100% Pass 2026 Linux Foundation KCNA: Useful New Kubernetes and Cloud Native Associate Exam Testking

P.S. Free & New KCNA dumps are available on Google Drive shared by BraindumpsPass: https://drive.google.com/open?id=12vlraRJOleUIMnaoFsa6-cx3eyV_11uU

You can install Linux Foundation KCNA PRACTICE TEST file and desktop practice test software on your devices and easily start Kubernetes and Cloud Native Associate (KCNA) exam preparation right now. Whereas the "BraindumpsPass" KCNA web-based practice test software is concerned, it is a simple browser-based application that works with all the latest web browsers. Just put the link of BraindumpsPass KCNA web-based practice test application in your browser and start Linux Foundation KCNA exam preparation without wasting further time. The "BraindumpsPass" is quite confident that you will be the next successful Kubernetes and Cloud Native Associate exam candidate.

Linux Foundation KCNA Exam is an excellent opportunity for IT professionals to validate their skills and knowledge in cloud-native technologies. Whether you are a developer, system administrator, or IT manager, the certification can help you advance your career and stay competitive in the rapidly evolving world of cloud computing.

>> New KCNA Exam Testking <<

Linux Foundation KCNA PDF Dumps Format

The only goal of all experts and professors in our company is to design the best and suitable study materials for all people. According to the different demands of many customers, they have designed the three different versions of the KCNA Study Materials for all customers. They sincerely hope that all people who use the KCNA study materials from our company can pass the exam and get the related certification successfully.

Linux Foundation Kubernetes and Cloud Native Associate (KCNA) Certification Exam is a performance-based exam that assesses an individual's knowledge and skills in the field of Kubernetes and cloud native technologies. Kubernetes and Cloud Native Associate certification is designed for those who are new to these technologies or those who have some experience but want to validate their expertise. KCNA Exam covers a wide range of topics, including Kubernetes architecture, deployment, and management, as well as cloud native technologies such as containerization, microservices, and serverless computing.

Linux Foundation Kubernetes and Cloud Native Associate Sample Questions (Q23-Q28):

NEW QUESTION # 23
How does dynamic storage provisioning work?

- A. A Pod requests dynamically provisioned storage by including a StorageClass and the Pod name in their PersistentVolumeClaim.
- B. A user requests dynamically provisioned storage by including an existing StorageClass in their PersistentVolumeClaim.
- C. An administrator creates a StorageClass and includes it in their Pod YAML definition file without creating a PersistentVolumeClaim.
- D. An administrator creates a PersistentVolume and includes the name of the PersistentVolume in their Pod YAML definition file.

Answer: B

Explanation:
Dynamic provisioning is the Kubernetes mechanism where storage is created on-demand when a user creates a PersistentVolumeClaim (PVC) that references a StorageClass, so A is correct. In this model, the user does not need to pre-create a PersistentVolume (PV). Instead, the StorageClass points to a provisioner (typically a CSI driver) that knows how to create a volume in the underlying storage system (cloud disk, SAN, NAS, etc.). When the PVC is created with storageClassName: <class>, Kubernetes triggers the provisioner to create a new volume and then binds the resulting PV to that PVC.
This is why option B is incorrect: you do not put a StorageClass "in the Pod YAML" to request provisioning. Pods reference PVCs, not StorageClasses directly. Option C is incorrect because the PVC does not need the Pod name; binding is done via the PVC itself. Option D describes static provisioning: an admin pre-creates PVs and users claim them by creating PVCs that match the PV (capacity, access modes, selectors). Static provisioning can work, but it is not dynamic provisioning.
Under the hood, the StorageClass can define parameters like volume type, replication, encryption, and binding behavior (e.g., volumeBindingMode: WaitForFirstConsumer to delay provisioning until the Pod is scheduled, ensuring the volume is created in the correct zone). Reclaim policies (Delete/Retain) define what happens to the underlying volume after the PVC is deleted.
In cloud-native operations, dynamic provisioning is preferred because it improves developer self-service, reduces manual admin work, and makes scaling stateful workloads easier and faster. The essence is: PVC + StorageClass → automatic PV creation and binding.

NEW QUESTION # 24
In a Kubernetes cluster, which scenario best illustrates the use case for a StatefulSet?

- A. A service that routes traffic to various microservices in the cluster.
- B. A web application that requires multiple replicas for load balancing.
- C. A database that requires persistent storage and stable network identities.
- D. A background job that runs periodically and does not maintain state.

