'db-credentials'
3. Ensure the pods start successfully after the fix

### Namespace

staging

### Solution

#### Step 1: Check Pod Status

```
kubectl get pods -n staging  
kubectl describe pod -l app=backend -n staging | grep -A5 "Events:"
```

#### Step 2: Verify Available Secrets

```
kubectl get secrets -n staging
```

#### Step 3: Fix the Deployment

##### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit deployment backend-deployment -n staging
```

Find the envFrom section and fix the secretRef name:

```
spec:
  template:
    spec:
      containers:
      - name: backend
        envFrom:
        - secretRef:
            name: db-credentials    # CHANGE FROM: db-credentials-old
```

Save and exit (:wq in vim)

### Method B: Using kubectl patch

```
kubectl patch deployment backend-deployment -n staging --type='json' \
-p='[{"op": "replace", "path":
"/spec/template/spec/containers/0/envFrom/0/secretRef/name", "value": "db-
credentials"}]'
```

### Step 4: Verify Fix

```
kubectl rollout status deployment/backend-deployment -n staging
kubectl get pods -n staging
```

### ⚡ Fastest Exam Approach (< 45 seconds)

```
# Step 1: Find the problem (15 sec)
kubectl describe pod -l app=backend -n staging | grep -E
"Secret|Error|Warning"
kubectl get secrets -n staging

# Step 2: Fix with kubectl edit (20 sec)
kubectl edit deployment backend-deployment -n staging
# Find secretRef.name and change to: db-credentials
# Save (:wq)

# Step 3: Verify (10 sec)
kubectl rollout status deployment/backend-deployment -n staging
```

**TIP:** Events section in kubectl describe shows exactly what's wrong!

---

## Question 4: NetworkPolicy - Adjust Pod Labels for Communication

**Weight:** ~7% | **Domain:** Services & Networking

### Context

```
kubectl config use-context k8s-cluster3
```

### Scenario

In namespace 'app-ns', there are 4 NetworkPolicies that control traffic flow between pods (front-pod, api-pod, db-pod). The 'api-pod' needs to receive inbound traffic from 'front-pod' AND send outbound traffic to 'db-pod'.

**Important:** Do NOT create, modify, or delete any existing NetworkPolicies. Make changes ONLY to the pod labels.

### Tasks

1. Examine ALL 4 NetworkPolicies in the namespace to understand their pod selectors and ingress/egress rules
2. WITHOUT modifying any NetworkPolicy, add the appropriate labels to the 'api-pod' so that:
  - 'api-pod' can receive traffic from 'front-pod'
  - 'api-pod' can send traffic to 'db-pod'
3. Verify connectivity is established after adding the labels

### Namespace

app-ns

### Solution

#### Step 1: List NetworkPolicies

```
kubectl get networkpolicy -n app-ns
```

#### Step 2: Examine Each Policy

```
kubectl describe networkpolicy deny-all -n app-ns | grep -A10 "Spec:"
kubectl describe networkpolicy allow-front-to-api -n app-ns | grep -A10
"Spec:"
kubectl describe networkpolicy allow-api-to-db -n app-ns | grep -A10
"Spec:"
```

### Step 3: Check Current Labels

```
kubectl get pod api-pod -n app-ns --show-labels
```

### Step 4: Add Required Label

```
kubectl label pod api-pod -n app-ns tier=api
```

### Key Points

- deny-all blocks everything by default (empty podSelector = all pods)
- allow-front-to-api: pods with tier=api receive traffic from tier=frontend
- allow-api-to-db: pods with tier=database receive traffic from tier=api
- Solution: Add tier=api label to api-pod to enable both flows

### ⚡ Fastest Exam Approach (< 45 seconds)

```
# Step 1: See what label is needed (30 sec)
kubectl get networkpolicy -n app-ns -o yaml | grep -A5 "podSelector"
# Look for the common label in ingress/egress rules!

# Step 2: Add label to pod (15 sec)
kubectl label pod api-pod -n app-ns tier=api
```

**TIP:** NetworkPolicies use AND logic - ONE label can satisfy multiple policies!

---

## Question 5: ResourceQuota and LimitRange Compliance

**Weight:** ~5% | **Domain:** Application Environment, Configuration and Security

### Context

```
kubectl config use-context k8s-cluster2
```

### Scenario

The namespace 'limited-ns' has resource constraints configured through both a ResourceQuota and a LimitRange.

### Tasks

1. Examine the existing ResourceQuota and LimitRange in namespace 'limited-ns'
2. Create a Pod named 'quota-pod' in namespace 'limited-ns' with:
  - Image: nginx
  - Container name: nginx-container
  - Memory requests/limits: HALF of the maximum memory limit (320Mi)
  - CPU requests/limits: HALF of the maximum CPU limit (500m)

### Namespace

limited-ns

### Solution

#### Step 1: Check LimitRange

