ChatGPT

Certified Kubernetes Application Developer (CKAD) Study Guide

Introduction to Kubernetes and CKAD

Kubernetes is a container orchestration platform that manages the deployment, scaling, and operation of application containers across a cluster of servers (nodes). The CKAD certification focuses on the **application developer** perspective – using Kubernetes primitives to design and build cloud-native applications. This guide is organized by key topic areas (Pods, Deployments, Services, Volumes, etc.), with **every concept and command** from the original CKAD notes, plus expanded explanations, YAML examples, real-world analogies, **diagrams**, and exam tips.

Tip: In the CKAD exam you have access to official Kubernetes documentation. Practice using `kubectl` commands and writing YAML by hand – but remember you can always refer to docs or use generator commands (`kubectl create ... -o yaml --dry-run=client`) to speed up in the exam.

Pods (Core Concepts)

A **Pod** is the smallest deployable unit in Kubernetes – essentially a wrapper around one or more containers that run on a node [1]. Every Pod is scheduled to a **node** (a worker machine, physical or virtual) and gets its own network IP and port space. Pods encapsulate application containers, storage resources, a unique network IP, and options that govern how the containers run.

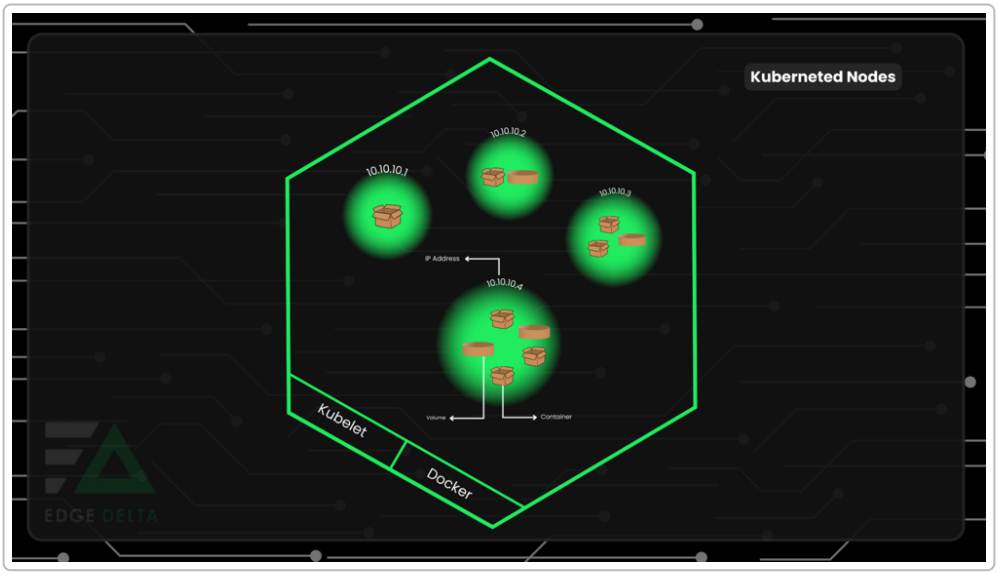

Figure: A Kubernetes Node (green hexagon) with multiple Pods inside. Each green circle represents a Pod running one or more containers (shown as box icons) and a shared volume (disk icon). All Pods on a node are managed by

the node's kubelet and container runtime (e.g. Docker). Each Pod has its own IP address [2], and containers within a Pod share the network namespace (localhost) and volumes.

In everyday terms, you can think of a Pod like a **pod of whales** or a **pea pod** – a group of containers that belong together [3]. A simpler analogy: If containers are like shipping boxes containing an app and its dependencies, then a Pod is like a **shipping pallet** that can hold one or more boxes together. The Pod ensures the boxes (containers) are kept together on the same truck (node) and shipped as a unit.

What's inside a Pod? In a single-container Pod (the most common case), the Pod just contains one main application container. In a multi-container Pod, containers share the Pod's network (they communicate via `localhost`) and can also share storage volumes. Containers in the same Pod are tightly coupled – for example, a web app container might be paired with a logging sidecar container in the same Pod. They always co-locate: if the Pod is scheduled to Node A, all containers in that Pod run on Node A. Pods are considered *ephemeral*; you should not treat any individual Pod as durable or unique. If a Pod crashes or a node dies, Kubernetes can replace the Pod with a new one (with a new IP) to maintain the desired state [4] [5].

Pod Lifecycle: A Pod runs until its process(es) end or it's explicitly destroyed. Pods are managed by higher-level controllers (like Deployments) for scaling and healing. When a Pod is terminated (e.g. the process exits or the Pod object is deleted), Kubernetes will not restart the same Pod – instead a controller would create a new Pod if needed. In summary, a Pod lives on a node until one of these happens: (1) the container process stops or crashes, (2) the Pod object is deleted, (3) the Pod is evicted due to lack of resources, or (4) the node itself fails or is stopped.

Multi-Container Pod Patterns: While most Pods have a single container, Kubernetes supports multi-container Pods for cases where containers need to work together closely. For example: - **Sidecar**: A helper container augments the main container. (Analogy: main app is a car, sidecar container is a motorcycle sidecar providing extra functionality like logging or proxying). Both share the Pod's localhost and can communicate internally. - **Adapter or Ambassador**: A container that helps the main container by adapting output or communicating with the outside world on behalf of the main app.
Containers in one Pod can talk via `localhost` since they share the same network namespace [6]. They also share storage volumes, making it easy to exchange files. Kubernetes **does not** use Pods to scale an app horizontally (that's what Deployments/ReplicaSets do). Instead, each Pod is usually one instance of your application. To scale up, you create more Pods (not add more containers to a Pod) [7] [8]. **! Gotcha:** If you need 5 instances of an application, Kubernetes will schedule 5 Pods (each perhaps with 1 container) rather than 1 Pod with 5 containers. Pods are meant to represent *one instance* of a running service [9] [10].

Pod YAML Example: Below is a basic Pod manifest. It defines a Pod running a single container (nginx). In YAML, every Kubernetes object has the 4 top-level fields: `apiVersion`, `kind`, `metadata`, and `spec` [11]. Under `spec`, the Pod defines a list of containers. In this example we also illustrate mounting a ConfigMap as a volume and setting an environment variable from a Secret (to preview ConfigMap/Secret usage in Pods):

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: myapp-pod
  labels:
    app: myapp
    tier: frontend
spec:
  containers:
  - name: web
    image: nginx:1.21
    ports:
    - containerPort: 80
    env:
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: api-credentials    # refers to a Secret containing API_KEY
          key: api-key
    volumeMounts:
    - name: config-volume
      mountPath: /etc/myapp/config   # mount ConfigMap data into container
  volumes:
  - name: config-volume
    configMap:
      name: myapp-config             # refers to a ConfigMap with config data
```

To create this Pod in the cluster, save it as `pod.yaml` and run:

```
kubectl create -f pod.yaml
```

After creation, you can check status with `kubectl get pods` or inspect details with `kubectl describe pod myapp-pod`.

Common Pod Commands: - `kubectl get pods` – List pods in the current namespace (use `-A` for all namespaces). - `kubectl describe pod <name>` – Show detailed information about a pod (state, events). - `kubectl logs <pod-name>` – Fetch logs of the Pod's main container (use `-c <container>` if multiple containers). - `kubectl exec -it <pod-name> -- <command>` – Execute a command in a running container (great for debugging, e.g. open a shell). - `kubectl delete pod <name>` – Delete a Pod (useful for forcing a reschedule if under a Deployment).

Tip: For quick experimentation or exams, `kubectl run nginx --image=nginx --restart=Never` creates a pod object (since `--restart=Never` means no higher controller). This is a fast way to create a Pod *imperatively* without writing YAML. Just remember that in modern Kubernetes, `kubectl run` defaults to a Deployment unless `--restart=Never` is specified for a Pod.

Exam Watchouts: - **Context & Namespace**: Ensure you're working in the correct namespace. In CKAD tasks, the question may specify a namespace. Use `kubectl config set-context --current --`

`namespace=<name>` to switch, or `-n <ns>` in commands. - **Imperative to YAML**: You can generate YAML quickly with imperative commands. For example, `kubectl run temp --image=busybox --dry-run=client -o yaml > pod.yaml` will output the YAML for a Pod using the busybox image (you can then edit it). - **Multi-container Pod logs**: If a Pod has multiple containers, use `kubectl logs pod/name -c <container-name>` to get logs from the specific container. - **Ephemeral Containers**: Kubernetes supports adding an ephemeral container to a running Pod for debugging (via `kubectl debug`). This is advanced, but remember it **won't be running your app** – just a tool container to inspect the Pod.

Deployments (Managing Replica Pods)

While Pods are the basic units, **Deployments** are the recommended way to manage stateless applications in Kubernetes. A Deployment defines a desired state for a set of Pods: how many replicas should be running, what container image they use, etc., and the Deployment controller will ensure the reality matches the desired state (by creating or removing Pods via ReplicaSets). Deployments build upon ReplicaSets (which in turn manage Pods). In the hierarchy: **Deployment > ReplicaSet > Pod/Container**.

Key features of Deployments: - They maintain the specified number of pod replicas (through a ReplicaSet) for high availability. - They allow **rolling updates** to update the Pod template (e.g., rolling out a new version of an image) with zero downtime and the ability to **rollback** if something goes wrong [12]. - They support **pause/resume** of deployments, letting you apply multiple changes and then resume to roll them out together (useful for batch changes) [13] [12]. - They ensure that if a node fails or a Pod is deleted, new Pod replicas are launched elsewhere to maintain the count.

Think of a Deployment as a **manager or supervisor** for pods: you tell it "run 3 instances of my app using this container image," and it takes care of creating those pods (via ReplicaSet) and keeping them running.

Deployment YAML Example: Here is a sample Deployment manifest for an app with 3 replicas. It also shows how the Deployment's Pod template can specify labels, container image, ports, etc.:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp     # Selector must match Pod template's labels
      tier: frontend
  template:
    metadata:
      labels:
```

```
      app: myapp
      tier: frontend
  spec:
    containers:
    - name: myapp-container
      image: myapp:v2           # container image for the app
      ports:
      - containerPort: 80
      env:
      - name: CONFIG_MODE
        value: "production"
```

Important fields: - `.spec.replicas`: how many Pod replicas to run. - `.spec.selector`: how the Deployment finds the Pods it manages (should match the labels on the Pod template). - `.spec.template`: this is essentially a Pod definition (same fields under `spec:` as a Pod) – it's the blueprint for each replica.

When you `kubectl create -f deployment.yaml`, Kubernetes creates the Deployment object. The Deployment immediately creates a new **ReplicaSet** (you can see it with `kubectl get rs`) which then creates the specified number of Pods. The naming convention often is `<deployment-name>-<random suffix>` for the ReplicaSet, and Pods inherit that plus their own ID. For example, if your Deployment is `myapp-deployment`, you might see a ReplicaSet `myapp-deployment-6795844b58` and Pods `myapp-deployment-6795844b58-abc12`, etc. [14] [15].

You normally interact with the Deployment as a whole (scaling it, updating it), rather than dealing with ReplicaSets directly. If you scale the Deployment (via `kubectl scale deploy myapp-deployment --replicas=5`), it will adjust the ReplicaSet's pod count to 5, creating new Pods as needed. If you update the Deployment's Pod template (e.g., change the image), it triggers a **rolling update**: the Deployment creates a new ReplicaSet (for the new version) and gradually increases it while decreasing the old one, ensuring some Pods of the new version come up before old ones are removed (this ensures no downtime). Each update to a Deployment's pod template generates a new **revision** (accessible via `kubectl rollout history deploy/myapp-deployment`).

Rolling Updates & Rollbacks: Deployments support rolling updates by default (the update strategy is RollingUpdate with a configurable batch size). For example, you can update the `image` in the YAML and do `kubectl apply -f deployment.yaml`. The Deployment will rollout the change gradually. Use `kubectl rollout status deployment/myapp-deployment` to watch the progress. If something goes wrong (e.g. the new Pods crash), you can **rollback** to a previous revision with `kubectl rollout undo deployment/myapp-deployment` – this will revert the Deployment's template to the previous version and recreate the old pods [16] [12].

Tip: Know the rollout commands: - `kubectl rollout status deployment/<name>` – watch rollout progress. - `kubectl rollout history deployment/<name>` – see revision history. - `kubectl rollout undo deployment/<name>` – rollback to previous (or specify `--to-revision=n` for a specific revision).

Pause and Resume: If you need to make multiple changes (e.g., update image and increase replicas together), you can pause the Deployment (`kubectl rollout pause deployment/<name>`), apply all changes, then resume (`kubectl rollout resume deployment/<name>`) [13]. This ensures the changes rollout together.

Deployment vs. ReplicaSet vs. Pod: A ReplicaSet is the immediate owner of Pods (ensures N Pods exist), but Deployments bring extra functionality (declarative updates, rollbacks, etc.). In CKAD, you'll mostly work with Deployments for running apps. A ReplicaSet can be created on its own (and was used prior to Deployment in older versions or for special cases), but generally, use Deployment unless you have a specific reason.

Common Deployment Commands: - `kubectl create deployment nginx --image=nginx:1.21` – quick imperative creation (creates a deployment with one replica). Add `--replicas=3` to set replica count. - `kubectl get deployments` – list deployments. - `kubectl describe deployment <name>` – detailed info, including events (like if pods failed to schedule). - `kubectl scale deployment <name> --replicas=10` – scale to 10. - `kubectl set image deployment/<name> <container>=<new-image>` – update the image for a deployment's container (triggers rolling update). E.g., `kubectl set image deployment/myapp-deployment myapp-container=myapp:v3`. - `kubectl rollout undo deployment/<name>` – rollback as discussed.