Answer: C

Explanation:
A StatefulSet is a Kubernetes workload API object specifically designed to manage stateful applications.
Unlike Deployments or ReplicaSets, which are intended for stateless workloads, StatefulSets provide guarantees about the ordering, uniqueness, and persistence of Pods. These guarantees are critical for applications that rely on stable identities and durable storage, such as databases, message brokers, and distributed systems.
The defining characteristics of a StatefulSet include stable network identities, persistent storage, and ordered deployment and scaling. Each Pod created by a StatefulSet receives a unique and predictable name (for example, database-0, database-1), which remains consistent across Pod restarts. This stable identity is essential for stateful applications that depend on fixed hostnames for leader election, replication, or peer discovery. Additionally, StatefulSets are commonly used with PersistentVolumeClaims, ensuring that each Pod is bound to its own persistent storage that is retained even if the Pod is rescheduled or restarted.
Option A is incorrect because web applications that scale horizontally for load balancing are typically stateless and are best managed by Deployments, which allow Pods to be created and destroyed freely without preserving identity. Option B is incorrect because traffic routing to microservices is handled by Services or Ingress resources, not StatefulSets. Option C is incorrect because periodic background jobs that do not maintain state are better suited for Jobs or CronJobs.
Option D correctly represents the ideal use case for a StatefulSet. Databases require persistent data storage, stable network identities, and predictable startup and shutdown behavior. StatefulSets ensure that Pods are started, stopped, and updated in a controlled order, which helps maintain data consistency and application reliability. According to Kubernetes documentation, whenever an application requires stable identities, ordered deployment, and persistent state, a StatefulSet is the recommended and verified solution, making option D the correct answer.

NEW QUESTION # 25
Which Kubernetes API resource is responsible for defining the structure of your Kubernetes cluster, including the number and types

of nodes?

- A. Node
- B. pod
- C. Cluster
- D. Deployment
- E. Service

Answer: C

Explanation:
The Cluster resource is responsible for defining the structure of the Kubernetes cluster, including the number and types of nodes. It is a top-level resource in Kubernetes. Deployments, Services, Pods, and Nodes are all lower-level resources that are defined within a cluster.

NEW QUESTION # 26
Imagine there is a requirement to run a database backup every day. Which Kubernetes resource could be used to achieve that?

- A. Task
- B. Job
- C. kube-scheduler
- D. CronJob

Answer: D

Explanation:
To run a workload on a repeating schedule (like "every day"), Kubernetes provides CronJob, making B correct. A CronJob creates Jobs according to a cron-formatted schedule, and then each Job creates one or more Pods that run to completion. This is the Kubernetes-native replacement for traditional cron scheduling, but implemented as a declarative resource managed by controllers in the cluster.
For a daily database backup, you'd define a CronJob with a schedule (e.g., "0 2 * * *" for 2:00 AM daily), and specify the Pod template that performs the backup (invokes backup scripts/tools, writes output to durable storage, uploads to object storage, etc.). Kubernetes will then create a Job at each scheduled time. CronJobs also support operational controls like concurrencyPolicy (Allow/Forbid/Replace) to decide what happens if a previous backup is still running, startingDeadlineSeconds to handle missed schedules, and history limits to retain recent successful/failed Job records for debugging.
Option D (Job) is close but not sufficient for "every day." A Job runs a workload until completion once; you would need an external scheduler to create a Job every day. Option A (kube-scheduler) is a control plane component responsible for placing Pods onto nodes and does not schedule recurring tasks. Option C ("Task") is not a standard Kubernetes workload resource.
This question is fundamentally about mapping a recurring operational requirement (backup cadence) to Kubernetes primitives. The correct design is: CronJob triggers Job creation on a schedule; Job runs Pods to completion. Therefore, the correct answer is B.
==========

NEW QUESTION # 27
What is the main purpose of a DaemonSet?

- A. A DaemonSet ensures that a process (agent) runs on every node.
- B. A DaemonSet ensures that there are as many pods running as specified in the replicas field.
- C. A DaemonSet ensures that the kubelet is constantly up and running.
- D. A DaemonSet ensures that all (or certain) nodes run a copy of a Pod.

Answer: D

Explanation:
The correct answer is A. A DaemonSet is a workload controller whose job is to ensure that a specific Pod runs on all nodes (or on a selected subset of nodes) in the cluster. This is fundamentally different from Deployments/ReplicaSets, which aim to maintain a certain replica count regardless of node count. With a DaemonSet, the number of Pods is implicitly tied to the number of eligible nodes: add a node, and the DaemonSet automatically schedules a Pod there; remove a node, and its Pod goes away.
DaemonSets are commonly used for node-level services and background agents: log collectors, node monitoring agents, storage daemons, CNI components, or security agents-anything where you want a presence on each node to interact with node resources.

This aligns with option D's phrasing ("agent on every node"), but option A is the canonical definition and is slightly broader because it covers "all or certain nodes" (via node selectors/affinity/taints-tolerations) and the fact that the unit is a Pod.
Why the other options are wrong: DaemonSets do not "keep kubelet running" (B); kubelet is a node service managed by the OS. DaemonSets do not use a replicas field to maintain a specific count (C); that's Deployment/ReplicaSet behavior.
Operationally, DaemonSets matter for cluster operations because they provide consistent node coverage and automatically react to node pool scaling. They also require careful scheduling constraints so they land only where intended (e.g., only Linux nodes, only GPU nodes). But the main purpose remains: ensure a copy of a Pod runs on each relevant node-option A.

NEW QUESTION # 28
.....