Linux Foundation KCNA Practice Questions

Shared by Reilly on 18-08-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

What is the goal of load balancing?

Options:

A- Automatically measure request performance across instances of an application.
B- Automatically distribute requests across different versions of an application.
C- Automatically distribute instances of an application across the cluster.
D- Automatically distribute requests across instances of an application.

Answer:

D

Explanation:

The core goal of load balancing is to distribute incoming requests across multiple instances of a service so that no single instance becomes overloaded and so that the overall service is more available and responsive. That matches option D, which is the correct answer.

In Kubernetes, load balancing commonly appears through the Service abstraction. A Service selects a set of Pods using labels and provides stable access via a virtual IP (ClusterIP) and DNS name. Traffic sent to the Service is then forwarded to one of the healthy backend Pods. This spreads load across replicas and provides resilience: if one Pod fails, it is removed from endpoints (or becomes NotReady) and traffic shifts to remaining replicas. The actual traffic distribution mechanism depends on the networking implementation (kube-proxy using iptables/IPVS or an eBPF dataplane), but the intent remains consistent: distribute requests across multiple backends.

Option A describes monitoring/observability, not load balancing. Option B describes progressive delivery patterns like canary or A/B routing; that can be implemented with advanced routing layers (Ingress controllers, service meshes), but it's not the general definition of load balancing. Option C describes scheduling/placement of instances (Pods) across cluster nodes, which is the role of the scheduler and controllers, not load balancing.

In cloud environments, load balancing may also be implemented by external load balancers (cloud LBs) in front of the cluster, then forwarded to NodePorts or ingress endpoints, and again balanced internally to Pods. At each layer, the objective is the same: spread request traffic across multiple service instances to improve performance and availability.

=========

Question 2

Question Type: MultipleChoice

Which GitOps engine can be used to orchestrate parallel jobs on Kubernetes?

Options:

A- Jenkins X
B- Flagger
C- Flux
D- Argo Workflows

Answer:

D

Explanation:

Argo Workflows (D) is the correct answer because it is a Kubernetes-native workflow engine designed to define and run multi-step workflows---often with parallelization---directly on Kubernetes. Argo Workflows models workflows as DAGs (directed acyclic graphs) or step-based sequences, where each step is typically a Pod. Because each step is expressed as Kubernetes resources (custom resources), Argo can schedule many tasks concurrently, control fan-out/fan-in patterns, and manage dependencies between steps (e.g., ''run these 10 jobs in parallel, then aggregate results'').

The question calls it a ''GitOps engine,'' but the capability being tested is ''orchestrate parallel jobs.'' Argo Workflows fits because it is purpose-built for running complex job orchestration, including parallel tasks, retries, timeouts, artifacts passing, and conditional execution. In practice, many teams store workflow manifests in Git and apply GitOps practices around them, but the distinguishing feature here is the workflow orchestration engine itself.

Why the other options are not best:

Flux (C) is a GitOps controller that reconciles cluster state from Git; it doesn't orchestrate parallel job graphs as its core function.

Flagger (B) is a progressive delivery operator (canary/blue-green) often paired with GitOps and service meshes/Ingress; it's not a general workflow orchestrator for parallel batch jobs.

Jenkins X (A) is CI/CD-focused (pipelines), not primarily a Kubernetes-native workflow engine for parallel job DAGs in the way Argo Workflows is.

So, the Kubernetes-native tool specifically used to orchestrate parallel jobs and workflows is Argo Workflows (D).

=========

Question 3

Question Type: MultipleChoice

What is CRD?

Options:

A- Custom Resource Definition
B- Custom Restricted Definition
C- Customized RUST Definition
D- Custom RUST Definition

Answer:

A

Explanation:

A CRD is a CustomResourceDefinition, making A correct. Kubernetes is built around an API-driven model: resources like Pods, Services, and Deployments are all objects served by the Kubernetes API. CRDs allow you to extend the Kubernetes API by defining your own resource types. Once a CRD is installed, the API server can store and serve custom objects (Custom Resources) of that new type, and Kubernetes tooling (kubectl, RBAC, admission, watch mechanisms) can interact with them just like built-in resources.

CRDs are a core building block of the Kubernetes ecosystem because they enable operators and platform extensions. A typical pattern is: define a CRD that represents the desired state of some higher-level concept (for example, a database cluster, a certificate request, an application release), and then run a controller (often called an ''operator'') that watches those custom resources and reconciles the cluster to match. That controller may create Deployments, StatefulSets, Services, Secrets, or cloud resources to implement the desired state encoded in the custom resource.