```
kubectl describe limitrange resource-limits -n limited-ns
```

#### Step 2: Check ResourceQuota

```
kubectl describe resourcequota compute-quota -n limited-ns
```

#### Step 3: Create Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: quota-pod
  namespace: limited-ns
```

```
spec:
  containers:
  - name: nginx-container
    image: nginx
    resources:
      requests:
        cpu: "500m"
        memory: "320Mi"
      limits:
        cpu: "500m"
        memory: "320Mi"
```

## Key Points

- LimitRange max CPU: 1 (1000m) → Half = 500m
- LimitRange max Memory: 640Mi → Half = 320Mi
- Both requests AND limits are required when quota enforces them

## ⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: Check limits (10 sec)
kubectl describe limitrange -n limited-ns

# Step 2: Generate YAML and apply (50 sec)
kubectl run quota-pod --image=nginx -n limited-ns --dry-run=client -o yaml
> pod.yaml
# Edit pod.yaml to add resources (half of max limits)
kubectl apply -f pod.yaml
```

**TIP:** When quota exists, BOTH requests AND limits are required - don't skip either!

---

## Question 6: CronJob Configuration

**Weight:** ~5% | **Domain:** Application Design and Build

### Context

```
kubectl config use-context k8s-cluster1
```

### Tasks

1. Create a CronJob named 'my-cronjob' that runs every 30 minutes with the following specifications:

- Image: busybox
- Command: /bin/sh -c "date; echo Hello"
- completions: 8 (number of times the job must complete successfully)
- startingDeadlineSeconds: 17 (deadline to start if missed)
- activeDeadlineSeconds: 8 (max time for Job to complete)
- successfulJobsHistoryLimit: 3 (number of successful jobs to keep)
- failedJobsHistoryLimit: 1 (number of failed jobs to keep)

**NOTE:** In the actual exam, specific values will be provided in the question

2. Create a Job named 'my-job' from the CronJob to trigger it manually

### Namespace

default

### Solution

#### Step 1: Create CronJob

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/30 * * * *"
  startingDeadlineSeconds: 17
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      completions: 8
      activeDeadlineSeconds: 8
      template:
        spec:
          containers:
            - name: cronjob-container
```

```
image: busybox
command: ["/bin/sh", "-c", "date; echo Hello"]
restartPolicy: OnFailure
```

## Step 2: Create Job from CronJob

```
kubectl create job my-job --from=cronjob/my-cronjob
```

## Key Points

- **CronJob level (spec.):** startingDeadlineSeconds, successfulJobsHistoryLimit, failedJobsHistoryLimit
- **Job level (jobTemplate.spec.):** completions, activeDeadlineSeconds
- Schedule format: `*/30 * * * *` = every 30 minutes

## ⚡ Fastest Exam Approach (< 90 seconds)

```
# Step 1: Generate CronJob YAML (20 sec)
kubectl create cronjob my-cronjob --image=busybox \
  --schedule="*/30 * * * *" --dry-run=client -o yaml > cj.yaml

# Step 2: Edit cj.yaml to add required fields (50 sec)
# CronJob level: startingDeadlineSeconds: 17, history limits
# Job level: completions: 8, activeDeadlineSeconds: 8
# Pod level: command: ["/bin/sh", "-c", "date; echo Hello"]
kubectl apply -f cj.yaml

# Step 3: Create Job from CronJob (20 sec)
kubectl create job my-job --from=cronjob/my-cronjob
```

**KEY:** startingDeadlineSeconds → CronJob level; activeDeadlineSeconds → Job level

---

## Question 7: RBAC - ServiceAccount, Role, and RoleBinding

**Weight:** ~6% | **Domain:** Configuration and Security (RBAC)

### Context

```
kubectl config use-context k8s-cluster2
```

### Scenario

A Deployment named 'secure-app' in namespace 'secure-ns' is failing with "Forbidden" errors when trying to list pods.

### Tasks

1. Create ServiceAccount 'pod-reader-sa' in namespace 'secure-ns'
2. Create Role 'pod-reader-role' that allows get, list, watch on pods
3. Create RoleBinding 'pod-reader-binding' to bind the Role to the ServiceAccount
4. Update Deployment 'secure-app' to use the new ServiceAccount

### Namespace

secure-ns

### Solution

#### Step 1: Create ServiceAccount

```
kubectl create serviceaccount pod-reader-sa -n secure-ns
```

#### Step 2: Create Role

```
kubectl create role pod-reader-role \
  --verb=get,list,watch \
  --resource=pods \
  -n secure-ns
```

