PassQueen
a.k.a dumpsqueen.com

Kubernetes and Cloud Native Associate (KCNA)

Linux Foundation KCNA

Version Demo

Total Demo Questions: 39

Total Premium Questions: 390

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Kubernetes Fundamentals	223
Topic 2, Container Orchestration	44
Topic 3, Cloud Native Architecture	56
Topic 4, Cloud Native Observability	33
Topic 5, Cloud Native Application Delivery	34
Total	390

QUESTION NO: 1

Which statements about the HorizontalPodAutoscaler are correct?

A. It can adjust the number of replicas of a workload

B. It can use CPU utilization metrics

C. It can use custom metrics

D. It increases memory assigned to each individual Pod

E. It replaces the kube-scheduler

ANSWER: A B C

Explanation:

It can adjust the number of replicas of a workload is correct because the HorizontalPodAutoscaler changes the replica count of supported scalable resources, such as Deployments and StatefulSets. **It can use CPU utilization metrics** is correct when resource metrics are available. **It can use custom metrics** is also correct when the appropriate metrics APIs are configured. The HorizontalPodAutoscaler does not increase CPU or memory assigned to an individual Pod; that is vertical scaling. See the Kubernetes Horizontal Pod Autoscaling documentation.

QUESTION NO: 2

What kind of limitation cgroups allows?

A. Prioritization

B. Resource limiting

C. Accounting

D. None of the options

E. Control

F. Server cpu and memory

ANSWER: A B C E

Explanation:

Control groups (cgroups) are a Linux kernel mechanism for organizing processes into hierarchical groups and applying resource-management policies to each group. **Resource limiting** is a central cgroup capability: administrators or an orchestrator can constrain a group's use of resources such as CPU time, memory, I/O bandwidth, and process identifiers. **Prioritization** is supported by assigning relative resource weights or shares, allowing one workload group to receive preferential access when resources are contended. **Accounting** is also a core use: cgroups expose resource-usage statistics for a group, enabling monitoring, chargeback, capacity analysis, and informed scheduling decisions. Finally, **Control** accurately describes cgroups' broader role because controllers apply policies and restrictions to member processes, including controls that can suspend, isolate, or otherwise manage workloads. These capabilities are particularly important for containers, where a runtime uses cgroups to ensure that one container's activity is governed separately from another's and does not consume unrestricted host resources. The Linux kernel's cgroup documentation describes cgroups as a way to organize processes hierarchically and distribute system resources in a controlled manner. See the Linux kernel cgroup v2 documentation and the Kubernetes cgroups documentation.

QUESTION NO: 3

What is a sidecar container?

A. A Pod that runs next to another container within the same Pod.

B. A container that runs next to another Pod within the same namespace.

C. A container that runs next to another container within the same Pod.

D. A Pod that runs next to another Pod within the same namespace.

ANSWER: C

Explanation:

A sidecar container is a helper container that runs alongside an application container within the same Kubernetes Pod. This arrangement lets the containers share the Pod's network namespace, so they can communicate through `localhost`, and they can also share data through volumes defined for that Pod. The sidecar pattern adds reusable supporting capabilities without requiring changes to the primary application's code or container image.

For example, a sidecar may collect and forward logs, proxy inbound and outbound traffic, refresh configuration, provide credential-related functionality, or expose monitoring data. Because the application and sidecar are part of one Pod, Kubernetes schedules them together onto the same node and manages them as components of the same workload. Their resource requests and limits must therefore be considered together when sizing the Pod.

Kubernetes documentation describes a sidecar as a container that extends or enhances the primary application container, commonly by providing services such as logging, monitoring, or network proxying. Kubernetes also supports native sidecar behavior through restartable init containers for workloads that need controlled startup and shutdown ordering. See the Kubernetes Sidecar Containers documentation and the Pods concept documentation.

QUESTION NO: 4

Open Container Initiative set container standards for

A. Code, Build, Distribute, Deploy containers

B. Run, build, and image

C. Code, Build, Distribute containers

D. Run, Build, Distribute containers

E. Container runtime, image, and distribution specifications

ANSWER: E

Explanation:

The Open Container Initiative sets open, vendor-neutral specifications for container images, container runtimes, and image distribution. Its Image Specification defines the format and metadata for a container image, enabling tools to create, package, and consume images consistently. Its Runtime Specification defines how a compliant runtime should execute an unpacked container filesystem bundle, including the configuration needed to create and run the container process. The Distribution Specification standardizes the API interactions used to push and pull content, such as container images, between clients and registries. Together, these specifications make container artifacts portable across different build tools, registries, and runtimes, rather than tying users to a particular vendor implementation. OCI does not define a general container build standard; build tooling may produce OCI-compliant images, but image creation itself is not the subject of a separate OCI build specification. The OCI project describes its work as creating open industry standards around container formats and runtimes, with the Image, Runtime, and Distribution specifications as its core specifications. See the Open Container Initiative and the OCI Image Specification.