The incorrect answers are made-up expansions. CRDs are not related to Rust in Kubernetes terminology, and ''custom restricted definition'' is not the standard meaning.

So the verified meaning is: CRD = CustomResourceDefinition, used to extend Kubernetes APIs and enable Kubernetes-native automation via controllers/operators.

Question 4

Question Type: MultipleChoice

Which mechanism allows extending the Kubernetes API?

Options:

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.

=========

Question 5

Question Type: MultipleChoice

How do you perform a command in a running container of a Pod?

Options:

A- kubectl exec -- <command>
B- docker exec <command>
C- kubectl run -- <command>
D- kubectl attach -i <command>

Answer:

A

Explanation:

In Kubernetes, the standard way to execute a command inside a running container is kubectl exec, which is why A is correct. kubectl exec calls the Kubernetes API (API server), which then coordinates with the kubelet on the target node to run the requested command inside the container using the container runtime's exec mechanism. The -- separator is important: it tells kubectl that everything after -- is the command to run in the container rather than flags for kubectl itself.

This is fundamentally different from docker exec. In Kubernetes, you don't normally target containers through Docker/CRI tools directly because Kubernetes abstracts the runtime behind CRI. Also, ''Docker'' might not even be installed on nodes in modern clusters (containerd/CRI-O are common). So option B is not the Kubernetes-native approach and often won't work.

kubectl run (option C) is for creating a new Pod (or generating workload resources), not for executing a command in an existing container. kubectl attach (option D) attaches your terminal to a running container's process streams (stdin/stdout/stderr), which is useful for interactive sessions, but it does not execute an arbitrary new command like exec does.

In real usage, you often specify the container when a Pod has multiple containers: kubectl exec -it -c <container> -- /bin/sh. This is common for debugging, verifying config files mounted from ConfigMaps/Secrets, testing DNS resolution, or checking network connectivity from within the Pod network namespace. Because exec uses the API and kubelet, it respects Kubernetes access control (RBAC) and audit logging---another reason it's the correct operational method.

=========

Question 6

Question Type: MultipleChoice

Why is Cloud-Native Architecture important?

Options:

A- Cloud Native Architecture revolves around containers, microservices and pipelines.
B- Cloud Native Architecture removes constraints to rapid innovation.
C- Cloud Native Architecture is modern for application deployment and pipelines.
D- Cloud Native Architecture is a bleeding edge technology and service.

Answer:

B

Explanation:

Cloud-native architecture is important because it enables organizations to build and run software in a way that supports rapid innovation while maintaining reliability, scalability, and efficient operations. Option B best captures this: cloud native removes constraints to rapid innovation, so B is correct.

In traditional environments, innovation is slowed by heavyweight release processes, tightly coupled systems, manual operations, and limited elasticity. Cloud-native approaches---containers, declarative APIs, automation, and microservices-friendly patterns---reduce those constraints. Kubernetes exemplifies this by offering a consistent deployment model, self-healing, automated rollouts, scaling primitives, and a large ecosystem of delivery and observability tools. This makes it easier to ship changes more frequently and safely: teams can iterate quickly, roll back confidently, and standardize operations across environments.

Option A is partly descriptive (containers/microservices/pipelines are common in cloud native), but it doesn't explain why it matters; it lists ingredients rather than the benefit. Option C is vague (''modern'') and again doesn't capture the core value proposition. Option D is incorrect because cloud native is not primarily about being ''bleeding edge''---it's about proven practices that improve time-to-market and operational stability.

A good way to interpret ''removes constraints'' is: cloud native shifts the bottleneck away from infrastructure friction. With automation (IaC/GitOps), standardized runtime packaging (containers), and platform capabilities (Kubernetes controllers), teams spend less time on repetitive manual work and more time delivering features. Combined with observability and policy automation, this results in faster delivery with better reliability---exactly the reason cloud-native architecture is emphasized across the Kubernetes ecosystem.

=========

Question 7

Question Type: MultipleChoice

In Kubernetes, which command is the most efficient way to check the progress of a Deployment rollout and confirm if it has completed successfully?