#### Step 3: Create RoleBinding

```
kubectl create rolebinding pod-reader-binding \
  --role=pod-reader-role \
  --serviceaccount=secure-ns:pod-reader-sa \
  -n secure-ns
```

## Step 4: Update Deployment

```
kubectl set serviceaccount deployment/secure-app pod-reader-sa -n secure-ns
```

## Step 5: Verify Permissions

```
kubectl auth can-i list pods -n secure-ns --as=system:serviceaccount:secure-ns:pod-reader-sa
```

### ⚡ Fastest Exam Approach (< 60 seconds)

```
# All FOUR commands - memorize them!
kubectl create serviceaccount pod-reader-sa -n secure-ns

kubectl create role pod-reader-role --verb=get,list,watch --resource=pods -n secure-ns

kubectl create rolebinding pod-reader-binding \
  --role=pod-reader-role \
  --serviceaccount=secure-ns:pod-reader-sa -n secure-ns

kubectl set serviceaccount deployment/secure-app pod-reader-sa -n secure-ns
```

**TIP:** ServiceAccount format in rolebinding: namespace:sa-name (with colon!)

---

## Question 8: Canary Deployment

**Weight:** ~8% | **Domain:** Application Deployment

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'web-app' with 5 replicas exists in namespace 'canary-ns', using image 'nginx:1.19'. A Service 'web-service' routes traffic to pods with label 'app=web-app'.

**Important:** There is a 10 pod limit in the namespace.

### Tasks

1. Implement a canary deployment creating 'web-app-canary' with:
  - 20% traffic to canary (1 replica)
  - Image: nginx:1.20
  - Labels: app=web-app, version=canary
2. Scale stable deployment to 4 replicas
3. Final state: stable=4, canary=1 (5 total)

### Namespace

canary-ns

### Solution

#### Step 1: Scale Down Stable

```
kubectl scale deployment web-app -n canary-ns --replicas=4
```

#### Step 2: Create Canary Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app-canary
  namespace: canary-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web-app
```

```
    version: canary
  template:
    metadata:
      labels:
        app: web-app
        version: canary
    spec:
      containers:
      - name: nginx
        image: nginx:1.20
        ports:
        - containerPort: 80
```

### Step 3: Verify

```
kubectl get endpoints web-service -n canary-ns
```

### Key Points

- Total:  $4 + 1 = 5$  pods (within 10 pod limit)
- Traffic split:  $4/5 = 80\%$  stable,  $1/5 = 20\%$  canary
- Both deployments must have `app=web-app` label for service selector

### ⚡ Fastest Exam Approach (< 90 seconds)

```
# Step 1: Scale down stable (10 sec)
kubectl scale deployment web-app -n canary-ns --replicas=4

# Step 2: Get existing deployment YAML as base (20 sec)
kubectl get deployment web-app -n canary-ns -o yaml > canary.yaml

# Step 3: Edit canary.yaml (40 sec)
# Change: name, image, add version=canary label, set replicas=1
kubectl apply -f canary.yaml

# Step 4: Verify service routes to both (20 sec)
kubectl get endpoints web-service -n canary-ns
```

**KEY:** Both deployments MUST have `app=web-app` for service selector!

---

## Question 9: Multi-Container Sidecar Pod

**Weight:** ~7% | **Domain:** Application Design and Build

### Context

```
kubectl config use-context k8s-cluster3
```

### Scenario

Create a pod with two containers that share a volume for log aggregation.

### Tasks

Create Pod 'sidecar-pod' with:

1. Main container:
  - Name: main-app
  - Image: busybox
  - Command: writes logs to /var/log/app.log
  - Mount volume at /var/log
2. Sidecar container:
  - Name: log-sidecar
  - Image: busybox
  - Command: tail -f /var/log/app.log
  - Mount volume at /var/log
3. Shared volume: log-volume (emptyDir)

### Namespace

default

### Solution

```
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-pod
  namespace: default
spec:
  volumes:
    - name: log-volume
      emptyDir: {}
  containers:
    - name: main-app
      image: busybox
      command: ["/bin/sh", "-c"]
      args:
```

```
- while true; do echo $(date) - Log entry >> /var/log/app.log; sleep
5; done
volumeMounts:
- name: log-volume
  mountPath: /var/log
- name: log-sidecar
  image: busybox
  command: ["/bin/sh", "-c"]
  args:
  - tail -f /var/log/app.log
volumeMounts:
- name: log-volume
  mountPath: /var/log
```

## Key Points

- emptyDir volume is created when pod starts, deleted when pod terminates
- volumeMounts.name must match volumes.name
- Use -c to specify container for logs/exec

## ⚡ Fastest Exam Approach (< 90 seconds)

```
# Step 1: Generate base pod (10 sec)
kubectl run sidecar-pod --image=busybox --dry-run=client -o yaml >
pod.yaml