Exam Tips: - **Quick YAML generation**: `kubectl create deploy <name> --image=<img> -o yaml --dry-run=client > deploy.yaml` gives you a starting YAML which you can then edit (e.g., add environment variables, volume mounts, etc.) before applying. - **Pod selectors**: Ensure the `spec.selector.matchLabels` matches the labels in the pod template; otherwise, the Deployment will not manage its own pods correctly (and creation can fail in recent Kubernetes which validates this) [17] [14]. - **Cleanup**: If you mistakenly create wrong resources, you can delete them and reapply. Pay attention to not leaving stray objects (e.g., an old ReplicaSet lingering after a failed apply – normally Deployment adoption handles it, but just be mindful of what exists with `kubectl get all`). - **Strategy**: By default, maxUnavailable=25%, maxSurge=25%. Usually you won't need to tweak these in the exam unless a question explicitly asks.

! Gotchas: - Changing certain Pod template fields (like labels selectors) on a Deployment that already exists is not allowed – you'd have to create a new Deployment. Typically, you won't change selectors in-place; you change images, commands, etc. - Deployments only manage **Pods** (and their ReplicaSets). For managing other resources (like Jobs or DaemonSets), there are different controllers. - If you edit a Deployment and nothing seems to happen, ensure that your changes were actually in the `spec.template` section. Only changes under `.spec.template` trigger pod recreations (e.g., changing a Deployment label that's not used in the pod template won't rollout anything).

Services and Networking

In Kubernetes, a **Service** is an abstraction that defines a logical set of Pods and a policy by which to access them – essentially it provides a **stable network endpoint** (a single DNS name and IP) to a group of ephemeral Pods [18]. Services enable decoupling between the clients and the pods: the set of Pod IPs may change over time (pods are created/destroyed, scaled, etc.), but clients can always hit the Service's virtual IP or DNS name and get routed to an available Pod [4] [5].

Why Services? Imagine you have a set of Pods (replicas of a backend). They each have their own IP that could change whenever pods restart or reschedule. A Service gives that group of Pods a single fixed IP address (called ClusterIP) and DNS name. The Service will automatically load-balance traffic to the member Pods. Pods find each other or other services via these stable endpoints instead of needing to track IPs.

How Services work: When you create a Service, if it's a ClusterIP type, Kubernetes allocates a **virtual IP** (ClusterIP) from a pool (usually in a range like 10.x.x.x) and maps that to your Pods. **Kube-proxy** on each node implements the load balancing: traffic to the Service's IP is transparently forwarded to one of the Pods (endpoints) backing that Service (using IPTables or IPVS rules). Services select pods by labels (you specify a selector, e.g. `app: myapp`) to define the group of Pods in the service.

There are different **Service types**: - **ClusterIP** (default): Accessible only within the cluster (has a cluster-internal IP). This is great for inter-pod communication (e.g., frontend pod calls backend service via its ClusterIP). Most services start as ClusterIP by default. - **NodePort**: Allocates a port on each Node (e.g., 30000-32767 range by default) and maps it to the Service. This allows accessing the service from outside the cluster by hitting any node's IP on that NodePort [19] [20]. For example, NodePort 30007 on any node will route to the service's pods. NodePorts are often used for simple dev/testing or in combination with external load balancers. - **LoadBalancer**: Available in cloud environments – it provisions an external load balancer (e.g., AWS ELB, Google Cloud LB) that forwards to the Service's pods. When you create a LoadBalancer service, the cloud integration will set up a load balancer and usually open a cloud VIP. Internally, Kubernetes still allocates a ClusterIP and NodePorts for it (the cloud LB typically forwards to NodePorts) [21] [22]. On managed K8s, you often use this to expose apps publicly. - **Headless Service** (`ClusterIP: None`): Doesn't allocate a ClusterIP at all. Instead, it creates DNS records that map to Pod IPs directly. Used when you want service discovery without load-balancing, or with stateful sets to get each pod address.

Service YAML Example: Here's a Service manifest exposing the `myapp` deployment we made, on port 80:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: ClusterIP        # Could also be NodePort, LoadBalancer, etc.
  selector:
    app: myapp           # will target pods with label app=myapp
    tier: frontend
  ports:
  - protocol: TCP
    port: 80             # Service port (clients use this)
    targetPort: 80       # Container port to forward to (can be name or number)
```

This Service will get a cluster IP (say 10.96.0.5) and any Pod in the cluster can hit `10.96.0.5:80` (or the DNS name `myapp-service.default.svc.cluster.local`) to reach one of the Pods (on their port 80) [23]. The `selector` ties the Service to the Pods from our Deployment (they had labels app=myapp,tier=frontend, which we reused here).

If we wanted to expose it externally via NodePort, we'd set `type: NodePort`. Kubernetes would allocate a random port in the NodePort range (e.g., 31555) or we could specify one under `nodePort:`. External clients could then reach any node's IP at that port. **! Gotcha:** With NodePort, make sure your cluster nodes' firewall/security groups allow that port if you're testing externally.

For a cloud LoadBalancer, we'd use `type: LoadBalancer`. This assumes your cluster is on a cloud provider or metal LB is configured. The result is that you get a cloud LB with an external IP, which forwards to the Service (which in turn goes to pods). In many CKA/CKAD test clusters (like KIND or bare-metal), LoadBalancer might not work unless a specific solution is installed. In exam, if they ask for LoadBalancer and it's a simulated cloud, just create it; otherwise, you might simulate with NodePort.

Service Discovery (DNS): Kubernetes clusters typically run a DNS service (like CoreDNS) that automatically creates DNS entries for Services. For example, a Service named "myapp-service" in namespace "default" gets a DNS name `myapp-service.default.svc.cluster.local`. Pods in the same namespace can often resolve just `myapp-service` (because of DNS search domains) [24] [25]. This means your applications can connect to `http://myapp-service:80` without hardcoding IPs. **Tip:** In exam tasks, prefer using the service DNS name to connect between pods/services. It's less error-prone than IPs.

Figure: Service DNS resolution. When a Pod tries to reach `myapp-service` by name, the cluster's DNS (CoreDNS) resolves it to the Service's ClusterIP (e.g., 10.96.0.5). Then kube-proxy routes the traffic to one of the Pod IPs (e.g., 10.1.2.3 or 10.1.2.4) backing that Service. This way, backend Pods can scale or change, and clients don't need to track the changes [18] [26].

Exam Tips: - Given a task "expose an app internally", you'd create a ClusterIP service. If it says "externally", you might use NodePort or LoadBalancer. Check context – if cloud-managed, LoadBalancer is fine. - You can quickly create a Service with `kubectl expose`. For example: `kubectl expose deployment myapp-deployment --port=80 --target-port=80 --name=myapp-service`. By default this makes a ClusterIP service (add `--type=NodePort` if needed). This imperative command is handy in exam to save time. - Services won't route to pods that are not "Ready". Kubernetes automatically integrates readiness probes with services – only pods that are Ready (passed readiness checks) are considered endpoints for the Service [27]. So if your pods are failing readiness, the Service might have zero endpoints. Use `kubectl describe svc myapp-service` and `kubectl get endpoints myapp-service` to see which Pod IPs are linked. - **NodePort range**: By default 30000-32767. Remember you cannot pick a port outside this range unless cluster was configured differently [20] [28]. If you specify a `nodePort` and it's taken or invalid, the service creation will fail. - **targetPort vs port**: The Service `port` is the port clients use to reach the service. `targetPort` is where the traffic goes on the Pod. Often they are the same number (for convenience you can just specify `port:80` and `targetPort` defaults to same if not given). But they can differ if, say, your pod container listens on 8080 but you want the service reachable on 80. - **Headless service**: If you see `clusterIP: None` in YAML, it's headless. Instead of load-balancing, DNS returns all Pod IPs. Used in scenarios like StatefulSets where each pod might need a stable DNS (e.g., `pod-0.service-name.namespace.svc.cluster.local` mapping to that Pod's IP).

Network Policies: In a default Kubernetes cluster, all pods can talk to all other pods (flat network). A **NetworkPolicy** is like a firewall for Pods – it lets you allow or restrict traffic to/from certain Pods. NetworkPolicies work at Layer 3/4 (IP addresses, ports) and are usually implemented by the CNI plugin (like

Calico, Cilium, etc.). They are namespace-scoped and **default deny** any traffic not explicitly allowed when a policy selects a pod [29] [30].

For example, you might have a policy that says "Pods with label `app=db` only accept traffic from Pods with `app=web` on port 5432". Once you apply such a policy, any other source would be blocked from contacting the `db` Pods.

NetworkPolicy YAML skeleton:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-deny-external
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: web
    ports:
    - protocol: TCP
      port: 5432
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          name: monitoring
```

