CONDENSED REFERENCE · GENERATED 2026-09-02

CKA One-Book Reference

Decision trees, the imperative commands worth memorising, mnemonics for what must be recalled cold, and the flags that cost minutes when you have to look them up.

CNCF curriculum v1.35 · fetched 2026-09-02 · exam is 120 minutes, ~17 performance-based tasks, 66% to pass.
Generated from the site's markdown and JSON sources — do not edit this PDF, edit the source and rebuild.
Contains no exam content. Every practice scenario is original, inferred from published curriculum weights and public community retrospectives.

Contents

Command cheat sheet

Everything here is imperative. If a task can be done without opening an editor, do it without opening an editor.

Setup — the first 60 seconds of the exam

```
alias k=kubectl
source <(kubectl completion bash)
complete -o default -F __start_kubectl k
export do='--dry-run=client -o yaml'
export now='--force --grace-period=0'
export ns='-n '        # then:  k get po $ns dev

cat >> ~/.vimrc <<'EOF'
set expandtab tabstop=2 shiftwidth=2 number
EOF
```

Why: `$do` turns every `create` / `run` into a manifest generator; `set expandtab` prevents literal tabs, which YAML rejects outright. Community write-ups put the saving at 20–30 minutes across the exam.

Per-task, before anything else:

```
kubectl config use-context <given-in-the-task>
kubectl config current-context
```

Discovery — instead of the docs tab

```
kubectl api-resources                      # every kind, its group, short name, namespaced?
kubectl api-resources --namespaced=false    # cluster-scoped kinds (PV, node, CRD, ClusterRole)
kubectl explain deploy.spec.template.spec.containers --recursive
kubectl explain networkpolicy.spec --recursive | head -40
kubectl <verb> --help | grep -A2 example    # every kubectl subcommand ships examples
```

Why: `explain --recursive` prints the schema from the API server you're connected to — always the right version, always faster than a browser tab.

Generate, don't author

```
kubectl run web --image=nginx $do > pod.yaml
kubectl run web --image=nginx --restart=Never --rm -it -- sh      # throwaway debug Pod
kubectl create deploy web --image=nginx --replicas=3 $do > deploy.yaml
kubectl create job pi --image=perl $do -- perl -Mbignum=bpi -wle 'print bpi(200)'
kubectl create cronjob rep --image=busybox --schedule="*/5 * * * *" $do -- /bin/sh -c date
kubectl expose deploy web --port=80 --target-port=8080 $do > svc.yaml
kubectl create ingress web --rule="host/path*=svc:80" --class=nginx $do
kubectl create configmap cfg --from-literal=k=v $do
kubectl create secret generic s --from-literal=p=x $do
kubectl create role r --verb=get,list --resource=pods $do
kubectl create rolebinding rb --role=r --serviceaccount=ns:sa $do
kubectl create quota q --hard=cpu=2,memory=4Gi $do
kubectl create poddisruptionbudget pdb --selector=app=web --min-available=2 $do
```

> There is **no** imperative generator for: PersistentVolume, PersistentVolumeClaim, StorageClass, NetworkPolicy, DaemonSet, StatefulSet, Gateway, HTTPRoute, or tolerations/affinity stanzas. That list is your memorisation set — everything else on this page you can generate.

Inspect

```
kubectl get po -A -o wide
kubectl get po --show-labels
kubectl get po -l app=web,tier!=cache
kubectl get po --field-selector status.phase=Running
kubectl get po --sort-by=.status.startTime
kubectl get events -A --sort-by=.lastTimestamp | tail -20
kubectl get po -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName,IMG:.spec.containers[*].image
kubectl get po <p> -o jsonpath='{.spec.containers[*].image}{"\n"}'
kubectl get no -o json | jq '.items[].status.capacity'
kubectl describe po <p> | sed -n '/Events/,$p'
kubectl top no ; kubectl top po -A --sort-by=memory
```

Modify without an editor

```
kubectl set image deploy/web nginx=nginx:1.27
kubectl set resources deploy/web --requests=cpu=100m --limits=memory=256Mi
kubectl set env deploy/web --from=configmap/cfg
kubectl set serviceaccount deploy/web app-sa
kubectl label node w1 disktype=ssd
kubectl annotate deploy web owner=team-a
kubectl scale deploy/web --replicas=5
kubectl patch deploy web -p '{"spec":{"replicas":4}}'
kubectl patch pv pv1 -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
kubectl patch deploy web --type=json -p='[{"op":"replace","path":"/spec/replicas","value":4}]'
```

Why two patch types: a strategic-merge `-p '{...}'` handles most fields; `--type=json` is for list surgery (add/remove an element at an index) where merge semantics are ambiguous.

Delete

```
kubectl delete po web $now
kubectl delete po -l app=web
kubectl delete all -l app=web -n dev          # NB: "all" ≠ everything; no PVC, no Secret, no CM
kubectl delete -k ./overlays/prod
```

Logs and exec

```
kubectl logs <p> -f
kubectl logs <p> --previous
kubectl logs <p> -c <container> --tail=50 --since=10m
kubectl logs -l app=web --all-containers --prefix
kubectl logs <p> | grep ERROR > /opt/answer.txt && cat /opt/answer.txt
kubectl exec -it <p> -- sh
kubectl exec <p> -c <c> -- env
kubectl cp <ns>/<p>:/etc/config/app.conf ./app.conf
kubectl port-forward svc/web 8080:80
kubectl debug -it <p> --image=busybox:1.36 --target=<container>   # ephemeral container
```

Node and cluster lifecycle

```
kubectl cordon w1 ; kubectl uncordon w1
kubectl drain w1 --ignore-daemonsets --delete-emptydir-data --force
kubectl get no -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
kubectl taint no w1 key=val:NoSchedule
kubectl taint no w1 key=val:NoSchedule-
kubeadm token create --print-join-command
kubeadm certs check-expiration
kubeadm upgrade plan
kubeadm upgrade apply v1.35.0      # first control-plane node
kubeadm upgrade node               # every other node
```

On the node, when kubectl is gone

```
systemctl status kubelet
journalctl -u kubelet --no-pager -n 50
sudo crictl ps -a
sudo crictl logs <id>
sudo crictl pods
ls /etc/kubernetes/manifests/
sudo ss -lntp | grep 6443
```

etcd

```
export ETCDCTL_API=3
E="--cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --
key=/etc/kubernetes/pki/etcd/server.key --endpoints=https://127.0.0.1:2379"
etcdctl $E snapshot save /opt/snap.db
etcdctl --write-out=table snapshot status /opt/snap.db
etcdctl $E endpoint health
etcdctl $E member list --write-out=table
etcdctl snapshot restore /opt/snap.db --data-dir=/var/lib/etcd-restore
# then edit /etc/kubernetes/manifests/etcd.yaml: --data-dir AND the hostPath volume
```

Helm and Kustomize

```
helm repo add x <url> && helm repo update
helm search repo x --versions
helm show values x/chart > values.yaml
helm install rel x/chart -n ns --create-namespace --version 1.2.3 -f values.yaml
helm upgrade --install rel x/chart -n ns --set replicaCount=3
helm list -A ; helm history rel -n ns ; helm rollback rel 1 -n ns
helm uninstall rel -n ns
kubectl kustomize ./overlays/prod
kubectl apply -k ./overlays/prod
```

RBAC checks

```
kubectl auth can-i list pods -n dev --as=system:serviceaccount:dev:app-sa
kubectl auth can-i --list --as=jane
kubectl auth whoami
```

Flags worth memorising

Flag	Where	Does
`--dry-run=client -o yaml`	create/run/expose	generate a manifest, don't send it
`-o wide`	get	node and Pod IP columns
`-o jsonpath='{...}'`	get	pull one field for a scripted answer
`--sort-by=`	get	`.lastTimestamp`, `.status.startTime`, `memory`
`--show-labels`	get	the labels a selector has to match
`-l k=v` / `--field-selector`	get/delete	label vs. server-side field filtering
`--previous`	logs	the container instance that died
`--all-containers`	logs	every container in the Pod
`--ignore-daemonsets`	drain	almost always required
`--delete-emptydir-data`	drain	required if any Pod has an emptyDir
`--force --grace-period=0`	delete	immediate; only when you mean it
`--record`	*removed*	don't reach for it; it's gone
`-k`	apply/delete	run kustomize first
`--as`	any	impersonate, for RBAC verification
`-A` / `--all-namespaces`	get	cluster-wide

Mnemonics

Ordering and exact sets — the material that fails under time pressure for reasons other than comprehension. Generated from `data/mnemonics.json`.

Cluster Architecture

kubeadm upgrade order

DUPA-KRU — Drain, Unhold, Plan, Apply, Kubelet, Restart, Uncordon

D	`Drain the node (--ignore-daemonsets)`
U	`Unhold + install the new kubeadm, then hold again`
P	`Plan — kubeadm upgrade plan`
A	`Apply — 'upgrade apply vX.Y.Z' on the FIRST control-plane node; 'upgrade node' on every other node`
K	`Kubelet + kubectl: unhold, install, hold`
R	`Restart — daemon-reload && systemctl restart kubelet`
U	`Uncordon the node`

Why: kubeadm is only an installer: the binary upgrade does nothing until 'upgrade apply' rewrites the static Pod manifests, and kubeadm never touches the kubelet package, which is why K and R are separate steps you do by hand.

etcd restore — the two edits

Restore, then change BOTH: the flag and the volume

1	`etcdctl snapshot restore <file> --data-dir=/var/lib/etcd-restore`
2	`etcd.yaml: --data-dir=/var/lib/etcd-restore (the container's view)`
3	`etcd.yaml: volumes[etcd-data].hostPath.path=/var/lib/etcd-restore (the host's view)`
4	`Wait — the kubelet recreates the static Pod on file change. Restart nothing.`

Why: --data-dir is a path inside the container and the hostPath volume is what maps it to disk. Change one without the other and the restore appears to succeed while etcd serves the old data.

etcdctl's three mandatory TLS flags

CA, Cert, Key — Can't Connect Keyless

`--cacert`	`/etc/kubernetes/pki/etcd/ca.crt`
`--cert`	`/etc/kubernetes/pki/etcd/server.crt`
`--key`	`/etc/kubernetes/pki/etcd/server.key`
`--endpoints`	`https://127.0.0.1:2379`

Why: etcd speaks mutual TLS only. Omit any of the three and the rejection looks exactly like an outage. If you blank on the paths, grep them out of /etc/kubernetes/manifests/etcd.yaml.