# Step 2: Edit pod.yaml (70 sec)
# - Add volumes section with emptyDir
# - Add second container (log-sidecar)
# - Add volumeMounts to both containers
# - Add commands to both containers
kubectl apply -f pod.yaml

# Step 3: Verify (10 sec)
kubectl logs sidecar-pod -c log-sidecar
```

**TIP:** emptyDir volume name MUST match in volumeMounts!

---

## Question 10: Fix Deprecated API Version

**Weight:** ~5% | **Domain:** Application Environment, Configuration

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A legacy manifest at `/tmp/legacy-deployment.yaml` uses deprecated API version (`apps/v1beta1`).

### Tasks

1. Review the manifest at `/tmp/legacy-deployment.yaml`
2. Update to `apps/v1`
3. Add selector field (required in `apps/v1`)
4. Ensure `selector.matchLabels` matches `template.metadata.labels`
5. Apply to namespace `'migration-ns'`

### Namespace

`migration-ns`

### Solution

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: legacy-app
  namespace: migration-ns
spec:
  replicas: 2
  selector:
    matchLabels:
      app: legacy-app
      version: v1
  template:
    metadata:
      labels:
        app: legacy-app
        version: v1
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
```

## Key Differences: apps/v1beta1 vs apps/v1

- apps/v1beta1: No selector required
- apps/v1: selector field is REQUIRED
- selector.matchLabels MUST match template.metadata.labels

### ⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: Fix API version + add selector (40 sec)
vim /tmp/legacy-deployment.yaml
# Change: apiVersion: apps/v1beta1 -> apiVersion: apps/v1
# Add selector.matchLabels with SAME labels as template.metadata.labels
# Save (:wq)

# Step 2: Apply (10 sec)
kubectl apply -f /tmp/legacy-deployment.yaml

# Step 3: Verify (10 sec)
kubectl get deployment legacy-app -n migration-ns
```

**KEY:** selector.matchLabels MUST exactly match template.metadata.labels!

---

## Question 11: Modify Deployment Twice, Then Rollback

**Weight:** ~7% | **Domain:** Application Design and Build

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'web-app' exists in namespace 'health-ns'. Make changes and demonstrate rollback.

### Tasks

1. First modification: Add Readiness Probe (httpGet, path: /healthz, port: 8080, initialDelaySeconds: 5, periodSeconds: 10)
2. Second modification: Update image from nginx:1.21 to nginx:1.22
3. Verify both changes
4. Rollback to PREVIOUS revision
5. After rollback: image should be nginx:1.21 but readiness probe should remain

### Namespace

health-ns

### Solution

#### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit deployment web-app -n health-ns
```

Find the containers section and add readinessProbe:

```
spec:
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        # ADD THIS SECTION:
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 10
```

Save and exit (:wq in vim)

### Method B: Using kubectl patch

```
kubectl patch deployment web-app -n health-ns --type='json' -p='[
  {
    "op": "add",
    "path": "/spec/template/spec/containers/0/readinessProbe",
    "value": {
      "httpGet": {
        "path": "/healthz",
        "port": 8080
      },
      "initialDelaySeconds": 5,
      "periodSeconds": 10
    }
  }
]'
```

```
kubectl rollout status deployment/web-app -n health-ns
```

### Step 2: Update Image

```
kubectl set image deployment/web-app nginx=nginx:1.22 -n health-ns
kubectl rollout status deployment/web-app -n health-ns
```

### Step 3: Check History

```
kubectl rollout history deployment/web-app -n health-ns
```

### Step 4: Rollback

```
kubectl rollout undo deployment/web-app -n health-ns
kubectl rollout status deployment/web-app -n health-ns
```

### Rollout Commands

```
kubectl rollout status deployment/<name> -n <ns>      # Watch progress
kubectl rollout history deployment/<name> -n <ns>     # View history
kubectl rollout undo deployment/<name> -n <ns>        # Rollback to previous
kubectl rollout undo deployment/<name> -n <ns> --to-revision=2 # Specific
revision
```

 Fastest Exam Approach (< 90 seconds)

```
# Step 1: Add readiness probe with kubectl edit (40 sec)
kubectl edit deployment web-app -n health-ns
# Add readinessProbe section under containers[0]
# Save (:wq)
kubectl rollout status deployment/web-app -n health-ns

# Step 2: Update image (20 sec)
kubectl set image deployment/web-app nginx=nginx:1.22 -n health-ns
kubectl rollout status deployment/web-app -n health-ns

# Step 3: Rollback to previous (30 sec)
kubectl rollout undo deployment/web-app -n health-ns
kubectl rollout status deployment/web-app -n health-ns
```

**TIP:** kubectl set image creates a new revision - kubectl edit also does!

---

## Question 12: Job with Failure Policy

**Weight:** ~6% | **Domain:** Application Design and Build

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

Create a Kubernetes Job with specific failure handling policies.