This example says: For pods labeled `app=db`: - Ingress: only allow from pods labeled `role=web` on TCP 5432. (All other ingress to these db pods is denied.) - Egress: only allow connections from these db pods to any pods in namespaces labeled `name=monitoring` (perhaps to allow metrics export). All other egress would be denied from the db pods.

Important points for NetworkPolicy: - If a Pod is selected by **any** NetworkPolicy, then any traffic not explicitly allowed by at least one policy is denied. By default, pods are non-isolated (no restrictions) until a policy selects them [31] [32]. - Policies can select by pod labels (`podSelector`), namespace labels (`namespaceSelector`), and also allow CIDR IP blocks (for external CIDRs) [33] [34]. - `policyTypes` can be Ingress, Egress, or both. You can write separate rules for each direction. - If no NetworkPolicy selects a

pod, that pod remains fully open (all traffic allowed). - Common use case: Create a default deny on a namespace (an NP with empty `from` which denies all ingress to pods in that namespace except whitelisted ones).

Tip: In tasks, if they say "apply a NetworkPolicy to allow traffic only from X to Y", remember to include a rule for DNS or other necessary traffic if needed (sometimes forgetting DNS (UDP 53) or not allowing egress to the internet for updates can break things). In an exam scenario, they usually focus on the core aspect though. Always double-check that your `podSelector` correctly matches the target pods and your `from` selectors match the source pods or namespaces intended.

Expose Services via Ingress (Bonus): An **Ingress** resource is not a Service, but it works with an Ingress Controller to route external HTTP/HTTPS traffic to Services inside the cluster. In CKAD, you might be expected to know basics of creating an Ingress. Example:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myapp-service
            port:
              number: 80
```

This would route requests coming to `myapp.example.com` (at the Ingress controller's address) to the Service `myapp-service` on port 80. Ingress often requires an ingress controller (like NGINX ingress) to be installed. **Exam note:** If they ask for an Ingress, ensure the cluster has an ingress controller available (in some exams clusters, an nginx ingress is pre-installed; if not, you may not get an Ingress question).

Configuration: ConfigMaps and Secrets

Kubernetes applications often need configuration data (like settings files, environment variables) and sensitive data (passwords, keys). Instead of baking these into images, ConfigMaps and Secrets let you decouple configuration from container images.

ConfigMap: A ConfigMap is a key-value dictionary to store non-sensitive configuration data, such as application settings, feature flags, etc. You can create a ConfigMap from literal values, from files, or from env files. Pods can then consume ConfigMaps in two ways: 1. **Environment Variables:** Map ConfigMap

entries to env vars in the container. 2. **Mounted Volume:** Mount the ConfigMap as files (each key becomes a file, where the file's content is the value).

For example, if your app needs a config file, you can store it in a ConfigMap and mount it, so the container sees a file at runtime. Or if it just needs some environment flags, use env.

ConfigMap YAML Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  app.mode: "production"
  welcome.message: "Hello from ConfigMap!"
  database.properties: |
    host=10.1.1.25
    port=3306
```

This ConfigMap has two simple key-values and one multi-line value (a file content for `database.properties`). Notice how multi-line data can be inserted using `|`.

You can create the above with `kubectl apply -f configmap.yaml`. Alternatively, imperatively: `kubectl create configmap myapp-config --from-literal=app.mode=production --from-literal=welcome.message="Hello from ConfigMap!" --from-file=database.properties=./db.properties`.

To use this ConfigMap in a Pod, in the Pod spec: - As env var:

```
env:
  - name: APP_MODE
    valueFrom:
      configMapKeyRef:
        name: myapp-config
        key: app.mode
```

This injects an env var `APP_MODE=production` in the container, reading from the ConfigMap's `app.mode`.

- As volume:

```
volumes:
  - name: config-vol
    configMap:
```

```
      name: myapp-config
volumeMounts:
  - name: config-vol
    mountPath: /etc/myapp/config
```

This will create files under `/etc/myapp/config` inside the container: `app.mode` containing "production", `welcome.message` containing "Hello from ConfigMap!", and `database.properties` containing the multi-line content.

Secret: A Secret is similar to ConfigMap but intended for sensitive data (passwords, API keys, certificates). Under the hood, Secrets data is just base64-encoded strings [35], which is **not true encryption** – it's obfuscation. Kubernetes does allow enabling encryption at rest for Secrets in etcd, but by default, anyone with cluster access could retrieve and decode them [36]. So treat Secrets as sensitive, but not foolproof – control access with RBAC.

