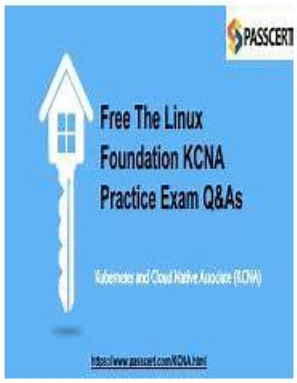
Free PDF Quiz KCNA - Kubernetes and Cloud Native Associate Fantastic Dumps PDF



BTW, DOWNLOAD part of TorrentVCE KCNA dumps from Cloud Storage: https://drive.google.com/open?id=1W_OMzUiv5X7vpwnFZMNJNH4bCVbPipHN

It is generally acknowledged that candidates who earn the Kubernetes and Cloud Native Associate (KCNA) certification ultimately get high-paying jobs in the tech market. Success in the Kubernetes and Cloud Native Associate (KCNA) exam not only validates your skills but also helps you get promotions. To pass the Kubernetes and Cloud Native Associate test in a short time, you must prepare with KCNA Exam Questions that are real and updated. Without studying with KCNA actual questions, candidates fail and waste their time and money.

Linux Foundation KCNA Certification is recognized globally and is highly valued by employers. Kubernetes and Cloud Native Associate certification demonstrates that an individual has the skills and knowledge required to deploy, manage, and scale containerized applications using Kubernetes and cloud-native technologies. This makes them a valuable asset to any organization that is looking to adopt modern application development practices.

>> KCNA Dumps PDF <<

100% Pass Quiz 2026 KCNA: Useful Kubernetes and Cloud Native Associate Dumps PDF

The Kubernetes and Cloud Native Associate (KCNA) web-based practice questions carry the above-mentioned notable features of the desktop-based software. This version of TorrentVCE's Kubernetes and Cloud Native Associate (KCNA) practice questions works on Mac, Linux, Android, iOS, and Windows. Our customer does not need troubling plugins or software installations to

attempt the web-based Linux Foundation in KCNA Practice Questions. Another benefit is that our Linux Foundation KCNA online mock test can be taken via all browsers, including Chrome, MS Edge, Internet Explorer, Safari, Opera, and Firefox.

The KCNA certification exam is a valuable credential for professionals who work with Kubernetes and cloud-native technologies. It is recognized by industry leaders and can help individuals advance their careers in the rapidly growing field of cloud-native computing. Passing KCNA Exam not only demonstrates one's proficiency in Kubernetes and cloud-native technologies, but also proves one's commitment to continuous learning and professional development.

Linux Foundation Kubernetes and Cloud Native Associate Sample Questions (Q98-Q103):

NEW QUESTION # 98
What fields must exist in any Kubernetes object (e.g. YAML) file?

- A. apiVersion, kind, metadata
- B. apiVersion, metadata, namespace
- C. kind, metadata, data
- D. kind, namespace, data

Answer: A

Explanation:
Any Kubernetes object manifest must include apiVersion, kind, and metadata, which makes A correct. This comes directly from how Kubernetes resources are represented and processed by the API server.
apiVersion tells Kubernetes which API group and version should be used to interpret the object (for example v1, apps/v1, batch/v1). This matters because schemas and available fields can change between versions.
kind specifies the type of object you are creating (for example Pod, Service, Deployment, ConfigMap). Kubernetes uses this to route the request to the correct API endpoint and schema.
metadata contains identifying and organizational information such as name, namespace (when namespaced), labels, and annotations. At minimum, most objects require a name; labels and annotations are optional but extremely common for selection and tooling.
A common point of confusion is spec. Many Kubernetes objects include spec because they define desired state (like a Deployment's replica count, Pod template, update strategy). However, the question asks what fields must exist in any Kubernetes object file. Not all objects require a spec in the same way (and some objects include other top-level sections like data for ConfigMaps/Secrets or rules for RBAC objects). The truly universal top-level requirements are the trio in option A.
Options B, C, and D include fields that are not universally required (namespace is not required for cluster-scoped objects, and data only applies to certain kinds like ConfigMaps/Secrets). Therefore, apiVersion + kind + metadata is the correct, general rule and matches Kubernetes object structure.

NEW QUESTION # 99
In CNCF, who develops specifications for industry standards around container formats and runtimes?