### Tasks

1. Create Job 'backup-job' in namespace 'job-ns'
2. Image: busybox:1.35
3. Command: sh -c "echo Starting backup && sleep 5 && echo Backup complete"
4. backoffLimit: 3 (retry count on failure)
5. activeDeadlineSeconds: 60 (max time for Job to complete)

**NOTE:** In the actual exam, specific values will be provided in the question

### Namespace

job-ns

### Solution

```
apiVersion: batch/v1
kind: Job
metadata:
  name: backup-job
  namespace: job-ns
spec:
  backoffLimit: 3
  activeDeadlineSeconds: 60
  template:
    spec:
      containers:
        - name: backup
          image: busybox:1.35
          command:
            - sh
            - -c
            - "echo Starting backup && sleep 5 && echo Backup complete"
      restartPolicy: Never
```

### Job Parameters

- **backoffLimit:** Number of retries before considering job failed (default 6)
- **activeDeadlineSeconds:** Max duration for job
- **restartPolicy:** Must be "Never" or "OnFailure" (not "Always")

⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: Generate Job YAML (15 sec)
kubectl create job backup-job --image=busybox:1.35 -n job-ns \
  --dry-run=client -o yaml > job.yaml

# Step 2: Edit job.yaml (35 sec)
# Add: backoffLimit, activeDeadlineSeconds
# Add: command section
kubectl apply -f job.yaml

# Step 3: Verify (10 sec)
kubectl get job backup-job -n job-ns
```

**TIP:** restartPolicy MUST be Never or OnFailure for Jobs!

---

## Question 13: Docker Build + OCI Export

**Weight:** ~7% | **Domain:** Application Design and Build

### Scenario

Build and export a container image from a Dockerfile at `/tmp/app/Dockerfile`.

### Tasks

1. Review Dockerfile
2. Build image with tag: `myapp:v1`
3. Export to OCI format at: `/tmp/myapp-v1.tar`

### Solution

#### Step 1: Build Image

```
cd /tmp/app
docker build -t myapp:v1 .
# OR: podman build -t myapp:v1 .
```

#### Step 2: Verify

```
docker images myapp:v1
```

#### Step 3: Export

```
docker save -o /tmp/myapp-v1.tar myapp:v1
# OR: podman save -o /tmp/myapp-v1.tar myapp:v1
```

### Docker/Podman Commands

```
docker build -t <tag> <path>           # Build image
docker save -o <output.tar> <image>    # Save image to tar
docker load -i <file.tar>              # Load image from tar
docker tag <image> <new_tag>           # Tag image
```

⚡ Fastest Exam Approach (< 45 seconds)

```
# THREE commands - that's it!
cd /tmp/app
```

```
docker build -t myapp:v1 .  
docker save -o /tmp/myapp-v1.tar myapp:v1
```

**TIP:** Exam may use podman instead of docker - same syntax!

---

## Question 14: SecurityContext - Edit Existing Deployment

**Weight:** ~7% | **Domain:** Application Environment, Configuration & Security

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'secure-app' in namespace 'secure-ns' is paused and needs security hardening.

### Tasks

1. Edit existing Deployment (do NOT delete and recreate)
2. Add Pod-level securityContext: runAsUser: 10000
3. Add Container-level securityContext:
  - allowPrivilegeEscalation: false
  - capabilities: add: ["NET\_BIND\_SERVICE"]
4. Resume the paused deployment

### Namespace

secure-ns

### Solution

#### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit deployment secure-app -n secure-ns
```

Find the spec.template.spec section and add securityContext:

```
spec:
  template:
    spec:
      securityContext:           # ADD THIS SECTION (pod-level)
        runAsUser: 10000
      containers:
      - name: app
        image: nginx
        securityContext:       # ADD THIS SECTION (container-level)
          allowPrivilegeEscalation: false
          capabilities:
            add:
            - NET_BIND_SERVICE
```

Save and exit (:wq in vim)

### Method B: Using kubectl patch

```
kubectl patch deployment secure-app -n secure-ns --type='json' -p='[
  {
    "op": "add",
    "path": "/spec/template/spec/securityContext",
    "value": {
      "runAsUser": 10000
    }
  },
  {
    "op": "add",
    "path": "/spec/template/spec/containers/0/securityContext",
    "value": {
      "allowPrivilegeEscalation": false,
      "capabilities": {
        "drop": ["ALL"],
        "add": ["NET_BIND_SERVICE"]
      }
    }
  }
]
```

### Step 2: Resume Deployment

```
kubectl rollout resume deployment/secure-app -n secure-ns
kubectl rollout status deployment/secure-app -n secure-ns
```