Creating a Secret requires base64 values in the YAML (if you write it by hand), or you can use imperative commands to auto-encode. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-creds
type: Opaque
data:
  username: dXNlcm5hbWU=   # base64 for "username"
  password: cGFzc3dvcmQ=   # base64 for "password"
```

In practice, you'd do `echo -n 'username' | base64` to get that string. But easier, use `kubectl create secret generic db-creds --from-literal=username=username --from-literal=password=password` and it will handle base64 encoding in the stored Secret.

You consume Secrets in Pods similarly to ConfigMaps: as env or volumes. For env:

```
env:
- name: DB_USER
  valueFrom:
    secretKeyRef:
      name: db-creds
      key: username
```

Now `DB_USER` will be set to the decoded value of `username` (i.e., "username"). For volume:

```
volumes:
- name: creds-vol
```

```
  secret:
    secretName: db-creds
volumeMounts:
- name: creds-vol
  mountPath: /etc/db-creds
  readOnly: true
```

This will create files `/etc/db-creds/username` and `.../password` with the contents. Many apps can read certs or keys from files, so that's useful.

Secret Types: We used type `Opaque` (basic key-value). There are also specific types like `kubernetes.io/tls` for TLS certs (expects `tls.crt` and `tls.key` keys), or `kubernetes.io/dockerconfigjson` for image pull secrets. For CKAD, mostly Opaque or TLS types.

ConfigMap vs Secret: - Both are similar in usage. The main difference is Secrets are supposed to be treated securely. The API server by default hides secret values in `kubectl get/describe` (it will show `<redacted>` or base64). - **! Gotcha:** *Secrets are not encrypted by default!* They are base64 encoded in etcd [37]. Use Kubernetes encryption config if this matters in real clusters [36]. In exam context, just recall that storing a secret is only slightly more secure (base64) than plain config. But exam likely won't dive into encryption configuration – just know how to create/use them.

Exam Tips: - **Creating quickly:** `kubectl create configmap ...` and `kubectl create secret ...` are your friends. Remember to use `--from-literal` for small values or `--from-file` for files. For secrets, you can also `--from-file` a file containing a password, etc. - If a question says "use a ConfigMap for configuration and a Secret for credentials", you'll need to create those first, then mount or reference them in a Pod/Deployment spec. The workload might already exist and you have to modify it to use the new ConfigMap/Secret. - **Mount vs Env:** If the app expects a config file, mount the ConfigMap as volume. If it just needs a value, env is fine. If not specified, env vars might be simplest for exam speed. - Check for pluralization: `kubectl get configmaps` and `kubectl get secrets`. - When mounting as volumes, by default each key becomes a file. If you only need specific keys as files, you can use `items:` to map them, but in exam usually using the whole thing is fine.

Real-World Analogy: A ConfigMap is like a **settings.ini file** for your application, but stored in the cluster, not baked into the app. A Secret is like a **sealed envelope** containing a password – Kubernetes will hand it to the app when needed, but you wouldn't print it out for everyone.

Gotchas: - If you update a ConfigMap or Secret, by default running Pods do **not** automatically reload those values. Containers would see updated values on restart. One exception: if mounted as a volume, the data in files will update (usually within minutes) automatically, *except* if the app has the file open – and env vars never auto-update. There's an option `subPath` mount that won't update automatically. For exam simplicity, assume config changes require Pod restart (unless you know the trickiness). - Using a Secret as env means it's visible in the Pod's process environment – any process in the container could read it. Mounting as file with strict permissions can be slightly safer. But again, in exam, they won't go deep on that nuance unless asking for best practice.

Volumes (Storage in Pods)

Kubernetes **Volumes** allow containers in a Pod to access shared storage. Unlike container-local storage, which is ephemeral to the container, a Pod volume can outlive one container restart (within the same Pod) and can be shared between multiple containers in the Pod. Volumes are defined at the Pod level.

There are two broad classes of volumes: - **Ephemeral Volumes:** These last only as long as the Pod exists. Examples: `emptyDir`, `configMap` and `secret` volumes (we saw those), `downwardAPI` (exposes Pod metadata to container), etc. When the Pod is deleted, the data is gone. Ephemeral volumes are often used for scratch space, sharing files between sidecar containers, etc. - **Persistent Volumes (PV/PVC):** These are storage resources that live beyond any single Pod's lifetime. They allow data to persist between Pod restarts or be shared among Pods. This includes network disks, cloud disks, NFS mounts, etc. Kubernetes uses the **PersistentVolume** (PV) and **PersistentVolumeClaim** (PVC) abstractions to manage these.

emptyDir: The simplest volume – an empty directory that is created when the Pod starts. All containers in the Pod can read/write it. When the Pod is removed from the node, the emptyDir is deleted (along with its data). If a container crashes and restarts (but Pod is still alive), the emptyDir content remains (since the Pod itself didn't die). Common use: scratch space or sharing data between containers in a Pod (e.g., an init container produces files that the main container uses).

Defining an emptyDir:

```
volumes:
- name: cache-vol
  emptyDir: {}
volumeMounts:
- name: cache-vol
  mountPath: /var/cache/myapp
```

That's it. No configuration inside emptyDir (there's an option for medium=Memory to use tmpfs in RAM, but default is disk).

PersistentVolume & PersistentVolumeClaim (PVC): This is how Kubernetes models durable storage. - A **PersistentVolume** (PV) is like a provisioned storage (e.g., a cloud disk, NFS share, etc.) that exists in the cluster, a resource with a certain capacity and access mode. - A **PersistentVolumeClaim** (PVC) is a request by a user for storage of a certain size and access mode [38] [39]. When you create a PVC, Kubernetes looks for a matching PV to bind, or it may dynamically provision one (if StorageClasses are set up).

In simpler terms: A PV is like a **pre-allocated storage unit** (like a cloud disk), and a PVC is like a **claim ticket** saying "I need 10Gi of storage, of type X". The cluster will either find a PV that fits or create one, then bind the PVC to it. After binding, the PVC can be used in a Pod as a volume.

Using a PVC in a Pod: 1. Create a PVC object (if not already existing):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myapp-pvc
spec:
  accessModes:
    - ReadWriteOnce           # (RWO means one node can mount it read-write;
typical for cloud disks)
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard  # assuming a StorageClass named standard exists
for dynamic provisioning
```

This requests a volume of 1Gi. If dynamic provisioning is enabled (a StorageClass is referenced), the cluster might create a PV (like a GCE Persistent Disk or AWS EBS) automatically. Otherwise, an admin must have created a PV of at least 1Gi that is not yet bound and matches this claim. 2. Once the PVC is **Bound** (you can check `kubectl get pvc myapp-pvc` to see status), you can use it in a Pod spec:

```
volumes:
- name: data-vol
  persistentVolumeClaim:
    claimName: myapp-pvc
volumeMounts:
- name: data-vol
  mountPath: /var/data
```

This will attach the storage backing the PVC into the container at `/var/data`. Now the app can read/write and it will persist even if the Pod is rescheduled (the volume will reattach to the new node where Pod lands, assuming storage supports that).

Lifecycle of PV/PVC: Once bound, the PVC continues to claim that PV until it's released. If the Pod using it is deleted, the PVC (and underlying PV) remain unless you delete the PVC. When a PVC is deleted, the PV's fate depends on its **reclaim policy**: - *Delete*: The PV and the actual storage are deleted (common for dynamically provisioned volumes) [40] [41]. - *Retain*: The PV remains and data remains, but PV is marked Released and won't be reused until an admin manually clears it [40]. - *Recycle*: (Deprecated in newer K8s) – it would scrub the data and make PV Available again.

Typically StorageClasses set the reclaim policy. In clouds it's often Delete (to clean up cloud disks when done).

Access Modes: - RWO (ReadWriteOnce): one node at a time can mount read-write (but could be mounted read-only elsewhere depending on volume plugin). - RWX (ReadWriteMany): multiple nodes can mount read-write (e.g., NFS, some cloud filesystems). - ROX (ReadOnlyMany): multiple can mount but read-only.