Which RBAC pair?

Namespaced resource → Role. Cluster-scoped resource → ClusterRole. Reusable definition, one namespace → ClusterRole + RoleBinding.

`Role + RoleBinding`	`namespaced resources, one namespace`
`ClusterRole + ClusterRoleBinding`	`cluster-scoped resources (nodes, PVs, namespaces, CRDs) or every namespace`
`ClusterRole + RoleBinding`	`define once, grant only inside one namespace`
`Role + ClusterRoleBinding`	`does not exist — invalid`

Why: The binding's kind decides the scope of the grant; the role's kind decides where the definition can live. That's why the third combination is legal and the fourth is not.

RBAC verbs

GLW-CUP-D-E — Get List Watch, Create Update Patch, Delete, and the odd ones

Read	`get, list, watch`
Write	`create, update, patch`
Remove	`delete, deletecollection`
Odd	`* (all), plus non-resource URLs via nonResourceURLs`

Why: 'get' fetches one object by name and does NOT imply 'list'. A role with get but not list makes 'kubectl get pods' fail while 'kubectl get pod web' works — a classic partially-correct answer.

Why drain refuses

DEF — DaemonSets, EmptyDir, Free-standing pods

D	`DaemonSet Pods can't be evicted → --ignore-daemonsets`
E	`emptyDir data would be destroyed → --delete-emptydir-data`
F	`Free-standing (uncontrolled) Pods won't come back → --force`
+	`Hangs forever on 'would violate the disruption budget' → a PodDisruptionBudget, not a bug`

Why: Each refusal names its own cause in the error text. The PDB case is the one that hangs instead of erroring, which is why it reads as a broken command.

Troubleshooting

Pod is Pending — the four causes

CTAP — Capacity, Taints, Affinity, PVC

C	`Capacity: 'Insufficient cpu/memory' — requests exceed what's allocatable`
T	`Taints: 'had untolerated taint'`
A	`Affinity: 'didn't match Pod's node affinity/selector'`
P	`PVC unbound, or no events at all → the scheduler itself is down`

Why: The scheduler writes the failed predicate verbatim into the FailedScheduling event, so describe answers this in one read. No events at all is the tell that nothing is scheduling anything.

Static Pods

File on disk, kubelet runs it, no scheduler, no kubectl delete.

Where	`/etc/kubernetes/manifests/ (staticPodPath in /var/lib/kubelet/config.yaml)`
Who	`the kubelet, directly — the API server is not involved`
Restart	`there is none; edit the file and the kubelet recreates the Pod`
Delete	`move the file out of the directory; kubectl delete just recreates it`

Why: This is exactly why a dead control plane is repairable: kube-apiserver, etcd, scheduler and controller-manager are static Pods, so they can be fixed from disk with no working API server.

Workloads Scheduling

Taint effects

NoSchedule keeps them out. PreferNoSchedule asks nicely. NoExecute throws them out.

NoSchedule	`no new Pods without a toleration; existing Pods stay`
PreferNoSchedule	`soft — the scheduler avoids the node if it can`
NoExecute	`as NoSchedule, and evicts running Pods that don't tolerate it`

Why: 'Make the existing Pods leave' is NoExecute. NoSchedule alone looks like it did nothing, because it only affects future scheduling.

The three probes

Liveness restarts. Readiness removes from endpoints. Startup buys time.

livenessProbe	`fails → container restarted`
readinessProbe	`fails → Pod removed from Service endpoints, NOT restarted`
startupProbe	`disables the other two until it first succeeds`

Why: A failing readiness probe presents as a networking bug — the Service has no endpoints — because unready Pods are excluded from endpoints. Without a startupProbe, a slow-booting app is killed by its own liveness probe in a loop that looks like a crash.

Storage

Why a PVC binds (or doesn't)

SAC — Size, Access modes, Class. All three, or nothing binds.

S	`PV capacity >= PVC request`
A	`PV accessModes is a superset of the PVC's`
C	`storageClassName matches EXACTLY — including the empty string`

Why: storageClassName: "" means 'no class, static binding only' and is not the same as omitting the field, which means 'use the default class'. With no default class, omitting it leaves the PVC Pending forever.

Access modes

RWO is per NODE, not per Pod. That's why RWOP exists.

RWO	`ReadWriteOnce — read-write by Pods on one node (several Pods on that node is fine)`
ROX	`ReadOnlyMany — read-only from many nodes`
RWX	`ReadWriteMany — read-write from many nodes; needs NFS/CephFS-class backing`
RWOP	`ReadWriteOncePod — exactly one Pod cluster-wide`

Why: The 'Once' in ReadWriteOnce counts nodes. Reading it as 'one Pod' is the most common wrong answer about storage.

Services Networking

NetworkPolicy: one dash changes the meaning

Two dashes = OR. One dash = AND.

OR	`- podSelector: {...}\n- namespaceSelector: {...} → web Pods, or anything in that namespace`
AND	`- podSelector: {...}\n namespaceSelector: {...} → web Pods that are in that namespace`
Deny-all	`podSelector: {} selects every Pod; policyTypes with no matching rules denies that direction`
Additive	`multiple policies are UNIONed — there is no deny rule in NetworkPolicy`

Why: Entries in the 'from' list are separate peers (OR); selectors inside a single entry must all match the same peer (AND). Indentation is the only thing distinguishing them.

Cluster DNS names

service.namespace.svc.cluster.local — always four labels after the name

Service	`<service>.<namespace>.svc.cluster.local`
StatefulSet Pod	`<pod>.<headless-service>.<namespace>.svc.cluster.local`
Pod	`<pod-ip-with-dashes>.<namespace>.pod.cluster.local`

Why: Test with the FQDN. Short names depend on the search list in the Pod's /etc/resolv.conf, so a short-name failure may mean 'wrong namespace', not 'DNS is broken'.

Troubleshooting — 30%

The biggest domain, and the one you cannot bluff. Every task here starts from a **symptom**, so this page does too. Nothing on this page is organised by component; you don't meet components, you meet broken things.

Curriculum competencies: troubleshoot clusters and nodes · troubleshoot cluster components · monitor cluster and application resource usage · manage and evaluate container output streams · troubleshoot services and networking.

> **Step 0, every single task.** Before you read the symptom:
>
> ```
> kubectl config use-context <context-from-the-task>
> kubectl config current-context
> ```
>
> *Why:* a perfect fix on the wrong cluster scores zero, and nothing in the environment will tell you.

The universal first move

```
kubectl get events -A --sort-by=.lastTimestamp | tail -20
```

Why it works: the control plane narrates its own failures into Events. Scheduling refusals, image pull failures, probe failures, volume attach errors and OOM kills all land here with a reason string, before you have to guess at a component.

Its companion, scoped to one object:

```
kubectl describe pod <pod> -n <ns>          # Events section is at the bottom — read it first
```

Tree 1 — Pod is not Running

Start: `kubectl get pod <pod> -n <ns> -o wide`. Branch on the **STATUS** column.

```
Pod status?
├── Pending ──────────────► nothing has scheduled it, or it can't start on its node
│   ├── describe says "0/N nodes are available"
│   │   ├── "Insufficient cpu/memory"     → node capacity vs. requests
│   │   ├── "node(s) had untolerated taint" → taint vs. toleration
│   │   ├── "didn't match Pod's node affinity/selector" → nodeSelector / affinity
│   │   └── "had volume node affinity conflict" → PV zone vs. node zone
│   ├── describe says "persistentvolumeclaim ... not found" / PVC is Pending → storage tree
│   └── no events at all                  → scheduler is not running (Tree 4)
├── ContainerCreating ────► scheduled, kubelet is stuck setting it up
│   ├── FailedMount / "timed out waiting for the condition" → volume/secret/configmap missing
│   └── "network plugin is not ready"     → CNI not installed / broken on that node
├── ImagePullBackOff / ErrImagePull ► image name, tag, or registry credentials
├── CrashLoopBackOff ─────► it starts and dies; read the PREVIOUS logs
├── Error / Completed ────► process exited; check exit code in describe
├── Terminating (stuck) ──► finalizer, or the node is gone
└── Running but 0/1 READY ► readiness probe is failing, not the container
```

Pending — capacity, taints, affinity

```
kubectl describe pod <pod> -n <ns> | sed -n '/Events/,$p'
kubectl describe node <node> | grep -A5 'Allocated resources'
kubectl get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
```

Why: the scheduler writes the exact predicate that failed into the `FailedScheduling` event; `Allocated resources` shows requests already committed (not usage), which is what the scheduler actually compares against.

Fix depending on branch:

```
# untolerated taint: remove it from the node ...
kubectl taint nodes <node> key=value:NoSchedule-
# ... or tolerate it on the Pod (must be authored; there is no imperative form)
kubectl patch deploy <d> -n <ns> --type=json \
  -p='[{"op":"add","path":"/spec/template/spec/tolerations","value":
[{"key":"key","operator":"Equal","value":"value","effect":"NoSchedule"}]}]'

# insufficient capacity: lower the request
kubectl set resources deploy <d> -n <ns> --requests=cpu=100m,memory=128Mi
```

Why: a trailing `-` on a taint means "remove"; `set resources` edits the Pod template in place and rolls it, which is faster and less error-prone than `kubectl edit`.

