# Linux Foundation
## KCNA
## Kubernetes and Cloud Native Associate

**For More Information – Visit link below:**

## https://www.examsboost.com/

## Product Version

# Latest Version: 10.0

## Question: 1

What native runtime is Open Container Initiative (OCI) compliant?

A. runC
B. runV
C. kata-containers
D. gvisor

### Answer: A

Explanation:
The Open Container Initiative (OCI) publishes open specifications for container images and container runtimes so that tools across the ecosystem remain interoperable. When a runtime is "OCI-compliant," it means it implements the OCI Runtime Specification (how to run a container from a filesystem bundle and configuration) and/or works cleanly with OCI image formats through the usual layers (image → unpack → runtime). runC is the best-known, widely used reference implementation of the OCI runtime specification and is the low-level runtime underneath many higher-level systems. In Kubernetes, you typically interact with a higher-level container runtime (such as containerd or CRI-O) through the Container Runtime Interface (CRI). That higher-level runtime then uses a low-level OCI runtime to actually create Linux namespaces/cgroups, set up the container process, and start it. In many default installations, containerd delegates to runC for this low-level "create/start" work.
The other options are related but differ in what they are: Kata Containers uses lightweight VMs to provide stronger isolation while still presenting a container-like workflow; gVisor provides a user-space kernel for sandboxing containers; these can be used with Kubernetes via compatible integrations, but the canonical "native OCI runtime" answer in most curricula is runC. Finally, "runV" is not a common modern Kubernetes runtime choice in typical OCI discussions. So the most correct, standards-based answer here is A (runC) because it directly implements the OCI runtime spec and is commonly used as the default low-level runtime behind CRI implementations.

## Question: 2

Which API object is the recommended way to run a scalable, stateless application on your cluster?

A. ReplicaSet
B. Deployment
C. DaemonSet
D. Pod

### Answer: B

Explanation:
For a scalable, stateless application, Kubernetes recommends using a Deployment because it provides a higher-level, declarative management layer over Pods. A Deployment doesn't just "run replicas"; it manages the entire lifecycle of rolling out new versions, scaling up/down, and recovering from failures by continuously reconciling the current cluster state to the desired state you define. Under the hood, a Deployment typically creates and manages a ReplicaSet, and that ReplicaSet ensures a specified number of Pod replicas are running at all times. This layering is the key: you get ReplicaSet's self-healing replica maintenance plus Deployment's rollout/rollback strategies and revision history.

Why not the other options? A Pod is the smallest deployable unit, but it's not a scalable controller—if a Pod dies, nothing automatically replaces it unless a controller owns it. A ReplicaSet can maintain N replicas, but it does not provide the full rollout orchestration (rolling updates, pause/resume, rollbacks, and revision tracking) that you typically want for stateless apps that ship frequent releases. A DaemonSet is for node-scoped workloads (one Pod per node or subset of nodes), like log shippers or node agents, not for "scale by replicas."

For stateless applications, the Deployment model is especially appropriate because individual replicas are interchangeable; the application does not require stable network identities or persistent storage per replica. Kubernetes can freely replace or reschedule Pods to maintain availability. Deployment strategies (like RollingUpdate) allow you to upgrade without downtime by gradually replacing old replicas with new ones while keeping the Service endpoints healthy. That combination—declarative desired state, selfhealing, and controlled updates—makes Deployment the recommended object for scalable stateless workloads.

## Question: 3

A CronJob is scheduled to run by a user every one hour. What happens in the cluster when it's time for this CronJob to run?

A. Kubelet watches API Server for CronJob objects. When it's time for a Job to run, it runs the Pod directly.
B. Kube-scheduler watches API Server for CronJob objects, and this is why it's called kube-scheduler.
C. CronJob controller component creates a Pod and waits until it finishes to run.
D. CronJob controller component creates a Job. Then the Job controller creates a Pod and waits until it finishes to run.

## Answer: D

Explanation:
CronJobs are implemented through Kubernetes controllers that reconcile desired state. When the scheduled time arrives, the CronJob controller (part of the controller-manager set of control plane controllers) evaluates the CronJob object's schedule and determines whether a run should be started. Importantly, CronJob does not create Pods directly as its primary mechanism. Instead, it creates a Job object for each scheduled execution. That Job object then becomes the responsibility of the Job controller, which creates one or more Pods to complete the Job's work and monitors them until completion. This separation of concerns is why option D is correct.