- A. Container Runtime Interface (CRI)
- B. Open Container Initiative (OCI)
- C. Container Network Interface (CNI)
- D. Linux Foundation Certification Group (LFCG)

Answer: B

Explanation:
The organization responsible for defining widely adopted standards around container formats and runtime specifications is the Open Container Initiative (OCI), so A is correct. OCI defines the image specification (how container images are structured and stored) and the runtime specification (how to run a container), enabling interoperability across tooling and vendors. This is foundational to the cloud-native ecosystem because it allows different build tools, registries, runtimes, and orchestration platforms to work together reliably.
Within Kubernetes and CNCF-adjacent ecosystems, OCI standards are the reason an image built by one tool can be pushed to a registry and pulled/run by many different runtimes. For example, a Kubernetes node running containerd or CRI-O can run OCI-compliant images consistently. OCI standardization reduces fragmentation and vendor lock-in, which is a core motivation in open source cloud-native architecture.
The other options are not correct for this question. CNI (Container Network Interface) is a standard for configuring container networking, not container image formats and runtimes. CRI (Container Runtime Interface) is a Kubernetes-specific interface

between kubelet and container runtimes-it enables pluggable runtimes for Kubernetes, but it is not the industry standard body for container format/runtime specifications.
"LFCG" is not the recognized standards body here.
In short: OCI defines the "language" for container images and runtime behavior, which is why the same image can be executed across environments. Kubernetes relies on those standards indirectly through runtimes and tooling, but the specification work is owned by OCI. Therefore, the verified correct answer is A.
==========

NEW QUESTION # 100
Which of these is a valid container restart policy?

- A. On update
- B. On login
- C. On failure
- D. On start

Answer: C

Explanation:
The correct answer is D: On failure. In Kubernetes, restart behavior is controlled by the Pod-level field spec.
restartPolicy, with valid values Always, OnFailure, and Never. The option presented here ("On failure") maps to Kubernetes' OnFailure policy. This setting determines what the kubelet should do when containers exit:
* Always: restart containers whenever they exit (typical for long-running services)
* OnFailure: restart containers only if they exit with a non-zero status (common for batch workloads)
* Never: do not restart containers (fail and leave it terminated)
So "On failure" is a valid restart policy concept and the only one in the list that matches Kubernetes semantics.
The other options are not Kubernetes restart policies. "On login," "On update," and "On start" are not recognized values and don't align with how Kubernetes models container lifecycle. Kubernetes is declarative and event-driven: it reacts to container exit codes and controller intent, not user "logins." Operationally, choosing the right restart policy is important. For example, Jobs typically use restartPolicy:
OnFailure or Never because the goal is completion, not continuous uptime. Deployments usually imply
"Always" because the workload should keep serving traffic, and a crashed container should be restarted. Also note that controllers interact with restarts: a Deployment may recreate Pods if they fail readiness, while a Job counts completions and failures based on Pod termination behavior.
Therefore, among the options, the only valid (Kubernetes-aligned) restart policy is D.
==========

NEW QUESTION # 101
Consider the following Kubernetes YAML file representing a deployment:
What is the primary benefit of using a GitOps approach to manage this deployment compared to traditional methods?

- A. GitOps simplifies the process of managing complex Kubernetes deployments by providing a centralized repository for configuration.
- B. GitOps enables easy rollbacks to previous deployments by leveraging the versioning capabilities of Git.
- C. GitOps automates the deployment process, eliminating the need for human intervention.
- D. GitOps provides faster deployment times by eliminating the need for manual interventions.
- E. GitOps enhances security by preventing unauthorized changes to the Kubernetes cluster.

Answer: B

Explanation:
GitOps leverages Git's version control system, providing a history of changes to your cluster's configuration. This enables easy rollbacks to previous deployments by simply reverting to a specific commit in the Git repository, making it easier to recover from errors or unwanted changes.

NEW QUESTION # 102
What is the main purpose of the Ingress in Kubernetes?

- A. Access HTTP and HTTPS services running in the cluster based on their IP address.
- B. Access HTTP and HTTPS services running in the cluster based on their path.
- C. Access services different from HTTP or HTTPS running in the cluster based on their path.
- D. Access services different from HTTP or HTTPS running in the cluster based on their IP address.

Answer: B

Explanation:
D is correct. Ingress is a Kubernetes API object that defines rules for external access to HTTP/HTTPS services in a cluster. The defining capability is Layer 7 routing-commonly host-based and path-based routing-so you can route requests like example.com/app1 to one Service and example.com/app2 to another. While the question mentions "based on their path," that's a classic and correct Ingress use case (and host routing is also common).
Ingress itself is only the specification of routing rules. An Ingress controller (e.g., NGINX Ingress Controller, HAProxy, Traefik, cloud-provider controllers) is what actually implements those rules by configuring a reverse proxy/load balancer. Ingress typically terminates TLS (HTTPS) and forwards traffic to internal Services, giving a more expressive alternative to exposing every service via NodePort/LoadBalancer.
Why the other options are wrong:
A suggests routing by IP address; Ingress is fundamentally about HTTP(S) routing rules (host/path), not direct Service IP access.
B and C describe non-HTTP protocols; Ingress is specifically for HTTP/HTTPS. For TCP/UDP or other protocols, you generally use Services of type LoadBalancer/NodePort, Gateway API implementations, or controller-specific TCP/UDP configuration.
Ingress is a foundational building block for cloud-native application delivery because it centralizes edge routing, enables TLS management, and supports gradual adoption patterns (multiple services under one domain). Therefore, the main purpose described here matches D.

NEW QUESTION # 103
.....