CrashLoopBackOff — read the corpse, not the patient

```
kubectl logs <pod> -n <ns> --previous
kubectl logs <pod> -n <ns> -c <container> --previous     # multi-container
kubectl describe pod <pod> -n <ns> | grep -A3 'Last State'
```

Why: the running container is a fresh restart with no history. `--previous` reads the log of the instance that died, and `Last State` gives you its exit code — 1 (app error), 137 (SIGKILL/OOM), 143 (SIGTERM).

Exit code **137** with `Reason: OOMKilled` means the memory limit, not the app:

```
kubectl set resources deploy <d> -n <ns> --limits=memory=512Mi
```

Running but not Ready

```
kubectl describe pod <pod> -n <ns> | grep -A10 'Readiness'
kubectl exec <pod> -n <ns> -- wget -qO- localhost:8080/healthz
```

Why: Ready is owned by the readiness probe alone. If the probe's port/path/scheme don't match what the container serves, the Pod runs forever and is silently removed from every Service's endpoints — which shows up as a *networking* symptom two trees down.

Terminating forever

```
kubectl get pod <pod> -n <ns> -o jsonpath='{.metadata.finalizers}'
kubectl delete pod <pod> -n <ns> --force --grace-period=0
```

Why: the API server won't remove an object while a finalizer is listed; `--force --grace-period=0` drops the object from etcd without waiting for the kubelet, which is the right move only when the node is genuinely gone.

Tree 2 — Node is NotReady

```
kubectl get nodes → NotReady
├── Can you SSH to it?  no → infrastructure, out of scope for the fix
└── yes:
    ├── systemctl is-active kubelet
    │   ├── inactive/failed → journalctl -u kubelet (config error, bad flag, cert)
    │   └── activating (loop) → kubelet crashing on start; read the same journal
    ├── kubelet active but node still NotReady
    │   ├── describe node → "container runtime network not ready" → CNI missing/broken
    │   ├── describe node → DiskPressure / MemoryPressure → free space or evict
    │   └── describe node → "Kubelet stopped posting node status" → clock, network, or API reachability
    └── crictl ps  →  runtime itself dead? systemctl status containerd
```

```
kubectl describe node <node> | sed -n '/Conditions/,/Addresses/p'
# then on the node:
systemctl status kubelet
journalctl -u kubelet --no-pager -n 50
systemctl restart kubelet && systemctl enable kubelet
```

Why: `Conditions` is the kubelet's own report card — `Ready=False` carries a `message` naming the cause. The journal is the only place a kubelet that never finished starting can tell you why.

Config lives in two files that are the usual culprits after an edit:

File	Holds
`/var/lib/kubelet/config.yaml`	kubelet's own config (cgroup driver, `staticPodPath`, cluster DNS)
`/etc/kubernetes/kubelet.conf`	kubeconfig the kubelet uses to reach the API server
`/etc/kubernetes/pki/`	certificates; expired certs make the node go NotReady in unison with others

```
df -h /var/lib/kubelet          # DiskPressure
kubeadm certs check-expiration  # cluster-wide NotReady with cert errors in the journal
```

Tree 3 — Cluster-wide: `kubectl` itself fails

`The connection to the server <host>:6443 was refused` means the API server is down. You now have no `kubectl`, so you work from the control-plane node with `crictl` and `journalctl`.

```
kubectl unreachable
├── ss -lntp | grep 6443  →  nothing listening
│   ├── ls /etc/kubernetes/manifests/kube-apiserver.yaml   → present?
│   │   ├── no  → someone moved it; restore it, kubelet will start it within ~20s
│   │   └── yes → the manifest is bad, or a dependency is down
│   ├── crictl ps -a | grep apiserver  → exited container?
│   │   └── crictl logs <id>   ← the actual error: bad flag, bad cert path, etcd unreachable
│   └── journalctl -u kubelet | grep -i apiserver
└── something IS listening → your kubeconfig, not the server
    └── kubectl config view --minify        (wrong cluster/server/user)
```

```
# on the control-plane node
sudo crictl ps -a --name kube-apiserver
sudo crictl logs <container-id> 2>&1 | tail -30
sudo journalctl -u kubelet --no-pager | grep -iE 'apiserver|manifest' | tail -20
```

Why: control-plane components are **static Pods** — the kubelet reads `/etc/kubernetes/manifests/*.yaml` directly from disk and runs them without the API server's involvement. That's exactly why they can be repaired when the API is down, and why editing the manifest is the fix: the kubelet notices the file change and recreates the Pod automatically. There is no `systemctl restart kube-apiserver`.

> **Trap:** always `cp /etc/kubernetes/manifests/kube-apiserver.yaml /tmp/` before editing. A typo'd flag makes the API server crash-loop, and now you're editing YAML with no `kubectl` and no undo.

Tree 4 — A control-plane component other than the API server

If `kubectl` works, the API server is fine. Check the rest:

```
kubectl get pods -n kube-system
kubectl get componentstatuses           # deprecated but still answers on many clusters
```

```
Symptom → suspect
├── Pods stay Pending, no FailedScheduling events → kube-scheduler
├── Deployment created, no ReplicaSet appears     → kube-controller-manager
├── ReplicaSet exists, no Pods                    → kube-controller-manager
├── Deleted node objects linger, no GC            → kube-controller-manager
├── Services get no endpoints anywhere            → kube-controller-manager (endpoints controller)
└── Everything intermittently 500s                → etcd
```

```
kubectl -n kube-system logs kube-scheduler-<node>
kubectl -n kube-system logs kube-controller-manager-<node>
kubectl -n kube-system logs etcd-<node> | tail -20
```

Why: "created but never acted on" is the controller-manager's signature; "acted on but never placed" is the scheduler's. Split the two by asking whether the *child object* exists.

Verify etcd directly when you suspect it:

```
ETCDCTL_API=3 etcdctl endpoint health \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

Why: etcd speaks mTLS only; without all three cert flags you get a connection error that looks like an outage but is your own client.

Tree 5 — Service / networking symptoms

```
"I can't reach the service"
├── kubectl get endpoints <svc>  (or: kubectl get endpointslice -l kubernetes.io/service-name=<svc>)
│   ├── <none> ► the Service selects nothing
│   │   ├── selector ≠ pod labels        → fix one of them
│   │   ├── pods exist but are 0/1 READY → readiness probe (Tree 1); unready pods are excluded
│   │   └── pods are in another namespace → Services are namespaced
│   └── endpoints present ► selection is fine, move on
├── endpoints present but connection refused
│   ├── targetPort ≠ containerPort the app listens on
│   └── app bound to 127.0.0.1 instead of 0.0.0.0
├── works by IP, fails by name ► DNS
│   ├── kubectl -n kube-system get pods -l k8s-app=kube-dns   → CoreDNS running?
│   └── kubectl -n kube-system logs -l k8s-app=kube-dns
└── works within namespace, fails across ► NetworkPolicy
    └── kubectl get netpol -A
```

```
kubectl get svc,endpoints <svc> -n <ns>
kubectl get pods -n <ns> --show-labels
kubectl run tmp --image=busybox:1.36 --rm -it --restart=Never -- \
  sh -c 'nslookup <svc>.<ns>.svc.cluster.local; wget -qO- --timeout=2 <svc>.<ns>:80'
```

Why: an empty `ENDPOINTS` column proves the fault is *label selection or readiness*, not routing — which collapses the search space immediately. The throwaway Pod tests DNS and connectivity from inside the cluster network, where the Service actually exists; from a node, ClusterIP may not resolve at all.

DNS name shape, which you must recall cold:

```
<service>.<namespace>.svc.cluster.local          # Services
<pod-ip-with-dashes>.<namespace>.pod.cluster.local
```

Test whether a NetworkPolicy is the cause by reading its `podSelector` — remember an empty `podSelector: {}` selects **every** Pod in that namespace, and a policy with `policyTypes: [Ingress]` and no `ingress:` rules denies everything.

Monitoring resource usage

```
kubectl top nodes
kubectl top pods -A --sort-by=memory
kubectl top pod <pod> -n <ns> --containers
```

Why: `top` reads live usage from the metrics-server. `describe node` shows *requests* (what the scheduler reserved) — different number, different question. If `top` errors with `Metrics API not available`, metrics-server isn't installed; that's the answer, not a failure.

Container output streams

```
kubectl logs <pod> -n <ns> -f                  # follow
kubectl logs <pod> -n <ns> --previous          # the instance that crashed
kubectl logs <pod> -n <ns> --all-containers    # every container in the Pod
kubectl logs -l app=web -n <ns> --tail=50      # by label, across Pods
kubectl logs <pod> -n <ns> --since=10m
kubectl logs <pod> -n <ns> > /opt/answer.txt   # tasks often want the output written to a file
```

Why: the kubelet captures each container's stdout/stderr to disk on the node; `kubectl logs` reads that, so it works even when the container has exited — but only for the current and previous instance.

> **Trap:** when a task says "write the log lines containing X to /opt/file", do it with `grep` and check the file: `kubectl logs <pod> | grep X > /opt/file && cat /opt/file`. Silent empty files are a common own-goal.

Verification habit

Never leave a task without one of these coming back correct:

```
kubectl get pod <pod> -n <ns>                    # Running, 1/1
kubectl get pvc -n <ns>                          # Bound
kubectl get endpoints <svc> -n <ns>              # non-empty
kubectl get nodes                                # Ready
kubectl -n <ns> rollout status deploy/<d>        # successfully rolled out
```

Cluster Architecture, Installation and Configuration — 25%

Second-largest domain and the one with the most *deterministic recipes* on the exam. etcd restore and `kubeadm upgrade` are near-guaranteed appearances, both are all-or-nothing, and both fail on **ordering**, not on understanding. Drill the sequences until they're cold recall.

Curriculum competencies: RBAC · prepare underlying infrastructure · create and manage clusters using kubeadm · manage the cluster lifecycle · implement and configure a highly-available control plane · use Helm and Kustomize to install cluster components · understand extension interfaces (CNI, CSI, CRI, etc.) · understand CRDs, install and configure operators.

Tree 1 — "X can't do Y" → RBAC