This design has practical benefits. Jobs encapsulate "run-to-completion" semantics: retries, backoff limits, completion counts, and tracking whether the work has succeeded. CronJob focuses on the

temporal triggering aspect (schedule, concurrency policy, starting deadlines, history limits), while Job focuses on the execution aspect (create Pods, ensure completion, retry on failure).
Option A is incorrect because kubelet is a node agent; it does not watch CronJob objects and doesn't decide when a schedule triggers. Kubelet reacts to Pods assigned to its node and ensures containers run there. Option B is incorrect because kube-scheduler schedules Pods to nodes after they exist (or are created by controllers); it does not trigger CronJobs. Option C is incorrect because CronJob does not usually create a Pod and wait directly; it delegates via a Job, which then manages Pods and completion. So, at runtime: CronJob controller creates a Job; Job controller creates the Pod(s); scheduler assigns those Pods to nodes; kubelet runs them; Job controller observes success/failure and updates status; CronJob controller manages run history and concurrency rules.

## Question: 4

What is the purpose of the kubelet component within a Kubernetes cluster?

A. A dashboard for Kubernetes clusters that allows management and troubleshooting of applications.
B. A network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.
C. A component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
D. An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

## Answer: D

Explanation:
The kubelet is the primary node agent in Kubernetes. It runs on every worker node (and often on control-plane nodes too if they run workloads) and is responsible for ensuring that containers described by PodSpecs are actually running and healthy on that node. The kubelet continuously watches the Kubernetes API (via the control plane) for Pods that have been scheduled to its node, then it collaborates with the node's container runtime (through CRI) to pull images, create containers, start them, and manage their lifecycle. It also mounts volumes, configures the Pod's networking (working with the CNI plugin), and reports Pod and node status back to the API server.
Option D captures the core: "an agent on each node that makes sure containers are running in a Pod." That includes executing probes (liveness, readiness, startup), restarting containers based on the Pod's restartPolicy, and enforcing resource constraints in coordination with the runtime and OS.
Why the other options are wrong: A describes the Kubernetes Dashboard (or similar UI tools), not kubelet. B describes kube-proxy, which programs node-level networking rules (iptables/ipvs/eBPF depending on implementation) to implement Service virtual IP behavior. C describes the kube-scheduler, which selects a node for Pods that do not yet have an assigned node.
A useful way to remember kubelet's role is: scheduler decides where, kubelet makes it happen there. Once the scheduler binds a Pod to a node, kubelet becomes responsible for reconciling "desired state" (PodSpec) with "observed state" (running containers). If a container crashes, kubelet will restart it according to policy; if an image is missing, it will pull it; if a Pod is deleted, it will stop containers and clean up. This node-local reconciliation loop is fundamental to Kubernetes' self-healing and declarative operation model.

## Question: 5

What is the default value for authorization-mode in Kubernetes API server?

A. --authorization-mode=RBAC
B. --authorization-mode=AlwaysAllow
C. --authorization-mode=AlwaysDeny
D. --authorization-mode=ABAC

## Answer: B

Explanation:
The Kubernetes API server supports multiple authorization modes that determine whether an authenticated request is allowed to perform an action (verb) on a resource. Historically, the API server's default authorization mode was AlwaysAllow, meaning that once a request was authenticated, it would be authorized without further checks. That is why the correct answer here is B.

However, it's crucial to distinguish "default flag value" from "recommended configuration." In production clusters, running with AlwaysAllow is insecure because it effectively removes authorization controls—any authenticated user (or component credential) could do anything the API permits. Modern Kubernetes best practices strongly recommend enabling RBAC (Role-Based Access Control), often alongside Node and Webhook authorization, so that permissions are granted explicitly using Roles/ClusterRoles and RoleBindings/ClusterRoleBindings. Many managed Kubernetes distributions and kubeadm-based setups commonly enable RBAC by default as part of cluster bootstrap profiles, even if the API server's historical default flag value is AlwaysAllow.

So, the exam-style interpretation of this question is about the API server flag default, not what most real clusters should run. With RBAC enabled, authorization becomes granular: you can control who can read Secrets, who can create Deployments, who can exec into Pods, and so on, scoped to namespaces or cluster-wide. ABAC (Attribute-Based Access Control) exists but is generally discouraged compared to RBAC because it relies on policy files and is less ergonomic and less commonly used. AlwaysDeny is useful for hard lockdown testing but not for normal clusters.

In short: AlwaysAllow is the API server's default mode (answer B), but RBAC is the secure, recommended choice you should expect to see enabled in almost any serious Kubernetes environment.

# Thank You for Trying Our Product

**For More Information** – <span style="color:red">**Visit link below:**</span>

## https://www.examsboost.com/

**15 USD Discount Coupon Code:**

## G74JA8UF

# FEATURES

- ✓ **90 Days Free Updates**
- ✓ **Money Back Pass Guarantee**
- ✓ **Instant Download or Email Attachment**
- ✓ **24/7 Live Chat Support**
- ✓ **PDF file could be used at any Platform**
- ✓ **50,000 Happy Customer**