# 100% Pass Linux Foundation - KCNA - Kubernetes and Cloud Native Associate–High Pass-Rate New Dumps Ppt


BONUS!!! Download part of It-Tests KCNA dumps for free: https://drive.google.com/open?id=1kWFpxYTR-s4L5fMPGoXa1irxnbARAHQv

The KCNA test materials are mainly through three learning modes, Pdf, Online and software respectively.The KCNA test materials have a biggest advantage that is different from some online learning platform which has using terminal number limitation, the KCNA quiz torrent can meet the client to log in to learn more, at the same time, the user can be conducted on multiple computers online learning, greatly reducing the time, and people can use the machine online of KCNA Test Prep more conveniently at the same time.

In fact, our KCNA study materials are not expensive at all. The prices of the KCNA exam questions are reasonable and affordable while the quality of them are unmatched high. So with minimum costs you can harvest desirable outcomes more than you can imagine. By using our KCNA Training Materials you can gain immensely without incurring a large amount of expenditure. And we give some discounts on special festivals.

**>> New KCNA Dumps Ppt <<**

## KCNA Dumps Guide - Accurate KCNA Test

As a professional IT exam dumps provider, our website gives you more than just KCNA exam answers and questions, we also offer you the comprehensive service when you buy and after sales. Our valid KCNA dumps torrent and training materials are the guarantee of passing exam and the way to get succeed in IT field. We will send the latest KCNA vce pdf immediately once we have any updating about this dump.

Linux Foundation is a non-profit organization that is dedicated to promoting open-source technology, collaboration, and innovation. One of the ways in which it does this is through its certification program, which is designed to recognize and validate the skills and expertise of individuals working in the field of open-source technology. As part of this program, the Linux Foundation offers the KCNA (Kubernetes and Cloud Native Associate) exam.

## Linux Foundation Kubernetes and Cloud Native Associate Sample Questions (Q94-Q99):

**NEW QUESTION # 94**
What is the reference implementation of the OCI runtime specification?

- A. lxc
- B. CRI-O
- C. Docker
- D. runc

**Answer: D**

Explanation:
The verified correct answer is C (runc). The Open Container Initiative (OCI) defines standards for container image format and

runtime behavior. The OCI runtime specification describes how to run a container (process execution, namespaces, cgroups, filesystem mounts, capabilities, etc.). runc is widely recognized as the reference implementation of that runtime spec and is used underneath many higher-level container runtimes.
In common container stacks, Kubernetes nodes typically run a CRI-compliant runtime such as containerd or CRI-O. Those runtimes handle image management, container lifecycle coordination, and CRI integration, but they usually invoke an OCI runtime to actually create and start containers. In many deployments, that OCI runtime is runc (or a compatible alternative). This layering helps keep responsibilities separated: CRI runtime manages orchestration-facing operations; OCI runtime performs the low-level container creation according to the standardized spec.
Option A (lxc) is an older Linux containers technology and tooling ecosystem, but it is not the OCI runtime reference implementation. Option B (CRI-O) is a Kubernetes-focused container runtime that implements CRI; it uses OCI runtimes (often runc) underneath, so it's not the reference implementation itself. Option D (Docker) is a broader platform/tooling suite; while Docker historically used runc under the hood and helped popularize containers, the OCI reference runtime implementation is runc, not Docker.
Understanding this matters in container orchestration contexts because it clarifies what Kubernetes depends on: Kubernetes relies on CRI for runtime integration, and runtimes rely on OCI standards for interoperability. OCI standards ensure that images and runtime behavior are portable across tools and vendors, and runc is the canonical implementation that demonstrates those standards in practice.
Therefore, the correct answer is C: runc.

**NEW QUESTION # 95**
How many different Kubernetes service types can you define?

- A. 0
- B. 1
- C. 2
- D. 3

**Answer: A**

Explanation:
Kubernetes defines four primary Service types, which is why C (4) is correct. The commonly recognized Service spec.type values are:
* ClusterIP: The default type. Exposes the Service on an internal virtual IP reachable only within the cluster. This supports typical east-west traffic between workloads.
* NodePort: Exposes the Service on a static port on each node. Traffic to <NodeIP>:<NodePort> is forwarded to the Service endpoints. This is often used for simple external access in environments without load balancers, or as a building block for other systems.
* LoadBalancer: Integrates with a cloud provider (or load balancer implementation) to provision an external load balancer and route traffic to the Service. This is common in managed Kubernetes.
* ExternalName: Maps the Service name to an external DNS name via a CNAME record, allowing in- cluster clients to use a consistent Service DNS name to reach an external dependency.
Some people also talk about "Headless Services," but headless is not a separate type; it's a behavior achieved by setting clusterIP: None. Headless Services still use the Service API object but change DNS and virtual-IP behavior to return endpoint IPs directly rather than a ClusterIP. That's why the canonical count of "Service types" is four.
This question tests understanding of the Service abstraction: Service type controls how a stable service identity is exposed (internal VIP, node port, external LB, or DNS alias), while selectors/endpoints control where traffic goes (the backend Pods). Different environments will favor different types: ClusterIP for internal microservices, LoadBalancer for external exposure in cloud, NodePort for bare-metal or simple access, ExternalName for bridging to outside services.
Therefore, the verified answer is C (4).
==========

**NEW QUESTION # 96**
Which of the following resources helps in managing a stateless application workload on a Kubernetes cluster?