```
Permission problem
├── Who is the subject?
│   ├── a ServiceAccount → system:serviceaccount:<ns>:<name>
│   ├── a user           → the CN of their client certificate
│   └── a group          → the O of their client certificate
├── Is the resource namespaced?   (kubectl api-resources --namespaced=true)
│   ├── yes → Role + RoleBinding   (scoped to one namespace)
│   └── no  → ClusterRole + ClusterRoleBinding   (nodes, PVs, namespaces, CRDs…)
└── Special case: namespaced permission, reusable definition
    → ClusterRole + RoleBinding   (grants the ClusterRole only inside that namespace)
```

The whole domain in four imperative commands — never author RBAC YAML by hand:

```
kubectl create role pod-reader --verb=get,list,watch --resource=pods -n dev
kubectl create rolebinding read-pods --role=pod-reader --serviceaccount=dev:app-sa -n dev

kubectl create clusterrole node-reader --verb=get,list,watch --resource=nodes
kubectl create clusterrolebinding read-nodes --clusterrole=node-reader --user=jane
```

Why it works: `create role` / `create clusterrole` take `--verb` and `--resource` as repeatable comma lists and emit correct `apiGroups` for you — which is the field people get wrong by hand (`""` for core, `apps` for Deployments, `rbac.authorization.k8s.io` for RBAC itself).

Binding subject flags, which is where the typos live:

Subject	Flag
ServiceAccount	`--serviceaccount=<namespace>:<name>`
User	`--user=<name>`
Group	`--group=<name>`

Always verify with the impersonation check, not by reading YAML:

```
kubectl auth can-i list pods -n dev --as=system:serviceaccount:dev:app-sa
kubectl auth can-i --list -n dev --as=system:serviceaccount:dev:app-sa
kubectl auth can-i '*' '*' --as=jane          # is this subject cluster-admin?
```

Why: `auth can-i --as` asks the API server's own authorizer. It is the same code path the real request takes, so a `yes` is proof and a `no` tells you the binding didn't land.

Sub-resources and named objects, when a task is deliberately narrow:

```
kubectl create role log-reader --verb=get --resource=pods/log -n dev
kubectl create role cm-editor --verb=get,update --resource=configmaps \
  --resource-name=app-config -n dev
```

Why: `--resource-name` restricts the rule to a named object, and `pods/log` is a distinct resource from `pods` — a role granting `pods` does **not** grant `kubectl logs`.

Grant a Pod an identity:

```
kubectl create serviceaccount app-sa -n dev
kubectl set serviceaccount deploy/web app-sa -n dev
```

> **Trap:** RBAC is purely additive — there are no deny rules. If a subject can still do the thing, look for a *second* binding (often to `system:authenticated` or a group) rather than trying to subtract from the one you found.

Tree 2 — etcd backup and restore

The single highest points-per-minute task on the exam. Get it under six minutes.

```
etcd task
├── "take a snapshot"  → snapshot save (cluster keeps running; nothing else to do)
└── "restore from <file>"
    ├── 1. restore the snapshot to a NEW data dir
    ├── 2. point the static Pod at that dir   ← the step everyone forgets
    │      /etc/kubernetes/manifests/etcd.yaml:
    │        --data-dir=<new dir>            (command flag)
    │        volumes[etcd-data].hostPath.path=<new dir>   ← and this one
    └── 3. wait for the kubelet to recreate the Pod, then verify
```

Backup

```
ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /opt/etcd-backup.db

etcdctl --write-out=table snapshot status /opt/etcd-backup.db   # verify
```

Why: etcd only speaks mutual TLS, so all three of `--cacert/--cert/--key` are mandatory; without them the failure looks like an outage but is your own client being rejected.

> **Don't memorise the cert paths — read them off the running Pod:**
>
> ```
> grep -E 'cert-file|key-file|trusted-ca-file|data-dir|listen-client-urls' \
>   /etc/kubernetes/manifests/etcd.yaml
> ```
>
> That file is the source of truth for this cluster and takes ten seconds.

Restore

```
# 1. restore into a new directory (never over the live one)
sudo ETCDCTL_API=3 etcdctl snapshot restore /opt/etcd-backup.db \
  --data-dir=/var/lib/etcd-restore
# etcd 3.5+ prefers: sudo etcdutl snapshot restore /opt/etcd-backup.db --data-dir=/var/lib/etcd-restore

# 2. repoint the static Pod
sudo cp /etc/kubernetes/manifests/etcd.yaml /tmp/etcd.yaml.bak
sudo vi /etc/kubernetes/manifests/etcd.yaml
#   spec.containers[0].command:  --data-dir=/var/lib/etcd-restore
#   spec.volumes[name=etcd-data].hostPath.path: /var/lib/etcd-restore

# 3. the kubelet recreates the Pod on file change — wait, don't restart anything
sudo crictl ps | grep etcd
kubectl get pods -A
```

Why the second edit matters: `--data-dir` is a path *inside the container*. The `etcd-data` volume is what maps that path to the host. Change only the flag and etcd reads an empty directory; change only the volume and it reads the old data. Restores "succeed" and serve stale state precisely because one of the two was missed.

Why no restart command: etcd is a static Pod. The kubelet watches `/etc/kubernetes/manifests/` and recreates the Pod when the file's mtime changes. If it seems stuck, `mv` the manifest out of the directory, wait for the container to disappear, and `mv` it back.

Tree 3 — kubeadm cluster lifecycle

Upgrade (the order is the whole exam question)

Control-plane node first, always.

```
# from a machine with kubectl:
kubectl drain cp-1 --ignore-daemonsets

# on cp-1 — update the package repo to the target MINOR version first (pkgs.k8s.io is per-minor)
sudo sed -i 's|v1.34|v1.35|' /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-mark unhold kubeadm
sudo apt-get install -y kubeadm=1.35.0-1.1
sudo apt-mark hold kubeadm
kubeadm version

sudo kubeadm upgrade plan                 # shows what it will do; reads the target version
sudo kubeadm upgrade apply v1.35.0        # control plane ONLY on the first node

sudo apt-mark unhold kubelet kubectl
sudo apt-get install -y kubelet=1.35.0-1.1 kubectl=1.35.0-1.1
sudo apt-mark hold kubelet kubectl
sudo systemctl daemon-reload
sudo systemctl restart kubelet

kubectl uncordon cp-1
```

Then each worker:

```
kubectl drain worker-1 --ignore-daemonsets --delete-emptydir-data
# on worker-1:
sudo sed -i 's|v1.34|v1.35|' /etc/apt/sources.list.d/kubernetes.list && sudo apt-get update
sudo apt-mark unhold kubeadm && sudo apt-get install -y kubeadm=1.35.0-1.1 && sudo apt-mark hold kubeadm
sudo kubeadm upgrade node                 # NOT "apply" — workers only refresh local config
sudo apt-mark unhold kubelet kubectl
sudo apt-get install -y kubelet=1.35.0-1.1 kubectl=1.35.0-1.1
sudo apt-mark hold kubelet kubectl
sudo systemctl daemon-reload && sudo systemctl restart kubelet
kubectl uncordon worker-1
```

Why this order: `kubeadm` is only an installer — upgrading the binary changes nothing until `upgrade apply` rewrites the static Pod manifests. The kubelet is a *separate* package that `kubeadm` never touches, which is why it is upgraded and restarted by hand afterwards. `apt-mark hold` exists because the packages are pinned to stop accidental upgrades; forgetting `unhold` makes `apt-get install` silently keep the old version.

> **Traps:** you may only upgrade **one minor version at a time** (1.33 → 1.34 → 1.35, never 1.33 → 1.35). `upgrade apply` runs on the first control-plane node only; every other node — control plane or worker — uses `upgrade node`. And the `pkgs.k8s.io` apt repository URL is per-minor-version, so skipping the `sed` makes every `apt-get install` fail to find the version you asked for.

Join a new node

```
# on a control-plane node:
kubeadm token create --print-join-command
# on the new node, run what it printed:
sudo kubeadm join <cp-endpoint>:6443 --token <t> --discovery-token-ca-cert-hash sha256:<hash>
```

Why: join tokens expire (24h by default), so the stored one from install time is usually dead; `--print-join-command` mints a fresh token and computes the CA hash in one shot.

Remove a node

```
kubectl drain worker-2 --ignore-daemonsets --delete-emptydir-data --force
kubectl delete node worker-2
# on worker-2:
sudo kubeadm reset -f && sudo rm -rf /etc/cni/net.d $HOME/.kube/config
```

Why: `drain` evicts workloads and cordons; `delete node` removes the API object; `kubeadm reset` cleans the local state so the machine could rejoin. Doing them out of order strands Pods.

Cordon vs. drain

```
kubectl cordon <node>     # mark unschedulable; running Pods stay
kubectl drain <node>      # cordon + evict
kubectl uncordon <node>   # back into rotation
```

Drain flags you will need, and why:

Flag	Needed when
`--ignore-daemonsets`	almost always — DaemonSet Pods can't be evicted, only ignored
`--delete-emptydir-data`	a Pod uses an `emptyDir`; refuses without it
`--force`	bare Pods with no controller; they will be deleted, not rescheduled

Tree 4 — Highly-available control plane

```
HA topology
├── Stacked etcd     → etcd runs as a static Pod on each control-plane node (kubeadm default)
│                      simpler; losing a node loses both an apiserver and an etcd member
└── External etcd    → etcd on its own machines
                       more machines; control plane and storage fail independently
```

Both need a **load balancer in front of :6443** and an odd member count.