Options:

A- kubectl get deployments --show-labels -o wide
B- kubectl describe deployment my-deployment --namespace=default
C- kubectl logs deployment/my-deployment --all-containers=true
D- kubectl rollout status deployment/my-deployment

Answer:

D

Explanation:

When performing rolling updates in Kubernetes, it is important to have a clear and efficient way to track the progress of a Deployment rollout and determine whether it has completed successfully. The most direct and purpose-built command for this task is kubectl rollout status deployment/my-deployment, making option D the correct answer.

The kubectl rollout status command is specifically designed to monitor the state of rollouts for resources such as Deployments, StatefulSets, and DaemonSets. It provides real-time feedback on the rollout process, including whether new Pods have been created, old Pods are being terminated, and if the desired number of updated replicas has become available. The command blocks until the rollout either completes successfully or fails, which makes it especially useful in automation and CI/CD pipelines.

Option A is incorrect because kubectl get deployments only provides a snapshot view of deployment status fields and does not actively track rollout progress. Option B can provide detailed information and events, but it is verbose and not optimized for quickly confirming rollout completion. Option C is incorrect because Deployment objects themselves do not produce logs; logs are generated by Pods and containers, not higher-level workload resources.

The rollout status command also integrates with Kubernetes' revision history, ensuring that it accurately reflects the current state of the Deployment's update strategy. If a rollout is stuck due to failed Pods, readiness probe failures, or resource constraints, the command will indicate that the rollout is not progressing, helping operators quickly identify issues.

In summary, kubectl rollout status deployment/my-deployment is the most efficient and reliable

way to check rollout progress and confirm success. It is purpose-built for rollout tracking, easy to interpret, and widely used in production Kubernetes workflows, making Option D the correct and verified answer.

Question 8

Question Type: MultipleChoice

What is the name of the Kubernetes resource used to expose an application?

Options:

A- Port
B- Service
C- DNS
D- Deployment

Answer:

B

Explanation:

To expose an application running on Pods so that other components can reliably reach it, Kubernetes uses a Service, making B the correct answer. Pods are ephemeral: they can be recreated, rescheduled, and scaled, which means Pod IPs change. A Service provides a stable endpoint (virtual IP and DNS name) and load-balances traffic across the set of Pods selected by its label selector.

Services come in multiple forms. The default is ClusterIP, which exposes the application inside the cluster. NodePort exposes the Service on a static port on each node, and LoadBalancer (in supported clouds) provisions an external load balancer that routes traffic to the Service. ExternalName maps a Service name to an external DNS name. But across these variants, the abstraction is consistent: a Service defines how to access a logical group of Pods.

Option A (Port) is not a Kubernetes resource type; ports are fields within resources. Option C (DNS) is a supporting mechanism (CoreDNS creates DNS entries for Services), but DNS is not the resource you create to expose the app. Option D (Deployment) manages Pod replicas and rollouts, but it does not directly provide stable networking access; you typically pair a Deployment with a Service to expose it.

This is a core cloud-native pattern: controllers manage compute, Services manage stable connectivity, and higher-level gateways like Ingress provide L7 routing for HTTP/HTTPS. So, the

Kubernetes resource used to expose an application is Service (B).

=========

Question 9

Question Type: MultipleChoice

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

Options:

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, self-healing, and controlled updates---makes Deployment the recommended object for scalable stateless workloads.

=========

Question 10

Question Type: MultipleChoice

Which option best is a recommended security habit in Kubernetes?

Options:

A- Run the containers as the user with group ID 0 (root) and any user ID.
B- Disallow privilege escalation from within a container as the default option.
C- Run the containers as the user with user ID 0 (root) and any group ID.
D- Allow privilege escalation from within a container as the default option.

Answer:

B

Explanation:

The correct answer is B. A widely recommended Kubernetes security best practice is to disallow privilege escalation inside containers by default. In Kubernetes Pod/Container security context, this is represented by allowPrivilegeEscalation: false. This setting prevents a process from gaining more privileges than its parent process---commonly via setuid/setgid binaries or other privilege-escalation mechanisms. Disallowing privilege escalation reduces the blast radius of a compromised container and aligns with least-privilege principles.

Options A and C are explicitly unsafe because they encourage running as root (UID 0 and/or GID 0). Running containers as root increases risk: if an attacker breaks out of the application process or exploits kernel/runtime vulnerabilities, having root inside the container can make privilege escalation and lateral movement easier. Modern Kubernetes security guidance strongly favors running as non-root (runAsNonRoot: true, explicit runAsUser), dropping Linux capabilities, using read-only root filesystems, and applying restrictive seccomp/AppArmor/SELinux profiles where possible.