Ephemeral Inline Volume types: There are newer types like `emptyDir`, or `projected` (combining ConfigMap+Secret+DownwardAPI), and even CSI ephemeral volumes for a Pod. These are specified directly in Pod spec and don't use PV/PVC.

HostPath: Another volume type is `hostPath` – it mounts a directory from the host node's filesystem into the Pod. This can be useful but is dangerous if not used carefully (ties Pod to node, security concerns). For CKAD, hostPath might not commonly appear unless dealing with something like a local single-node scenario.

Exam Tips: - If a question says "use an **emptyDir** for shared storage between containers in the Pod", you know to add an `emptyDir` volume. Typically straightforward. - If it says "ensure data persists after Pod restarts" or "persist data across pod rescheduling", you need PVC/PV. Possibly they will have a StorageClass available to use. You'd create PVC, then update Pod spec to use it. - Know basic PVC creation and usage. E.g., "Mount a 10Gi volume at /data in the Pod" – you should create a PVC with 10Gi, then add it to volumes and volumeMounts. - **StorageClass**: Possibly a mention. The exam might say "create PVC with storageClass `fast`" or something – just set `storageClassName: fast`. If a default StorageClass exists, you can omit and it'll use default. - Check PVC status. If it's not binding, maybe storageclass name typo or no PV available. In exam environment they likely ensure it binds (maybe via default dynamic provisioning). - Clean up: If you created PVCs, they remain until deleted (which might or might not be needed in the context of one question). - **Accessing volume data**: To verify in exam that volume is working, you could exec into the Pod and check the mounted path for expected files. This might be useful for troubleshooting.

Analogies: A PVC is like a **cloud drive (USB stick)** that you request and plug into your Pod. As long as you have the claim, the drive is reserved for you. You can unplug (pod dies) and later plug into another Pod (on maybe another node), and your data's still there. EmptyDir is like a **temp directory** that exists only while your Pod is running on a node (if the pod goes away, so does it).

Gotchas: - Some volumes (like emptyDir) consume node storage – if the node disk is full, it affects pods. `emptyDir: { sizeLimit: 1Gi }` can impose a size if the runtime supports it. - Permissions: When mounting volumes, the files might have certain permissions. By default, ConfigMap/Secret volumes have `defaultMode: 0444` or so (can be changed). If an application complains about permissions, you might need an init container to `chmod`, or use securityContext to set FSGroup to fix perms. Possibly out of scope for CKAD unless a task specifically calls it out. - If using hostPath in exam (less likely), ensure the path exists or use `mkdir` via init container.

Jobs and CronJobs (Batch Workloads)

Kubernetes **Job** is a controller that supervises pods carrying out **batch processing** tasks, ensuring a certain number of them successfully complete. Unlike Deployments which run continuously, a Job is about **run-to-completion** workloads (think data processing, one-time scripts, backups, etc.). When a Job's task is done successfully, the Job completes and no further pods are run (depending on settings).

A **CronJob** is like a Linux cron – it runs Jobs on a scheduled time (e.g., every hour). CronJobs create Job objects on schedule which then spawn pods.

Job basics: A Job spec includes how many pods to run (`spec.parallelism`) and how many successes are needed (`spec.completions`). By default, completions = 1 and parallelism = 1 (meaning run one pod to completion). If it fails, it can retry based on `backoffLimit`. You can also have parallel Jobs where you want N pods to complete successfully (like process chunks of a dataset). There is also a `spec.restartPolicy` (for the pod template) which usually is OnFailure (so if container fails, Kubernetes will restart the container in the same pod until backoffLimit, rather than create new pod).

Job YAML Example:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  completions: 3         # total successful pods to complete
  parallelism: 3         # how many pods to run in parallel
  backoffLimit: 2        # retry a pod up to 2 times if it fails
  template:
    spec:
      restartPolicy: OnFailure
# each pod will restart on failure, rather than having job start a new pod
      containers:
      - name: worker
        image: busybox
        command: ["sh", "-c", "echo Hello $TASK; sleep 5; echo Job done $TASK"]
        env:
        - name: TASK
          value: "processing data"
```

This job will attempt to run 3 pods in parallel (parallelism=3) and expects each to succeed once (completions=3, parallelism=3 means it will start 3 pods simultaneously, and when all three finish successfully, the job is complete). If any pod fails, it can retry up to 2 times. Each pod prints some message and exits.

Monitoring Jobs: Use `kubectl get jobs` to see status. It shows the number of completions and duration. You can see the pods with `kubectl get pods` (Jobs will name pods like `example-job-<random>`). Each completed pod will remain by default until the Job is garbage-collected (after a TTL or by manual deletion). Completed pods show as status **Completed**. You might delete them or set `.spec.ttlSecondsAfterFinished` in a Job to let Kubernetes clean up finished Jobs' pods after a certain time.

CronJob YAML Example:

```
apiVersion: batch/v1
kind: CronJob
```

```
metadata:
  name: hello-cron
spec:
  schedule: "*/5 * * * *"    # every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: hello
            image: busybox
            args: ["echo", "Hello from CronJob"]
```

The `schedule` is a cron format string. The CronJob controller will, at each 5-minute mark, spawn a Job (with the template spec given). Each Job will create a pod that runs the echo and terminates. CronJobs have some additional fields like `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` (how many old Jobs to keep), and `concurrencyPolicy` (e.g., prevent overlapping jobs if the last one hasn't finished). Default concurrencyPolicy is Allow (jobs can overlap). If a cron schedule is missed (due to controller downtime), by default it might not catch up (or there's a startingDeadlineSeconds field controlling late starts).

Use Cases: - Use Job for one-off tasks: e.g., run a DB migration once, process a batch and exit. - Use CronJob for periodic tasks: e.g., nightly backup, regular report generation, cleanup jobs.

Exam Tips: - CronJob schedule format – remember it's `"min hour dom mon dow"`. They might give you a schedule like "run at 8:00 UTC daily", you'd do `0 8 * * *`. - If they ask for a Job that runs something to completion, ensure restartPolicy in pod template is OnFailure or Never (never means if it fails, the pod won't restart, and Job will create a new pod for a retry – subtle difference). - BackoffLimit: default is 6. This is number of retries before giving up. If a Job's pod fails beyond backoffLimit, the Job is marked **Failed**. - CronJobs in newer K8s (v1.21+) are stable. They use batch/v1 now (older was batch/v1beta1 – just noting in case). - Check the timezone – Kubernetes CronJobs use controller's timezone (often UTC) unless documented otherwise. Usually assume UTC for exam unless told differently. - For debugging: `kubectl logs <pod>` to see what your job's pod did. If a Job is stuck or failing, describe the Job or pods to see events.

Analogies: A Job is like issuing a **batch processing command** – say you need to run a script on a dataset, you submit a Job for it, and you'll either get results or an error, but it's not meant to run forever. A CronJob is like scheduling a **cron task** on a server, but instead of one server, the cluster ensures it runs (even if leader node changes).

Gotchas: - CronJobs can flood your cluster if mis-scheduled (e.g., `* * * * *` with a long-running job can create overlapping jobs). In exam they won't do something that tricky, but best practice: set concurrencyPolicy=Forbid or so for such cases. - A Job with parallel pods (parallelism > 1) and more completions than parallelism – the pods will run in batches. E.g., completions=10, parallelism=2 will run 2 at a time until 10 successful completions total. - If a Job's pods all fail until backoffLimit, the Job stops

spawning new pods and is marked failed. You'd need to delete or retry it. - Deleting a Job will by default delete its pods too (owner references). If you want to stop a running Job you can delete it (or scale its parallelism to 0 by editing, but deletion is quicker).

Observability and Pod Health (Probes, Logging, Monitoring)

A big part of operating applications is making sure they are healthy and debugged when issues occur. Kubernetes provides *Probes* to monitor container health, *Events* for lifecycle info, and integrates with logging and monitoring systems. In CKAD, typical tasks include setting up **liveness/readiness probes** and using `kubectl` to inspect logs or troubleshoot pods.

Liveness and Readiness Probes: These are health checks defined in the Pod spec that kubelet performs on containers: - A **Liveness probe** checks if a container is still "alive" (working). If the liveness probe fails, Kubernetes will **restart the container** [42]. This helps recover from deadlocks or bugs where the process is running but stuck. - A **Readiness probe** checks if the application is ready to serve requests. If a readiness probe fails, Kubernetes will mark the Pod as **not ready** and temporarily remove it from Service endpoints [27]. The container is not restarted, but traffic won't be sent to it until it passes readiness again. Use readiness to signal "I'm not ready to receive traffic yet" (e.g., during startup, or if dependencies are unavailable). - A **Startup probe** (added in later versions) is for slow-starting containers. It's like a initial liveness check with possibly a long timeout – if a startup probe is defined, regular liveness and readiness checks are held off until it succeeds [43].

Probes have **handlers**: most common are `httpGet` (make an HTTP GET to a given port/path), `tcpSocket` (attempt to connect on a port), or `exec` (run a command inside container) [44] [45]. Each probe can be configured with `initialDelaySeconds` (wait time after container start before starting probing), `periodSeconds` (how often), `failureThreshold`, `successThreshold`, and `timeoutSeconds`. If a probe fails `failureThreshold` times in a row, it's considered a failure. For readiness, one failure means pod is marked unready (but one success can mark ready again if successThreshold met).

Example Probe configuration:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
  failureThreshold: 3

readinessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

Here, liveness will GET `/healthz` on port 8080 every 20s after initial 15s delay. If that endpoint doesn't return a 200-399 for 3 consecutive attempts (failureThreshold 3) [46], container is restarted. Readiness will check TCP connection on port 8080 starting 5s after launch, every 10s. If it fails even once, the pod is marked unready (and presumably removed from service endpoints until it succeeds).

Choosing probe types: - Use HTTP probe if your app exposes a health URI. Many web frameworks have health endpoints. - Use TCP probe if simple – it just checks port open (not deep check). - Use exec if you need to run a custom command (like checking a file existence or internal status). But exec can consume more CPU if heavy – use with care.

! Gotcha: Misconfigured liveness probes can cause **restart loops** – if your thresholds or delays are wrong, you might kill containers that were actually just busy or slow. Only use a liveness probe for cases where a restart is the desired cure (like stuck processes) [47] [48]. Readiness is usually more commonly needed (to avoid sending traffic to unready pods on startup or during transient faults).

Exam considerations: - If asked to add probes, ensure the container actually has something at that endpoint or command. In a simulation environment, maybe they use a common image that has an endpoint. If uncertain, you could do a simple exec probe like `command: ["grep", "running", "/tmp/status"]` just as an example (assuming something writes to /tmp/status). But likely the question will guide, e.g., "use /healthz for liveness and /login for readiness on port 8080". - Calculating timings: If initialDelay=0, failureThreshold=3, period=10, then roughly 3 failures = ~30 seconds before restart. If startupProbe is used, liveness won't start until startup passes, which is helpful for slow boot apps.

Logging: Kubernetes doesn't have a built-in log storage, but you can fetch logs from running pods using `kubectl logs`. Each container's stdout/stderr is captured by container runtime and accessible via this command. Some tips: - `kubectl logs pod/name` – gets logs from the main (first) container. - `kubectl logs pod/name -c sidecar-name` – get logs from a specific container if multiple. - `kubectl logs pod/name --previous` – get logs from the last instance of the container if it crashed (useful after a crash to see why). - For long running or very verbose logs, you can add `--tail` or pipe it.

If an app writes to a file instead of stdout, `kubectl logs` won't see that – one would need a sidecar or use `kubectl exec` to check the file. In real clusters, a logging agent like Fluentd ships logs to ELK or similar, but not needed for CKAD tasks.

Monitoring & Metrics: Kubernetes itself doesn't provide application-level monitoring, but the exam might involve `kubectl top` which shows resource usage if metrics-server is installed. Also, events are important: `kubectl get events` or `kubectl describe pod` to see warnings (like CrashLoopBackOff, OOMKilled, etc.).

Debugging with kubectl: - If a Pod is crashing (CrashLoopBackOff), check `kubectl logs` (with `--previous` for the last crash). - `kubectl describe pod` to see events – often it shows if liveness probe failed (`Liveness probe failed: ...` events) or image pull errors, etc. - `kubectl exec -it pod -- sh` to get a shell inside (if the container image has a shell; busybox or ubuntu based ones do). - If the container image doesn't have shell and you need debugging, you can use `kubectl debug` (which in newer kubectl inserts an ephemeral container). E.g., `kubectl debug -it`

`mypod --image=busybox --target=myapp-container` will attach a new debug container to the Pod namespace, where you can investigate.

Example scenario: There is a Pod running but not responding. You added a readiness probe and see it's failing. You `kubectl exec` in and find some config is wrong. You could fix the ConfigMap and update it, then maybe restart the pod to pick it up.

Events & Status: When troubleshooting, note the Pod status fields: - `Waiting` with reason (e.g. ImagePullBackOff, CrashLoopBackOff, ErrImagePull). - CrashLoopBackOff means container failed, got restarted repeatedly. Use logs to find cause. - OOMKilled (OutOfMemory) appears in events if container hit its memory limit and kernel killed it. The event will say "Killed container" with reason OOMKilled. Then you know to increase memory limits or find leak.

Resource Limits/Requests: Not explicitly asked in the outline, but good to mention: You can specify resource requests/limits on containers (cpu, memory). For exam, if asked, syntax:

```
resources:
  requests:
    cpu: "0.5"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "512Mi"
```

Requests help scheduler decide which node (ensuring node has that capacity free), and limits enforce at runtime (if container tries to exceed, it gets throttled for CPU or killed for memory). Possibly an exam question could be "set memory limit to 512Mi and request 256Mi for the web container."

Tip: Use short units properly (Mi for mebibytes, not just M). CPU can be expressed in millicores (500m = 0.5 core).

Security and Service Accounts (Brief)

While not a heavy focus in CKAD compared to CKA, application developers should know: - **Service Accounts:** Pods by default use the "default" service account in their namespace, which provides a token for in-cluster API access. If your app needs to call the Kubernetes API, you might create a dedicated ServiceAccount and attach it to the Pod (via `.spec.serviceAccountName`). In exam context, they might ask to run a Pod with a specific ServiceAccount or to ensure it's using default token. - **SecurityContext:** Allows setting user IDs, capabilities, etc. For instance, you can run a container as a non-root user by `securityContext.runAsUser: 1000`. Or set `readOnlyRootFilesystem: true` in container securityContext. Possibly not a main exam point unless the question explicitly requires it (like "ensure the container runs as non-root"). - **ImagePullSecrets:** If you need to pull a private image, you'd create a Secret of type docker-registry and reference it in the Pod spec under `imagePullSecrets`. In CKAD this might be a corner scenario, likely not unless they mention it. - **Taints and Tolerations:** More admin, but if a node is tainted (like master nodes often have `NoSchedule` taint), your pod needs a toleration to schedule there.

Probably not needed in CKAD except maybe to schedule something on master (rare in exam tasks). - **Namespaces:** To organize pods/services. Some tasks might specify a namespace. Use `kubectl create ns xyz` to create if needed, and include `-n xyz` in your commands or set context accordingly.

Summary Tables and Cheat Sheet

Common `kubectl` Commands:

Task	Command Example
List resources	`kubectl get pods`, `kubectl get svc`, `kubectl get deploy` (add `-n` for namespace)
View detailed info	`kubectl describe pod <name>`
Stream logs	`kubectl logs -f <pod>` (add `-c` for container)
Execute in container	`kubectl exec -it <pod> -- /bin/sh` (or bash)
Run a one-off pod (no controller)	`kubectl run tmp --image=busybox --restart=Never -- <cmd>`
Create resource from YAML	`kubectl apply -f file.yaml` (or `create` for one-time create)
Edit resource (live)	`kubectl edit deployment/<name>` (opens editor, sends patch)
Scale a deployment	`kubectl scale deploy/<name> --replicas=NUM`
Expose a deployment as Service	`kubectl expose deploy/<name> --port=80 --target-port=8000 --type=ClusterIP`
Set image on a deployment	`kubectl set image deploy/<name> <container>=<image:tag>`
View rollout status/ history	`kubectl rollout status deploy/<name>` / `kubectl rollout history deploy/<name>`
Pause/Resume rollout	`kubectl rollout pause deploy/<name>` / `... resume ...`
Delete resource	`kubectl delete pod/<name>` (also works for deploy, svc etc.)

Probe Types Summary:

Probe Type	What it does	Typical Use
Liveness	Checks if container is alive (should be restarted if not) [42].	Restart deadlocked or stuck apps. (E.g., an HTTP 500 on /health could trigger restart)

Probe Type	What it does	Typical Use
Readiness	Checks if container is ready to serve traffic [27].	Control Service membership. Pod won't get traffic until ready. (E.g., app not ready until cache warmed)
Startup	Checks if container *has started* successfully [43].	For slow startups, delay liveness/readiness until this succeeds (prevent premature failure)

Volume Types:

Volume Type	Lasts beyond Pod?	Use Case
emptyDir	No (deleted with Pod)	Scratch space, cache, inter-container file sharing.
hostPath	No (tied to Node lifecycle)	Access host node files (only if needed, e.g., log directory or socket).
configMap/secret	No (managed by k8s, changes reflected live)	Inject configuration or secrets as files.
PersistentVolumeClaim (bound to PV)	Yes (PV lifecycle is independent) [38]	Persistent data, databases, storage that survives Pod restarts or re-scheduling.
downwardAPI	No	Expose Pod metadata (labels, etc.) to pod via files or env.
projected	No	Combine multiple sources (config, secret, downwardAPI) into one volume.

Service Types:

Type	Accessibility	Notes
ClusterIP (default)	In-cluster only (virtual IP) [18]	Most internal services use this. Has DNS entry inside cluster.
NodePort	On Node's IP & fixed port (each Node) [19] [20]	Usually 30000-32767. Allows external access if Node IP reachable.
LoadBalancer	External LB (cloud provider) fronting a Service [21]	Easiest way on cloud to expose to internet. Not used on bare-metal without LB solution.
ExternalName	No cluster IP; just DNS CNAME to external name	Used to give Services as aliases to external (e.g., legacy API external.svc).

Deployment vs Job vs DaemonSet vs StatefulSet: (for awareness)

- **Deployment:** Manages stateless apps (replicated Pods), supports rolling updates and rollback.
- **Job:** Manages pods that run to completion (finite work).

- **CronJob:** Schedules Jobs periodically (e.g., nightly).
- **DaemonSet:** Ensures one (or some number) of Pod per node (e.g., log collectors, monitoring agents).
- **StatefulSet:** Manages stateful apps that need stable network ID and stable storage. Provides ordered, unique pod identities (useful for databases). Uses Headless Service for stable DNS of pods.

For CKAD, Deployments, Jobs, CronJobs are most relevant; StatefulSets and DaemonSets are usually more CKA or specialized scenarios, but good to recognize if mentioned.

This concludes the CKAD study guide. Focus on practicing the creation and modification of these resources quickly and correctly. Use the tips and examples above to confidently tackle exam scenarios. **Good luck on your CKAD!** Remember to stay calm, use the documentation when needed, and apply the concepts you've learned – Kubernetes will take care of the rest .

---

[1] [2] [3] Pods | Kubernetes
https://kubernetes.io/docs/concepts/workloads/pods/

[4] [5] [18] [19] [20] [21] [22] [26] [28] Service | Kubernetes
https://kubernetes.io/docs/concepts/services-networking/service/

[6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] 6.1 KodeKloud Kubernetes CKAD PDF | PDF | Computer Cluster | Test (Assessment)
https://www.scribd.com/document/449221920/6-1-KodeKloud-Kubernetes-CKAD-pdf-PDF

[23] [24] [25] DNS for Services and Pods | Kubernetes
https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

[27] [42] [43] [47] [48] Configure Liveness, Readiness and Startup Probes | Kubernetes
https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

[29] [30] [31] [32] [33] [34] Network Policies | Kubernetes
https://kubernetes.io/docs/concepts/services-networking/network-policies/

[35] [36] Secure Kubernetes Secrets: A Complete Guide
https://www.plural.sh/blog/kubernetes-secret-guide/

[37] Why does k8s secrets need to be base64 encoded when ...
https://stackoverflow.com/questions/49046439/why-does-k8s-secrets-need-to-be-base64-encoded-when-configmaps-does-not

[38] [39] Persistent Volumes | Kubernetes
https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[40] [41] 7 Stages in the Life of a Kubernetes Persistent Volume (PV) | Spot.io
https://spot.io/resources/kubernetes-architecture/7-stages-in-the-life-of-a-kubernetes-persistent-volume-pv/

[44] [45] [46] Kubernetes Liveness Probes: Configuration & Best Practices
https://www.groundcover.com/blog/kubernetes-liveness-probe