```
# first control-plane node
sudo kubeadm init \
  --control-plane-endpoint "k8s-api.example.com:6443" \
  --upload-certs \
  --pod-network-cidr=10.244.0.0/16

# additional control-plane nodes (note the extra two flags)
sudo kubeadm join k8s-api.example.com:6443 --token <t> \
  --discovery-token-ca-cert-hash sha256:<hash> \
  --control-plane --certificate-key <key>
```

Why `--control-plane-endpoint` must be set at init time: it bakes the LB name into every generated kubeconfig and certificate SAN. Add it later and you're regenerating the PKI. `--upload-certs` stores the CA material in a `kubeadm-certs` Secret so joining control planes can pull it — that Secret **expires after two hours**; regenerate with `sudo kubeadm init phase upload-certs --upload-certs`.

Quorum arithmetic — memorise it: an etcd cluster of *n* members tolerates `(n-1)/2` failures.

Members	Tolerates	Note
1	0	not HA
3	1	the standard
5	2	large clusters
4	1	same tolerance as 3, more to break — **never use even numbers**

```
ETCDCTL_API=3 etcdctl member list --write-out=table \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

Tree 5 — Install cluster components: Helm and Kustomize

New in the post-2025 curriculum, and a reported weak spot. Both are about *installing things into the cluster*, so expect "install component X with values Y" rather than authoring charts.

Helm

```
helm repo add <name> <url> && helm repo update
helm search repo <name> --versions
helm show values <name>/<chart> > /tmp/values.yaml     # discover what's configurable
helm install <release> <name>/<chart> -n <ns> --create-namespace \
  --set key=value -f /tmp/values.yaml
helm upgrade --install <release> <name>/<chart> -n <ns> --version 1.2.3
helm list -A                                           # every release, all namespaces
helm history <release> -n <ns>
helm rollback <release> <revision> -n <ns>
helm uninstall <release> -n <ns>
helm template <release> <name>/<chart> | kubectl apply -f -   # render without Tiller-era magic
```

Why `helm show values` first: the chart's own defaults file names every key you're allowed to `--set`. Guessing key paths is the main way Helm tasks are lost.

Why `--version` matters: `helm install` takes the newest chart by default; a task naming a version fails silently without it.

Kustomize (built into kubectl)

```
kubectl kustomize ./overlays/prod          # render to stdout — always look before applying
kubectl apply -k ./overlays/prod
kubectl delete -k ./overlays/prod
```

A minimal `kustomization.yaml`:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: prod
resources:
  - ../../base
images:
  - name: nginx
    newTag: "1.27"
configMapGenerator:
  - name: app-config
    literals:
      - LOG_LEVEL=debug
patches:
  - path: replicas-patch.yaml
    target:
      kind: Deployment
      name: web
```

Why `-k` and not `-f`: `-k` tells kubectl to run the kustomize build first. `kubectl apply -f kustomization.yaml` tries to apply the kustomization file itself as an object and fails.

Tree 6 — Extension interfaces (CNI, CSI, CRI)

```
Which interface?
├── CRI — how the kubelet runs containers      → containerd / CRI-O ; crictl ; /etc/crictl.yaml
├── CNI — how Pods get network                 → /etc/cni/net.d/*.conflist ; /opt/cni/bin ; Calico/Cilium/Flannel
└── CSI — how volumes are attached/mounted     → CSIDriver / CSINode objects ; StorageClass provisioner
```

```
# CRI
sudo crictl version && sudo crictl info | head -20
sudo crictl ps -a && sudo crictl pods && sudo crictl images
sudo crictl logs <container-id>
cat /etc/crictl.yaml                       # runtime-endpoint: unix:///run/containerd/containerd.sock

# CNI
ls /etc/cni/net.d/ && ls /opt/cni/bin/
kubectl get pods -n kube-system -l k8s-app=calico-node -o wide

# CSI
kubectl get csidrivers
kubectl get csinodes
kubectl get storageclass -o custom-columns=NAME:.metadata.name,PROV:.provisioner
```

Why `crictl` and not `docker`: dockershim was removed in 1.24; the kubelet talks CRI to containerd. `crictl` is the only way to see and log containers when the API server is down — which is exactly when you need it.

Why an empty `/etc/cni/net.d` matters: with no CNI config the kubelet reports `container runtime network not ready`, the node goes NotReady, and every Pod sticks in `ContainerCreating`. That symptom chain is worth recognising instantly.

Tree 7 — CRDs and operators

```
kubectl get crds
kubectl api-resources --api-group=<group>            # what did this CRD add?
kubectl explain <kind> --recursive | head -40        # the whole schema, offline
kubectl get <plural>.<group> -A                      # list instances of a custom resource
kubectl describe crd <name> | grep -iE 'scope|versions|kind'
```

Why: after installing an operator, everything you need is discoverable from the cluster itself. `api-resources` gives you the short name, the group, and whether the kind is namespaced; `explain --recursive` gives you the field names without a browser. Together they answer almost any "create a resource of this custom kind" task.

Installing an operator is normally one of:

```
kubectl apply -f https://<vendor>/operator.yaml      # bundle: CRDs + RBAC + Deployment
helm install <rel> <repo>/<operator> -n <ns> --create-namespace
```

Then confirm in this order — the sequence is the answer to "the operator isn't working":

```
kubectl get crds | grep <group>                    # 1. CRDs registered?
kubectl get pods -n <operator-ns>                  # 2. controller running?
kubectl logs deploy/<operator> -n <operator-ns>    # 3. what is it complaining about?
kubectl get <customkind> -A                        # 4. does it act on instances?
```

> **Trap:** deleting a CRD deletes **every custom resource of that kind**, cluster-wide, immediately. There is no confirmation.

Verification habit

```
kubectl get nodes                                  # all Ready, all on the target version
kubectl get pods -n kube-system                    # control plane healthy
kubectl auth can-i <verb> <resource> --as=<subject>  # RBAC actually landed
helm list -A                                       # release deployed, right chart version
etcdctl --write-out=table snapshot status <file>   # backup is readable
```

Services and Networking — 20%

The v1.35 curriculum prints this domain's heading as "*Servicing and Networking*"; the weight is 20% and the competencies are service networking. Gateway API is new here and is examinable **alongside** Ingress, not instead of it.

Curriculum competencies: understand connectivity between Pods · define and enforce Network Policies · use ClusterIP, NodePort, LoadBalancer service types and endpoints · use the Gateway API to manage Ingress traffic · know how to use Ingress controllers and Ingress resources · understand and use CoreDNS.

Tree 1 — Expose a workload

```
What has to reach it?
├── another Pod, same cluster            → ClusterIP (default)
├── something outside, no cloud LB       → NodePort (<node-ip>:30000-32767)
├── something outside, cloud provider    → LoadBalancer
├── HTTP(S), name/path routing, many apps → Ingress + an Ingress controller
└── HTTP(S) with role separation / richer routing → Gateway API
```

```
kubectl expose deploy web --port=80 --target-port=8080           # ClusterIP
kubectl expose deploy web --port=80 --target-port=8080 --type=NodePort
kubectl create service nodeport web --tcp=80:8080 --node-port=30080
kubectl expose deploy web --port=80 --type=LoadBalancer
```

Why `expose` over authoring: it copies the Deployment's own labels into the Service selector, which is the field most often typo'd by hand. `--port` is what the Service listens on; `--target-port` is the container's port.

Confirm the selector actually matched something — this is the whole game:

```
kubectl get endpoints web
kubectl get pods --show-labels
```

Why: a Service with an empty `ENDPOINTS` list is selecting nothing (label mismatch) or selecting only **unready** Pods. Endpoints are populated from Pods that are `Ready`, so a failing readiness probe presents as a networking bug.

Tree 2 — NetworkPolicy

```
Policy question
├── Nothing selects the Pod            → all traffic allowed (default is open)
├── A policy selects the Pod
│   ├── policyTypes includes Ingress   → only listed ingress rules allowed; everything else denied
│   └── policyTypes includes Egress    → only listed egress rules allowed; everything else denied
└── Multiple policies select it        → rules are UNIONed (additive; there is no deny rule)
```

Default-deny for a namespace — the shape you must be able to write cold:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata: { name: default-deny, namespace: prod }
spec:
  podSelector: {}                 # {} = every Pod in this namespace
  policyTypes: [Ingress, Egress]  # no rules listed = deny all in those directions
```

Allow one app to reach another on one port:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata: { name: allow-web-to-db, namespace: prod }
spec:
  podSelector:
    matchLabels: { app: db }
  policyTypes: [Ingress]
  ingress:
    - from:
        - podSelector:
            matchLabels: { app: web }
        - namespaceSelector:
            matchLabels: { kubernetes.io/metadata.name: staging }
      ports:
        - protocol: TCP
          port: 5432
```

> **The list-indentation trap.** Under `from:`, two entries at the same level are **OR** ("web Pods, or anything in staging"). Putting `namespaceSelector` and `podSelector` under a *single* `-` makes them **AND** ("web Pods that are in staging"). One dash changes the meaning. Read the task's wording carefully and check your indentation before applying.