- A. kubectl
- B. Deployment
- C. StatefulSet
- D. DaemonSet

**Answer: B**

Explanation:
The correct answer is D: Deployment. A Deployment is the standard Kubernetes controller for managing stateless applications. It provides declarative updates, replica management, and rollout/rollback functionality.
You define the desired state (container image, environment variables, ports, replica count) in the Deployment spec, and Kubernetes ensures the specified number of Pods are running and updated according to strategy (RollingUpdate by default).
Stateless workloads are ideal for Deployments because each replica is interchangeable. If a Pod dies, a new one can be created anywhere; if traffic increases, replicas can be increased; if you need to update the app, a new ReplicaSet is created and traffic shifts gradually to new Pods. Deployments integrate naturally with Services for stable networking and load balancing.
Why the other options are incorrect:
* A DaemonSet ensures one Pod per node (or selected nodes). It's for node-level agents, not generic stateless service replicas.
* A StatefulSet is for workloads needing stable identity, ordered rollout, and persistent storage per replica (databases, quorum systems). That's not the typical stateless app case.
* kubectl is a CLI tool; it doesn't "manage" workloads as a controller resource.
In real cluster operations, almost every stateless microservice is represented as a Deployment plus a Service (and often an Ingress/Gateway for edge routing). Deployments also support advanced delivery patterns (maxSurge/maxUnavailable tuning) and easy integration with HPA for horizontal scaling. Because the question is specifically "managing a stateless application workload," the Kubernetes resource designed for that is clearly the Deployment.
==========

**NEW QUESTION # 97**
Which mechanism allows extending the Kubernetes API?

- A. ConfigMap
- B. CustomResourceDefinition
- C. MutatingAdmissionWebhook mechanism
- D. Kustomize

**Answer: B**

Explanation:
The correct answer is B: CustomResourceDefinition (CRD). Kubernetes is designed to be extensible. A CRD lets you define your own resource types (custom API objects) that behave like native Kubernetes resources: they can be created with YAML, stored in etcd, retrieved via the API server, and managed using kubectl. For example, operators commonly define CRDs such as Databases, RedisClusters, or Certificates to model higher-level application concepts.
A CRD extends the API by adding a new kind under a group/version (e.g., example.com/v1). You typically pair CRDs with a controller (often called an operator) that watches these custom objects and reconciles real-world resources (Deployments, StatefulSets, cloud resources) to match the desired state specified in the CRD instances. This is the same control-loop pattern used for built-in controllers-just applied to your custom domain.
Why the other options aren't correct: ConfigMaps store configuration data but do not add new API types. A MutatingAdmissionWebhook can modify or validate requests for existing resources, but it doesn't define new API kinds; it enforces policy or injects defaults. Kustomize is a manifest customization tool (patch/overlay) and doesn't extend the Kubernetes API surface.
CRDs are foundational to much of the Kubernetes ecosystem: cert-manager, Argo, Istio, and many operators rely heavily on CRDs. They also support schema validation via OpenAPI v3 schemas, which improves safety and tooling (better error messages, IDE hints). Therefore, the mechanism for extending the Kubernetes API is CustomResourceDefinition, option B.

**NEW QUESTION # 98**
What is the main purpose of the Open Container Initiative (OCI)?

- A. Creating industry standards around container formats and runtimes for private purposes.
- B. Improving the security of standards around container formats and runtimes.
- C. Accelerating the adoption of containers and Kubernetes in the industry.
- D. Creating open industry standards around container formats and runtimes.

**Answer: D**

Explanation:
B is correct: the OCI's main purpose is to create open, vendor-neutral industry standards for container image formats and container

runtimes. Standardization is critical in container orchestration because portability is a core promise: you should be able to build an image once and run it across different environments and runtimes without rewriting packaging or execution logic.
OCI defines (at a high level) two foundational specs:
* Image specification: how container images are packaged (layers, metadata, manifests).
* Runtime specification: how to run a container (filesystem setup, namespaces/cgroups behavior, lifecycle).These standards enable interoperability across tooling. For example, higher-level runtimes (like containerd or CRI-O) rely on OCI-compliant components (often runc or equivalents) to execute containers consistently.
Why the other options are not the best answer:
* A (accelerating adoption) might be an indirect outcome, but it's not the OCI's core charter.
* C is contradictory ("industry standards" but "for private purposes")-OCI is explicitly about open standards.
* D (improving security) can be helped by standardization and best practices, but OCI is not primarily a security standards body; its central function is format and runtime interoperability.
In Kubernetes specifically, OCI is part of the "plumbing" that makes runtimes replaceable. Kubernetes talks to runtimes via CRI; runtimes execute containers via OCI. This layering helps Kubernetes remain runtime- agnostic while still benefiting from consistent container behavior everywhere.
Therefore, the correct choice is B: OCI creates open standards around container formats and runtimes.
==========

**NEW QUESTION # 99**
.....

s4L5fMPGoXa1irxnbARAHQv