### Key Points

- Pod securityContext: spec.template.spec.securityContext
- Container securityContext: spec.template.spec.containers[].securityContext

### ⚡ Fastest Exam Approach (< 90 seconds)

```
# Step 1: Edit deployment to add securityContext (60 sec)
kubectl edit deployment secure-app -n secure-ns
# Add pod-level: spec.template.spec.securityContext.runAsUser: 10000
# Add container-level: spec.template.spec.containers[0].securityContext
#   - allowPrivilegeEscalation: false
#   - capabilities.add: ["NET_BIND_SERVICE"]
# Save (:wq)

# Step 2: Resume deployment (20 sec)
```

```
kubectl rollout resume deployment/secure-app -n secure-ns  
kubectl rollout status deployment/secure-app -n secure-ns
```

**KEY:** Pod=runAsUser, Container=capabilities,allowPrivilegeEscalation

---

## Question 15: Find Existing ServiceAccount and Apply to Deployment

**Weight:** ~8% | **Domain:** Application Environment, Configuration & Security

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'scraper-app' in namespace 'rbac-ns' needs to list pods but gets "Forbidden" errors. Several ServiceAccounts already exist - one has the correct permissions.

### Tasks

1. Examine existing ServiceAccounts in namespace 'rbac-ns'
2. Find which ServiceAccount has permissions to list pods
3. Update Deployment to use the correct existing ServiceAccount

### Namespace

**rbac-ns**

### Solution

#### Step 1: List ServiceAccounts

```
kubectl get sa -n rbac-ns
```

#### Step 2: Check RoleBinding

```
kubectl get rolebinding -n rbac-ns -o yaml | grep -A5 "subjects:"
```

#### Step 3: Test Permissions

```
kubectl auth can-i list pods --as=system:serviceaccount:rbac-ns:scraper-sa  
-n rbac-ns
```

#### Step 4: Update Deployment

```
kubectl set serviceaccount deployment/scraper-app scraper-sa -n rbac-ns
```

## Key Learning

1. List ServiceAccounts: `kubectl get sa -n <ns>`
2. Check RoleBindings: `kubectl get rolebinding -n <ns> -o yaml`
3. Test with: `kubectl auth can-i <verb> <resource> --as=system:serviceaccount:<ns>:<sa>`

### ⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: List ServiceAccounts (10 sec)
kubectl get sa -n rbac-ns

# Step 2: Find which SA has pod-list permissions (30 sec)
kubectl get rolebinding -n rbac-ns -o yaml | grep -A5 "subjects:"
# OR test each SA:
kubectl auth can-i list pods --as=system:serviceaccount:rbac-ns:scraper-sa -n rbac-ns

# Step 3: Update deployment (20 sec)
kubectl set serviceaccount deployment/scraper-app scraper-sa -n rbac-ns
```

**TIP:** `kubectl auth can-i` is FASTEST way to test permissions!

---

## Question 16: Secret with Multiple Keys

**Weight:** ~6% | **Domain:** Application Environment, Configuration & Security

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'db-app' in namespace 'secret-ns' is using hardcoded environment variables. Migrate to use a Secret.

### Tasks

1. Create Secret 'db-secret' with keys:
  - DB\_HOST=mysql.database.svc
  - DB\_USER=admin
  - DB\_PASSWORD=secret123
  - DB\_NAME=myapp
2. Update Deployment to load secret keys as environment variables

### Namespace

secret-ns

### Solution

#### Step 1: Create Secret (Imperative - Fast for Exam)

```
kubectl create secret generic db-secret -n secret-ns \
  --from-literal=DB_HOST=mysql.database.svc \
  --from-literal=DB_USER=admin \
  --from-literal=DB_PASSWORD=secret123 \
  --from-literal=DB_NAME=myapp
```

#### Step 2: Update Deployment (Fastest Method)

```
kubectl set env deployment/db-app -n secret-ns --from=secret/db-secret
```

#### Alternative: Using YAML secretKeyRef

```
env:
- name: DB_HOST
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: DB_HOST
- name: DB_USER
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: DB_USER
# ... etc
```

### ⚡ Fastest Exam Approach (< 45 seconds)

```
# Step 1: Create secret (20 sec)
kubectl create secret generic db-secret -n secret-ns \
  --from-literal=DB_HOST=mysql.database.svc \
  --from-literal=DB_USER=admin \
  --from-literal=DB_PASSWORD=secret123 \
  --from-literal=DB_NAME=myapp

# Step 2: Inject ALL keys as env vars (10 sec)
kubectl set env deployment/db-app -n secret-ns --from=secret/db-secret

# Step 3: Verify (15 sec)
kubectl rollout status deployment/db-app -n secret-ns
```

**TIP:** kubectl set env --from=secret/NAME injects ALL keys at once!

---

## Question 17: Expose Deployment with NodePort

**Weight:** ~5% | **Domain:** Services & Networking

### Context