```
kubectl get netpol -A
kubectl describe netpol <name> -n <ns>
kubectl explain networkpolicy.spec.ingress --recursive     # faster than the docs tab
```

Why: NetworkPolicy is enforced by the CNI plugin. If the CNI doesn't implement it (plain Flannel, for example), your policy applies cleanly to the API and does nothing — so verify with a connection test, not with `get netpol`.

Test it:

```
kubectl run probe --image=busybox:1.36 --rm -it --restart=Never -n prod -- \
  sh -c 'wget -qO- --timeout=2 db:5432 || echo BLOCKED'
```

Tree 3 — DNS / CoreDNS

```
Name resolution failing?
├── nslookup fails for everything      → CoreDNS Pods down, or kubelet clusterDNS wrong
├── fails only for one Service         → wrong name, or the Service has no endpoints
├── fails only cross-namespace         → short name used; needs <svc>.<ns>
└── external names fail, internal work → CoreDNS forward/upstream broken
```

```
kubectl -n kube-system get pods -l k8s-app=kube-dns
kubectl -n kube-system get svc kube-dns
kubectl -n kube-system get cm coredns -o yaml            # the Corefile
kubectl -n kube-system logs -l k8s-app=kube-dns --tail=30

kubectl run dnsutils --image=busybox:1.36 --rm -it --restart=Never -- \
  nslookup web.prod.svc.cluster.local
```

Why the fully-qualified name in tests: a Pod's `/etc/resolv.conf` has a `search` list, so short names resolve only from namespaces the search list covers. Testing with the FQDN separates "DNS is broken" from "you used the wrong short name".

Names to recall cold:

```
<service>.<namespace>.svc.cluster.local
<statefulset-pod>.<headless-service>.<namespace>.svc.cluster.local
<pod-ip-dashed>.<namespace>.pod.cluster.local
```

A **headless** Service (`clusterIP: None`) returns Pod IPs instead of a virtual IP — that's how StatefulSet members get stable, individually-addressable names:

```
kubectl create service clusterip db --clusterip="None" --tcp=5432:5432
```

Tree 4 — Ingress

```
kubectl create ingress web --rule="app.example.com/*=web:80" \
  --class=nginx -n prod
kubectl create ingress multi \
  --rule="example.com/api*=api-svc:8080" \
  --rule="example.com/*=web-svc:80" --class=nginx
kubectl get ingress -A
kubectl describe ingress web -n prod
```

Why the imperative form matters: `kubectl create ingress` with `--rule` is far quicker than authoring the nested `spec.rules[].http.paths[]` tree, and it fills in `pathType` for you.

Rule syntax: `host/path=service:port`. A `*` suffix on the path means `pathType: Prefix`; without it, `Exact`.

```
Ingress not working?
├── kubectl get ingress → no ADDRESS      → no controller is running / wrong ingressClassName
├── ADDRESS present, 404                  → host header or path doesn't match a rule
├── ADDRESS present, 503                  → backend Service has no endpoints (Tree 1)
└── controller logs                       → kubectl logs -n ingress-nginx deploy/ingress-nginx-controller
```

Why: an Ingress **object** is inert. A controller must be watching for it; without one, nothing reconciles and the object sits there looking correct.

Tree 5 — Gateway API

The successor to Ingress, and new to the CKA curriculum. The point is **role separation**: three resources instead of one.

Resource	Owned by	Says
`GatewayClass`	infrastructure provider	which controller implements gateways
`Gateway`	cluster operator	listeners: ports, protocols, hostnames, TLS
`HTTPRoute`	application team	matches and backends, attached to a Gateway

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata: { name: prod-gw, namespace: infra }
spec:
  gatewayClassName: nginx
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes:
        namespaces: { from: All }
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata: { name: web, namespace: prod }
spec:
  parentRefs:
    - name: prod-gw
      namespace: infra
  hostnames: ["app.example.com"]
  rules:
    - matches:
        - path: { type: PathPrefix, value: /api }
      backendRefs:
        - name: api-svc
          port: 8080
```

```
kubectl get gatewayclass,gateway,httproute -A
kubectl describe gateway prod-gw -n infra      # Status.Conditions: Accepted / Programmed
kubectl describe httproute web -n prod         # Status.Parents: did the Gateway accept it?
kubectl api-resources --api-group=gateway.networking.k8s.io
```

Why the status fields are the debugging tool: unlike Ingress, Gateway API reports acceptance explicitly. A route that didn't attach says so in `status.parents[].conditions` — usually `NotAllowedByListeners` (the Gateway's `allowedRoutes.namespaces` doesn't permit your namespace) or `NoMatchingParent`.

> Gateway API CRDs are **not** installed by default. `kubectl get gateway` returning `the server doesn't have a resource type "gateway"` means the CRDs are missing, not that your YAML is wrong.

Verification habit

```
kubectl get svc,endpoints -n <ns>
kubectl get ingress,gateway,httproute -A
kubectl run probe --image=busybox:1.36 --rm -it --restart=Never -- \
  sh -c 'nslookup <svc>.<ns>; wget -qO- --timeout=2 <svc>.<ns>:<port>'
```

Workloads and Scheduling — 15%

Almost entirely imperative-command territory. If you're authoring YAML here, you're losing time.

Curriculum competencies: application deployments, rolling updates and rollbacks · ConfigMaps and Secrets · workload autoscaling · primitives for robust self-healing deployments · Pod admission and scheduling (limits, node affinity, etc.).

Tree 1 — Create and roll

```
kubectl create deploy web --image=nginx:1.26 --replicas=3
kubectl set image deploy/web nginx=nginx:1.27          # container-name=new-image
kubectl rollout status deploy/web
kubectl rollout history deploy/web
kubectl rollout undo deploy/web                        # previous revision
kubectl rollout undo deploy/web --to-revision=2
kubectl rollout restart deploy/web                     # re-create Pods, same spec
kubectl scale deploy/web --replicas=5
```

Why `set image` and not `edit`: it patches only the container image and triggers exactly one rollout. `kubectl edit` on a bad day leaves you in vim with an invalid document and a Deployment that didn't change.

Why `rollout restart` exists: changing a mounted ConfigMap doesn't restart Pods. `restart` stamps a new annotation on the Pod template, which is a template change, which rolls the Deployment.

```
Rollout stuck?
├── kubectl rollout status hangs
│   ├── new Pods Pending          → scheduling (see Troubleshooting Tree 1)
│   ├── new Pods ImagePullBackOff → bad tag; rollout undo
│   └── new Pods Running, 0/1     → readiness probe never passes → maxUnavailable blocks progress
└── kubectl describe deploy → "ProgressDeadlineExceeded" after 600s
```

Strategy knobs, which a task will name explicitly:

```
kubectl patch deploy web -p \
  '{"spec":{"strategy":{"rollingUpdate":{"maxSurge":1,"maxUnavailable":0}}}}'
```

Why `maxUnavailable: 0`: it forces a new Pod to become Ready **before** an old one is removed — the zero-downtime shape. `maxSurge: 0` with `maxUnavailable: 1` is the opposite trade (no extra capacity, brief reduction).

Tree 2 — Configuration: ConfigMaps and Secrets

```
kubectl create configmap app-cfg --from-literal=LOG_LEVEL=debug --from-literal=MODE=prod
kubectl create configmap app-cfg --from-file=./app.properties
kubectl create configmap app-cfg --from-env-file=./app.env

kubectl create secret generic db-cred \
  --from-literal=username=admin --from-literal=password=s3cr3t
kubectl create secret docker-registry regcred \
  --docker-server=registry.io --docker-username=u --docker-password=p
kubectl create secret tls web-tls --cert=tls.crt --key=tls.key
```

Consume them — `--from-file` vs `--from-env-file` is the distinction that gets missed:

Flag	Result
`--from-file=app.properties`	**one** key `app.properties` whose value is the whole file
`--from-env-file=app.env`	**one key per line** of `KEY=value`

```
kubectl set env deploy/web --from=configmap/app-cfg      # all keys as env vars
kubectl set env deploy/web --from=secret/db-cred
kubectl set env deploy/web LOG_LEVEL=debug
kubectl set env deploy/web LOG_LEVEL-                    # trailing dash removes it
```

Why `set env --from`: it writes `envFrom` correctly in one command. Hand-authoring `envFrom.configMapRef.name` is a common misspelling (`configMapRef`, capital M).

Reading a Secret back:

```
kubectl get secret db-cred -o jsonpath='{.data.password}' | base64 -d
```

Why: Secret values are base64 in the API, not encrypted. A task asking you to "find the password" wants this pipe.

Mounting as a volume is the one place YAML is unavoidable:

```
spec:
  containers:
    - name: web
      volumeMounts:
        - name: cfg
          mountPath: /etc/config
          readOnly: true
  volumes:
    - name: cfg
      configMap:
        name: app-cfg
```

Why mount instead of env: mounted ConfigMap keys update in place (eventually) when the ConfigMap changes; environment variables are fixed at container start.

Tree 3 — Autoscaling

```
kubectl autoscale deploy web --min=2 --max=10 --cpu-percent=70
kubectl get hpa
kubectl describe hpa web            # the Events + Conditions tell you why it isn't scaling
```

```
HPA not scaling?
├── TARGETS shows <unknown>/70%
│   ├── metrics-server not installed  → kubectl top pods also fails
│   └── the Deployment has no CPU *requests* → HPA computes a percentage OF the request
├── TARGETS fine, replicas pinned at max/min → hit the bound; that's correct behaviour
└── scaling down is slow → stabilization window (5 min default), also correct
```

Why requests are mandatory: `--cpu-percent=70` means "70% of the container's CPU **request**". With no request there is no denominator, and the HPA reports `<unknown>` forever.

```
kubectl set resources deploy web --requests=cpu=100m,memory=128Mi --limits=cpu=500m,memory=256Mi
```

Vertical scaling is manual here; the exam's autoscaling competency is HPA, plus knowing that a **Cluster Autoscaler** adds nodes and is a separate component.

Tree 4 — Self-healing primitives

Primitive	Guarantees
`Deployment` → `ReplicaSet`	N stateless replicas, rolling updates, rollback history
`StatefulSet`	stable network identity + stable storage, ordered create/delete
`DaemonSet`	exactly one Pod per (matching) node, including new nodes
`Job`	run to completion, `completions` / `parallelism`, retries via `backoffLimit`
`CronJob`	Jobs on a schedule
`PodDisruptionBudget`	floor on availability during *voluntary* disruption (drain)

```
kubectl create job pi --image=perl -- perl -Mbignum=bpi -wle 'print bpi(200)'
kubectl create cronjob report --image=busybox --schedule="*/5 * * * *" -- /bin/sh -c 'date'
kubectl create job manual-run --from=cronjob/report        # trigger a CronJob now
kubectl create poddisruptionbudget web-pdb --selector=app=web --min-available=2
```

Why the PDB matters on this exam: it is what makes `kubectl drain` block. A drain that hangs forever with `cannot evict pod as it would violate the disruption budget` is a PDB, not a bug.