Option D is the opposite of best practice. Allowing privilege escalation by default increases the attack surface and violates the idea of secure defaults.

Operationally, this habit is often enforced via admission controls and policies (e.g., Pod Security Admission in ''restricted'' mode, or policy engines like OPA Gatekeeper/Kyverno). It's also important for compliance: many security baselines require containers to run as non-root and to prevent privilege escalation.

So, the recommended security habit among the choices is clearly B: Disallow privilege escalation.

=========

Question 11

Question Type: MultipleChoice

In Kubernetes, what is the main purpose of creating a Service resource for a Deployment?

Options:

A- To centrally manage and apply runtime configuration values for application components.
B- To provide a stable endpoint for accessing Pods even when their IP addresses change.
C- To automatically adjust the number of Pods based on CPU or memory utilization metrics.
D- To define and attach persistent volumes that store application data across Pod restarts.

Answer:

B

Explanation:

In Kubernetes, Pods are inherently ephemeral. They can be created, destroyed, restarted, or rescheduled at any time, and each time this happens, a Pod may receive a new IP address. This dynamic behavior is essential for resilience and scalability, but it also creates a challenge for reliably accessing application workloads. The Service resource addresses this problem by providing a stable network endpoint for a group of Pods, making option B the correct answer.

A Service selects Pods using label selectors---typically the same labels applied by a Deployment---and exposes them through a consistent virtual IP address (ClusterIP) and DNS name. Regardless of how many Pods are running or whether individual Pods are replaced, the Service remains stable and automatically routes traffic to healthy Pods. This abstraction allows clients to communicate with an application without needing to track individual Pod IPs.

Deployments are responsible for managing the lifecycle of Pods, including scaling, rolling updates, and self-healing. However, Deployments do not provide networking or service

discovery capabilities. Without a Service, consumers would need to directly reference Pod IPs, which would break as soon as Pods are rescheduled or updated.

Option A is incorrect because centralized configuration management is handled using ConfigMaps and Secrets, not Services. Option C is incorrect because automatic scaling based on CPU or memory is the responsibility of the Horizontal Pod Autoscaler (HPA), not Services. Option D is incorrect because persistent storage is managed using PersistentVolume and PersistentVolumeClaim resources, which are unrelated to Services.

Services can be configured for different access patterns, such as ClusterIP for internal communication, NodePort or LoadBalancer for external access, and headless Services for direct Pod discovery. Despite these variations, their core purpose remains the same: providing a reliable and stable way to access Pods managed by a Deployment.

Therefore, the correct and verified answer is Option B, which aligns with Kubernetes networking fundamentals and official documentation.

Question 12

Question Type: MultipleChoice

Which option best is a feature Kubernetes provides by default as a container orchestration tool?

Options:

A- A portable operating system.
B- File system redundancy.
C- A container image registry.
D- Automated rollouts and rollbacks.

Answer:

D

Explanation:

Kubernetes provides automated rollouts and rollbacks for workloads by default (via controllers like Deployments), so D is correct. In Kubernetes, application delivery is controller-driven: you declare the desired state (new image, new config), and controllers reconcile the cluster toward that state. Deployments implement rolling updates, gradually replacing old Pods with new ones while respecting availability constraints. Kubernetes tracks rollout history and supports rollback to previous ReplicaSets when an update fails or is deemed unhealthy.

This is a core orchestration capability: it reduces manual intervention and makes change safer. Rollouts use readiness checks and update strategies to avoid taking the service down, and kubectl rollout status/history/undo supports day-to-day release operations.

The other options are not ''default Kubernetes orchestration features'':

Kubernetes is not a portable operating system (A). It's a platform for orchestrating containers on top of an OS.

Kubernetes does not provide filesystem redundancy by itself (B). Storage redundancy is handled by underlying storage systems and CSI drivers (e.g., replicated block storage, distributed filesystems).

Kubernetes does not include a built-in container image registry (C). You use external registries (Docker Hub, ECR, GCR, Harbor, etc.). Kubernetes pulls images but does not host them as a core feature.

So the correct ''provided by default'' orchestration feature in this list is the ability to safely manage application updates via automated rollouts and rollbacks.

=========