```
kubectl config use-context k8s-cluster2
```

### Scenario

A Deployment 'frontend-app' exists in namespace 'web-ns' running on port 8080.

### Tasks

Create NodePort Service 'frontend-service':

- Service port: 80
- Target port: 8080
- NodePort: 30080

### Namespace

web-ns

### Solution

#### Method 1: Imperative + kubectl edit

```
kubectl expose deployment frontend-app -n web-ns \
  --name=frontend-service \
  --type=NodePort \
  --port=80 \
  --target-port=8080
```

Then use kubectl edit to set specific NodePort:

```
kubectl edit svc frontend-service -n web-ns
```

Find the ports section and add nodePort:

```
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
```

```
nodePort: 30080    # ADD THIS LINE
protocol: TCP
```

Save and exit (:wq in vim)

### Method 2: Imperative + kubectl patch

```
kubectl expose deployment frontend-app -n web-ns \
  --name=frontend-service \
  --type=NodePort \
  --port=80 \
  --target-port=8080

kubectl patch svc frontend-service -n web-ns \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/ports/0/nodePort", "value":
30080}]'
```

### Method 3: YAML (Recommended for specific nodePort)

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
  namespace: web-ns
spec:
  type: NodePort
  selector:
    app: frontend-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30080
```

### Key Points

- NodePort range: 30000-32767
- Service port (80): port clients use within cluster
- Target port (8080): container port
- NodePort (30080): external port on each node

### ⚡ Fastest Exam Approach (< 45 seconds)

```
# Step 1: Expose deployment (10 sec)
kubectl expose deployment frontend-app -n web-ns \
  --name=frontend-service --type=NodePort \
```

```
--port=80 --target-port=8080
```

```
# Step 2: Set specific nodePort with edit (20 sec)
```

```
kubectl edit svc frontend-service -n web-ns
```

```
# Under ports[0] add: nodePort: 30080
```

```
# Save (:wq)
```

```
# Step 3: Verify (15 sec)
```

```
kubectl get svc frontend-service -n web-ns
```

**TIP:** kubectl expose cannot set specific nodePort, so edit after!

---

## Question 18: Fix Deployment - Container Name & Image + Resume Rollout

**Weight:** ~8% | **Domain:** Application Deployment

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Deployment 'api-server' in namespace 'api-ns' is paused and has incorrect container configuration.

### Tasks

1. Edit existing Deployment (do NOT delete and recreate)
2. Fix container name: wrong-name → api-container
3. Fix image: nginx:wrong → nginx:1.21
4. Resume the paused deployment

### Namespace

api-ns

### Solution

#### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit deployment api-server -n api-ns
```

Find the containers section and fix:

```
spec:
  template:
    spec:
      containers:
      - name: api-container      # CHANGE FROM: wrong-name
        image: nginx:1.21      # CHANGE FROM: nginx:wrong
        ports:
        - containerPort: 80
```

Save and exit (:wq in vim)

#### Method B: Using kubectl patch

```
kubectl patch deployment api-server -n api-ns --type='json' -p='[
  {"op": "replace", "path": "/spec/template/spec/containers/0/name",
"value": "api-container"},
  {"op": "replace", "path": "/spec/template/spec/containers/0/image",
"value": "nginx:1.21"}
]'
```

## Step 2: Resume Deployment

```
kubectl rollout resume deployment/api-server -n api-ns
kubectl rollout status deployment/api-server -n api-ns
```

## Key Points

- NEVER delete and recreate - always edit in-place (preserves history)
- kubectl rollout resume to unpause
- kubectl rollout status to wait for completion

## ⚡ Fastest Exam Approach (< 60 seconds)

```
# Step 1: Edit deployment to fix issues (40 sec)
kubectl edit deployment api-server -n api-ns
# Fix container name: wrong-name -> api-container
# Fix image: nginx:wrong -> nginx:1.21
# Save (:wq)

# Step 2: Resume if paused (10 sec)
kubectl rollout resume deployment/api-server -n api-ns

# Step 3: Verify (10 sec)
kubectl rollout status deployment/api-server -n api-ns
```

**TIP:** kubectl edit is FASTEST for multiple field fixes in one session!

---

## Question 19: Service Selector Fix

**Weight:** ~6% | **Domain:** Services & Networking

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

A Service 'frontend-svc' in namespace 'svc-ns' has no endpoints. The Deployment 'frontend-app' exists and pods are running.

### Tasks

1. Troubleshoot why Service has no endpoints
2. Fix Service selector to match Deployment's pod labels
3. Verify Service now has endpoints

### Namespace

**svc-ns**

### Solution

#### Step 1: Check Service Selector