Probes — the three, and what each one does:

Probe	Failure means
`livenessProbe`	container is restarted
`readinessProbe`	Pod removed from Service endpoints (not restarted)
`startupProbe`	disables the other two until it first succeeds; for slow starters

Why startupProbe exists: without it, a slow-booting app gets killed by its own liveness probe in a loop that looks exactly like a crash.

Tree 5 — Scheduling and admission

```
"Place this Pod on/off node X"
├── hard requirement, simple label     → nodeSelector
├── hard requirement, expressions      → requiredDuringSchedulingIgnoredDuringExecution
├── soft preference                    → preferredDuringSchedulingIgnoredDuringExecution
├── keep Pods AWAY from a node         → taint the node (+ toleration to allow back)
├── co-locate / separate Pods          → podAffinity / podAntiAffinity + topologyKey
├── spread evenly across zones         → topologySpreadConstraints
└── bypass the scheduler entirely      → nodeName (or a static Pod)
```

```
kubectl label node worker-1 disktype=ssd
kubectl taint node worker-1 gpu=true:NoSchedule
kubectl taint node worker-1 gpu=true:NoSchedule-          # trailing dash removes
kubectl get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
kubectl run web --image=nginx --dry-run=client -o yaml > pod.yaml   # then add the stanza
```

Taint effects, which differ in ways tasks exploit:

Effect	Does
`NoSchedule`	no new Pods without a toleration; existing Pods stay
`PreferNoSchedule`	soft version; scheduler tries to avoid
`NoExecute`	as `NoSchedule`, **and evicts** running Pods that don't tolerate

Why the distinction matters: "make the existing Pods leave" is `NoExecute`; `NoSchedule` alone will look like it did nothing.

Resource admission at the namespace level:

```
kubectl create quota dev-quota -n dev \
  --hard=cpu=4,memory=8Gi,pods=20,persistentvolumeclaims=5
kubectl describe quota -n dev
kubectl describe limitrange -n dev
```

Why a ResourceQuota can break Pod creation: once a quota sets `cpu` or `memory`, every Pod in that namespace **must** declare matching requests/limits or it is rejected at admission with a `failed quota` error. A `LimitRange` supplies defaults so that doesn't happen.

Static Pods — the scheduler is not involved at all:

```
# on the node
sudo vi /etc/kubernetes/manifests/my-static.yaml    # path from staticPodPath in kubelet config
grep staticPodPath /var/lib/kubelet/config.yaml
```

Why they matter: the kubelet runs them straight from disk, they get `-<nodename>` appended to their name, and they cannot be deleted with `kubectl` — the kubelet just recreates them. Delete the file.

Verification habit

```
kubectl rollout status deploy/<d> -n <ns>
kubectl get pods -o wide                      # landed on the intended node?
kubectl get hpa,quota,pdb -n <ns>
kubectl describe pod <p> | grep -A5 'Environment\|Mounts'
```

Storage — 10%

The smallest domain, and the one with the tightest failure mode: a PVC that stays `Pending` blocks every Pod that wants it, and the reason is always one of four things.

Curriculum competencies: implement storage classes and dynamic volume provisioning · configure volume types, access modes and reclaim policies · manage persistent volumes and persistent volume claims.

Tree 1 — PVC is Pending

```
kubectl describe pvc <name>  → read the Events
├── "no persistent volumes available for this claim and no storage class is set"
│   → no matching PV, and storageClassName is "" or absent with no default class
├── "waiting for a volume to be created ... or by the system administrator"
│   → dynamic provisioning: the StorageClass's provisioner isn't running
├── "waiting for first consumer to be created before binding"
│   → volumeBindingMode: WaitForFirstConsumer — NORMAL. Bind happens when a Pod uses it.
└── bound to nothing, a PV exists but is Available
    → a mismatch: size, accessModes, or storageClassName
```

```
kubectl get pvc,pv
kubectl describe pvc <name> -n <ns>
kubectl get storageclass
kubectl get sc -o custom-columns=NAME:.metadata.name,PROV:.provisioner,BIND:.volumeBindingMode,RECLAIM:.reclaimPolicy
```

Why the four-way check: a PV binds to a PVC only when **capacity ≥ request**, **access modes are a superset**, and **storageClassName matches exactly** (including the empty string). Any one mismatch leaves both objects sitting there looking healthy.

> **Trap:** `storageClassName: ""` means "explicitly no class, static binding only" and is **not** the same as omitting the field, which means "use the default StorageClass". If there's no default class, omitting it leaves the PVC Pending forever.

```
kubectl get sc -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.metadata.annotations.storageclass\.kubernetes\.io/is-default-class}{"\n"}{end}'
```

Tree 2 — StorageClass and dynamic provisioning

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/no-provisioner     # or a CSI driver name
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Delete
allowVolumeExpansion: true
```

Field	Choose	Because
`provisioner`	a CSI driver name	`kubernetes.io/no-provisioner` means *static only* — PVCs will never auto-provision
`volumeBindingMode`	`WaitForFirstConsumer`	delays binding until a Pod is scheduled, so the volume lands in the right zone/node
	`Immediate`	binds at PVC creation; can strand a volume in a zone with no capacity
`reclaimPolicy`	`Delete`	PV and backing storage removed when the PVC goes
	`Retain`	PV survives as `Released`; data kept, needs manual cleanup before reuse
`allowVolumeExpansion`	`true`	required before you can grow a PVC by editing its request

Why `WaitForFirstConsumer` is the safe default: with `Immediate`, the volume is created before the scheduler has picked a node. If it lands in a zone the Pod can't be scheduled into, the Pod fails with `had volume node affinity conflict` — a symptom that reads as scheduling but is caused by storage.

Switching the default class:

```
kubectl patch sc standard -p \
  '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'
kubectl patch sc fast -p \
  '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

Why both patches: two defaults is an error state — the API accepts it and provisioning becomes non-deterministic.

Tree 3 — Access modes and reclaim policies

Access mode	Short	Means
`ReadWriteOnce`	RWO	read-write by Pods on **one node**
`ReadOnlyMany`	ROX	read-only from many nodes
`ReadWriteMany`	RWX	read-write from many nodes (needs NFS/CephFS-class backing)
`ReadWriteOncePod`	RWOP	read-write by exactly **one Pod**, cluster-wide

Why RWO trips people: it is per-**node**, not per-Pod. Two Pods on the same node can both mount an RWO volume. That's `ReadWriteOncePod`'s reason to exist.

Reclaim policy	On PVC deletion
`Delete`	PV and the underlying storage are deleted
`Retain`	PV goes `Released`; data kept; not reusable until the `claimRef` is cleared
`Recycle`	deprecated — don't

Reusing a `Retain`ed PV:

```
kubectl patch pv <pv> -p '{"spec":{"claimRef": null}}'
```

Why: a `Released` PV still points at its old claim. Clearing `claimRef` returns it to `Available`.

Change a policy in place:

```
kubectl patch pv <pv> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

Tree 4 — Static PV + PVC (the classic exam shape)

No imperative command creates a PV; this is one of the few manifests worth memorising.

```
apiVersion: v1
kind: PersistentVolume
metadata: { name: pv-data }
spec:
  capacity:
    storage: 2Gi
  accessModes: [ReadWriteOnce]
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  hostPath:
    path: /mnt/data
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata: { name: pvc-data, namespace: default }
spec:
  accessModes: [ReadWriteOnce]
  storageClassName: manual
  resources:
    requests:
      storage: 1Gi
```

```
kubectl apply -f pv.yaml && kubectl get pv,pvc
```

Why it binds: 2Gi ≥ 1Gi, access modes match, and `storageClassName: manual` is identical on both. Change any one and it stops.

Mount it:

```
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - name: data
          mountPath: /usr/share/nginx/html
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: pvc-data
```

Generate the Pod skeleton rather than typing it:

```
kubectl run app --image=nginx --dry-run=client -o yaml > pod.yaml
kubectl explain pod.spec.volumes.persistentVolumeClaim      # field names, offline
```

Tree 5 — Volume types worth knowing

Type	Lifetime	Use
`emptyDir`	the Pod	scratch space, sidecar hand-off; `medium: Memory` for tmpfs
`hostPath`	the node	node-local files; what exam clusters use for PVs
`persistentVolumeClaim`	independent of the Pod	real persistence
`configMap` / `secret`	the Pod	config and credentials as files
`projected`	the Pod	several sources into one directory

Why `emptyDir` matters for drain: `kubectl drain` refuses to evict Pods with an `emptyDir` unless you pass `--delete-emptydir-data`, because the data is destroyed. That's a Cluster Architecture task failing for a Storage reason.

Expanding a volume:

```
kubectl patch pvc pvc-data -p '{"spec":{"resources":{"requests":{"storage":"5Gi"}}}}'
kubectl get pvc pvc-data -o jsonpath='{.status.capacity.storage}'
```

Why it may not take effect immediately: expansion requires `allowVolumeExpansion: true` on the StorageClass, and some drivers only finish the filesystem resize after the Pod restarts. `status.capacity` is the truth; `spec.resources` is the request.

Verification habit

```
kubectl get pvc -n <ns>                    # STATUS must be Bound
kubectl get pv                             # correct CLAIM, correct RECLAIM POLICY
kubectl describe pod <p> | grep -A5 Mounts  # the volume is actually mounted where asked
kubectl exec <p> -- df -h /path            # and it is the right size
```

Exam strategy

Derived from the community retrospectives. Nearly every reported failure is a process failure, not a knowledge failure.

The arithmetic that drives everything

120 minutes ÷ ~17 tasks ≈ **7 minutes per task**, and that 7 minutes includes reading the prompt, switching context, and verifying. It is not 7 minutes of typing.

Passing is 66%. You do **not** need every task. You need most of them done correctly, which is a different objective from every task done perfectly — and optimising for the second is the most-reported cause of a second failed attempt.

The opening

1. **Before task 1:** aliases, completion, `$do`, `.vimrc`. Two minutes, saves twenty.
2. **Read every task first**, noting its point weight. Weights are shown and they are not equal.
3. **Order by points ÷ estimated minutes.** Bank cheap, heavy tasks; defer expensive ones.

Per task, in this order

```
1. kubectl config use-context <given>   ← FIRST ACTION. Then current-context to confirm.
2. Read the task twice. Note the namespace. Note the exact names asked for.
3. Generate, don't author:  kubectl create ... $do > x.yaml
4. Apply.
5. Verify with a get/describe that proves the end state, not the API call.
6. Move on. Do not polish.
```

Step 1 is the one that silently costs whole tasks

Solving on the wrong cluster scores zero and gives you **no signal at all** — your resource exists, your YAML is right, and the grader looks somewhere else. Two seconds of `current-context` eliminates the single most costly mistake reported.

The namespace is the same trap one level down. If the task names a namespace and you forget `-n`, you built the right thing in the wrong place, and `kubectl get` in your current namespace will happily show you nothing while you assume success.

Step 5 is what converts work into points

Twenty seconds. Confirm the *end state*:

You did	Verify with
created a Pod/Deployment	`kubectl get po` → `Running`, `1/1`
created a Service	`kubectl get endpoints <svc>` → non-empty
created a PVC	`kubectl get pvc` → `Bound`
fixed a node	`kubectl get nodes` → `Ready`
rolled a Deployment	`kubectl rollout status deploy/<d>`
wrote RBAC	`kubectl auth can-i ... --as=...` → `yes`
wrote a file	`cat` the file
restored etcd	`kubectl get po -A` returns the expected objects

"It applied without an error" is not verification. Objects apply cleanly and then fail to run all the time.

Flag and skip

- If you're **3 minutes in with no traction**, flag it and go.
- If reading it suggests **more than 8 minutes**, defer it on the first pass by default.
- Come back only after every cheap task is banked and verified.

The failure mode is emotional, not technical: a hard task feels like it needs finishing *now*. It doesn't. It's worth the same points at minute 100.

Things that quietly eat minutes

Time sink	Replace with
Writing YAML from scratch	`$do` generation, then edit
Browsing kubernetes.io for field names	`kubectl explain <kind>.<path> --recursive`
`kubectl edit` on a live object	`kubectl set image` / `set env` / `set resources` / `patch`
Retyping long resource names	shell completion (set it up in the first minute)
Re-reading the task for the namespace	write the namespace down before you start
Perfecting a task that already works	leave; it scores the same

Documentation you are allowed, and how to use it

One additional browser tab, official Kubernetes documentation. Bookmark the handful of pages you genuinely cannot produce cold — realistically: etcd backup/restore, kubeadm upgrade, NetworkPolicy schema, Gateway API examples, PV/PVC examples.

Everything else should come from `kubectl explain` and `kubectl <cmd> --help`, both of which are faster than a page load and always match the cluster's version.

Preparation calibration

- The **Killer.sh simulator** ships with the exam voucher: 2 sessions, 17 scenarios each. It is consistently reported as harder than the real exam. Not finishing it is not a fail signal; the value is disproportionately in the 34 hours of post-session access where you read the solutions.
- **Practise timed, always.** Untimed practice trains the wrong exam. Every practice task on this site shows a target time before you start for exactly this reason.
- **Break your own cluster weekly.** Stop a kubelet, corrupt a static Pod manifest, delete a CNI config, point etcd at an empty directory. Troubleshooting is 30% and it is the one domain that does not yield to reading.
- **Practise on Linux.** Muscle memory built on other shortcuts costs real minutes.

The evening before

- Confirm ID, webcam, and a clear desk against the current Candidate Handbook.
- Re-read your own weakest-domain notes — the dashboard surfaces them.
- Rehearse the two all-or-nothing recipes once: etcd restore, kubeadm upgrade.
- Sleep. The exam is 120 minutes of sustained attention, and every reported time-management failure gets worse when you're tired.

CKA Curriculum — v1.35

Fetched: 2026-09-02 **Curriculum document:** `CKA_Curriculum_v1.35.pdf`, a Cloud Native Computing Foundation (CNCF) publication **Source (authoritative):** https://github.com/cncf/curriculum → https://raw.githubusercontent.com/cncf/curriculum/master/CKA_Curriculum_v1.35.pdf

CNCF states in that repository: "*The document major and minor version... match the version of Kubernetes*", with the patch version representing documentation iterations for that Kubernetes version. So **curriculum v1.35 tracks Kubernetes v1.35**.

> Re-check this file's source URL before you book the exam. The curriculum is revised on the Kubernetes release cadence, and the January/February 2025 revision changed the competency list substantially even though the domain weights stayed put.

Domain weights

#	Domain	Weight
1	Troubleshooting	**30%**
2	Cluster Architecture, Installation and Configuration	**25%**
3	Services and Networking	**20%**
4	Workloads and Scheduling	**15%**
5	Storage	**10%**

Note on wording: the v1.35 PDF prints the third domain's heading as **"20% - Servicing and Networking"**. Every other CNCF/Linux Foundation surface calls it *Services and Networking*, and the competencies underneath it are service networking competencies. Treated here as a typo in the PDF; the weight (20%) is unambiguous.

Competencies, transcribed verbatim from the v1.35 PDF

30% — Troubleshooting

- Troubleshoot clusters and nodes
- Troubleshoot cluster components
- Monitor cluster and application resource usage
- Manage and evaluate container output streams
- Troubleshoot services and networking

25% — Cluster Architecture, Installation and Configuration

- Manage role based access control (RBAC)
- Prepare underlying infrastructure for installing a Kubernetes cluster
- Create and manage Kubernetes clusters using kubeadm
- Manage the lifecycle of Kubernetes clusters
- Implement and configure a highly-available control plane
- Use Helm and Kustomize to install cluster components
- Understand extension interfaces (CNI, CSI, CRI, etc.)
- Understand CRDs, install and configure operators

20% — Services and Networking ***(printed as "Servicing and Networking")***

- Understand connectivity between Pods
- Define and enforce Network Policies
- Use ClusterIP, NodePort, LoadBalancer service types and endpoints
- Use the Gateway API to manage Ingress traffic
- Know how to use Ingress controllers and Ingress resources
- Understand and use CoreDNS

15% — Workloads and Scheduling

- Understand application deployments and how to perform rolling update and rollbacks
- Use ConfigMaps and Secrets to configure applications
- Configure workload autoscaling

- Understand the primitives used to create robust, self-healing, application deployments
- Configure Pod admission and scheduling (limits, node affinity, etc.)

10% — Storage

- Implement storage classes and dynamic volume provisioning
- Configure volume types, access modes and reclaim policies
- Manage persistent volumes and persistent volume claims

What changed in the post-2025 revision (and why old guides mislead)

The domain names and percentages are the same as the pre-2025 CKA, which is exactly why stale study guides look correct and are not. The competency list gained items that did not exist in the older curriculum:

- **Helm and Kustomize** — "Use Helm and Kustomize to install cluster components" (Cluster Architecture)
- **CRDs and operators** — "Understand CRDs, install and configure operators" (Cluster Architecture)
- **Extension interfaces** — "Understand extension interfaces (CNI, CSI, CRI, etc.)" (Cluster Architecture)
- **Gateway API** — "Use the Gateway API to manage Ingress traffic" (Services and Networking)
- **Autoscaling** — "Configure workload autoscaling" (Workloads and Scheduling)

Community reports consistently flag these five as the weak spots of candidates who studied from pre-2025 material. See retrospectives.

Exam logistics

These figures are widely and consistently reported by community and vendor sources, and by the CNCF study material listed below. **They were not read from `training.linuxfoundation.org` or `docs.linuxfoundation.org` directly — both domains were unreachable from the network this site was built on.** Verify against the Linux Foundation's own Candidate Handbook and exam page before you rely on any of it.

Item	Reported value
Format	Performance-based, hands-on tasks in a live cluster environment; remote proctored
Duration	120 minutes
Tasks	15–20 (commonly reported as 16–17)
Passing score	66%
Kubernetes version	v1.35 (matches curriculum v1.35)
Permitted documentation	Official Kubernetes documentation, in one additional browser tab
Simulator	A Killer.sh exam simulator session is included with the exam purchase — 2 sessions, 17 scenarios each, 120-minute run then 36 hours of total access for reviewing solutions
Certification validity	Commonly reported as 2 years

Budget arithmetic that follows from the numbers above: **120 min ÷ ~17 tasks ≈ 7 minutes per task**, including reading the prompt, switching context, and verifying. That single number drives most of the strategy on this site.

Sources

Source	URL	Fetched	How
CNCF curriculum repository	https://github.com/cncf/curriculum	2026-09-02	direct fetch
CKA_Curriculum_v1.35.pdf	https://raw.githubusercontent.com/cncf/curriculum/master/CKA_Curriculum_v1.35.pdf	2026-09-02	direct download, text extracted
techiescamp CKA syllabus notes	https://github.com/techiescamp/cka-certification-guide/blob/main/SYLLABUS.md	2026-09-02	direct fetch
Linux Foundation CKA exam page	https://training.linuxfoundation.org/certification/certified-kubernetes-administrator-cka/	2026-09-02	**blocked by network egress; not read**
Linux Foundation CKA/CKAD/CKS FAQ	https://docs.linuxfoundation.org/tc-docs/certification/faq-cka-ckad-cks	2026-09-02	**blocked by network egress; not read**
Linux Foundation CKA program changes	https://training.linuxfoundation.org/certified-kubernetes-administrator-cka-program-changes/	2026-09-02	**blocked by network egress; not read**
Killer.sh FAQ	https://killer.sh/faq	2026-09-02	**blocked by network egress**; simulator details via search-result summary

The domain list and every competency bullet above came from the CNCF PDF itself, not from a summary. The logistics table did not.