```
kubectl get svc frontend-svc -n svc-ns -o yaml
```

#### Step 2: Check Pod Labels

```
kubectl get pods -n svc-ns --show-labels
```

#### Step 3: Check Endpoints

```
kubectl get endpoints frontend-svc -n svc-ns
```

#### Step 4: Fix Service Selector

##### Method A: Using kubectl edit (RECOMMENDED for exam)

```
kubectl edit svc frontend-svc -n svc-ns
```

Find the selector section and fix:

```
spec:
  selector:
    app: frontend      # CHANGE FROM: frontend-wrong
    tier: web
```

Save and exit (:wq in vim)

### Method B: Using kubectl patch

```
kubectl patch svc frontend-svc -n svc-ns \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/selector/app", "value":
"frontend"}]'
```

### Key Points

- Service selector MUST match Pod labels exactly
- No endpoints = selector mismatch or no running pods
- All selector labels must match (AND logic)

### ⚡ Fastest Exam Approach (< 45 seconds)

```
# Step 1: Compare service selector vs pod labels (15 sec)
kubectl get svc frontend-svc -n svc-ns -o jsonpath="{.spec.selector}"
kubectl get pods -n svc-ns --show-labels
# Find the mismatch!

# Step 2: Fix selector with kubectl edit (20 sec)
kubectl edit svc frontend-svc -n svc-ns
# Change: app: frontend-wrong -> app: frontend
# Save (:wq)

# Step 3: Verify endpoints exist (10 sec)
kubectl get endpoints frontend-svc -n svc-ns
```

**TIP:** No endpoints = selector mismatch! Use --show-labels for quick comparison.

## Question 20: Pod with Command

**Weight:** ~4% | **Domain:** Application Design and Build

### Context

```
kubectl config use-context k8s-cluster1
```

### Scenario

Create a simple Pod with a custom command.

### Tasks

1. Create Pod 'simple-pod' in namespace 'cmd-ns'
2. Image: busybox:1.35
3. Command: sleep 3600
4. Ensure Pod stays in Running state

### Namespace

cmd-ns

### Solution

#### Method 1: kubectl run (Fastest for Exam)

```
kubectl run simple-pod --image=busybox:1.35 -n cmd-ns --command -- sleep 3600
```

#### Method 2: YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-pod
  namespace: cmd-ns
spec:
  containers:
    - name: busybox
      image: busybox:1.35
      command: ["sleep", "3600"]
```

### Command vs Args

- **command** = Docker ENTRYPOINT (overrides it)
- **args** = Docker CMD (overrides it)

## kubectl run Flags

```
--command      # Use command array instead of args
--restart=Never  # Create a Pod (not a Deployment)
--dry-run=client -o yaml    # Generate YAML without creating
```

## ⚡ Fastest Exam Approach (< 15 seconds!)

```
# ONE COMMAND - that's it!
kubectl run simple-pod --image=busybox:1.35 -n cmd-ns --command -- sleep
3600

# Verify:
kubectl get pod simple-pod -n cmd-ns
```

**CRITICAL:** --command flag MUST come BEFORE the -- separator!

---

## Quick Reference Commands

### Viewing Resources

```
kubectl get <resource> -n <namespace>
kubectl describe <resource> <name> -n <namespace>
kubectl get <resource> -o yaml
kubectl get pods --show-labels
```

### Editing Resources

```
kubectl edit <resource> <name> -n <namespace>
kubectl patch <resource> <name> -n <namespace> --type='json' -p='[...]'
kubectl set image deployment/<name> <container>=<image>
kubectl set serviceaccount deployment/<name> <sa-name>
```

### Rollouts

```
kubectl rollout status deployment/<name>
kubectl rollout history deployment/<name>
kubectl rollout undo deployment/<name>
```

```
kubectl rollout resume deployment/<name>
kubectl rollout pause deployment/<name>
```

## RBAC

```
kubectl create serviceaccount <name> -n <namespace>
kubectl create role <name> --verb=<verbs> --resource=<resources>
kubectl create rolebinding <name> --role=<role> --serviceaccount=<ns>:<sa>
kubectl auth can-i <verb> <resource> --as=system:serviceaccount:<ns>:<sa>
```

## Secrets and ConfigMaps

```
kubectl create secret generic <name> --from-literal=<key>=<value>
kubectl create configmap <name> --from-literal=<key>=<value>
kubectl set env deployment/<name> --from=secret/<secret-name>
```

## Debugging

```
kubectl logs <pod> -c <container>
kubectl exec -it <pod> -- <command>
kubectl describe pod <name>
kubectl get events -n <namespace>
```

---

*Generated from CKAD-Exam-2025 practice repository*