QUESTION NO: 5

What can Kubernetes admission controllers do when an API request is submitted?

A. Validate an API request

B. Modify an API request

C. Enforce organizational policies

D. Select a Node for an unscheduled Pod

ANSWER: A B C

Explanation:

Validate an API request and **modify an API request** are correct. Admission controllers run after authentication and authorization but before an object is persisted. Validating admission can reject requests that do not meet policy requirements, while mutating admission can modify an object before it is stored, such as by applying default values or injecting configuration.

Enforce organizational policies is also correct because admission control can prevent noncompliant workload configurations, such as requests that violate security or resource policies. Admission controllers do not select Nodes for Pods; scheduling is performed by kube-scheduler. See the Kubernetes admission controllers documentation.

QUESTION NO: 6

Which of the following sentences is true about namespaces in Kubernetes?

A. You can create a namespace within another namespace in Kubernetes.

B. You can create two resources of the same kind and name in a namespace.

C. The default namespace exists when a new cluster is created.

D. All the objects in the cluster are namespaced by default.

ANSWER: C

Explanation:

The default namespace exists when a new cluster is created. Kubernetes automatically creates a set of initial namespaces as part of cluster setup, including `default`, `kube-system`, `kube-public`, and `kube-node-lease`. The `default` namespace is intended as the general-purpose namespace for user workloads that are not explicitly assigned to another namespace.

Namespace selection is part of the metadata for namespaced Kubernetes objects. When a command, manifest, or API request creates a namespaced resource without specifying a namespace, Kubernetes generally uses `default` as the target namespace. For example, a Deployment manifest with no `metadata.namespace` field is created in `default`, unless a client-side namespace context has been selected. This behavior makes the namespace immediately available for basic workload creation after cluster initialization.

Namespaces provide a logical scope for many workload-related resources and are commonly used to separate teams, applications, or environments. Kubernetes documentation identifies `default` as the namespace for objects with no other namespace specified. See the official Namespaces documentation and the initial namespaces reference.

QUESTION NO: 7

What is the main purpose of the Open Container Initiative (OCI)?

A. Accelerating the adoption of containers and Kubernetes in the industry.

B. Creating open industry standards around container formats and runtimes.

C. Creating industry standards around container formats and runtimes for private purposes.

D. Improving the security of standards around container formats and runtimes.

ANSWER: B

Explanation:

The correct answer is **"Creating open industry standards around container formats and runtimes."** The Open Container Initiative is an open-governance project hosted by the Linux Foundation that creates vendor-neutral, interoperable specifications for containers. Its core specifications define the standard image format used to package and distribute container images, the runtime behavior for unpacking and running those images, and image distribution mechanisms. These standards let images built with one compliant tool work with compatible registries and runtimes, reducing dependency on a single vendor or implementation.

OCI does not itself provide a particular container runtime or orchestration platform. Instead, it establishes the common technical contracts that implementations can follow. For example, container engines and low-level runtimes can implement the OCI Image Specification and OCI Runtime Specification so that a standardized image can be executed consistently across different environments. This interoperability is foundational to the broader cloud-native container ecosystem and supports portability of container workloads. OCI specifications and their governance are publicly developed and available for adoption by the industry. See the Open Container Initiative and the official OCI Runtime Specification repository.

QUESTION NO: 8

What is a benefits of Kubernetes federation?

A. Avoids scalability limits on pods and nodes

B. Creates highly available clusters in different regions

C. Low latency

ANSWER: A B C

Explanation:

Kubernetes federation coordinates workloads and configuration across multiple independent Kubernetes clusters. "Avoids scalability limits on pods and nodes" is a benefit because applications can be distributed among member clusters rather than being constrained by the capacity, control-plane scale, or failure domain of one cluster. This multi-cluster model supports growth beyond what is practical to operate as a single cluster.

"Creates highly available clusters in different regions" is also a benefit because workloads can be deployed across geographically separate clusters. If a region or an individual cluster becomes unavailable, traffic and application availability can be supported by healthy clusters elsewhere, provided the application, networking, and traffic-routing design account for failover.

"Low latency" follows from placing workload replicas in clusters closer to users or dependent services. Federation enables location-aware deployment patterns, reducing network distance and improving response times for distributed users. Kubernetes describes federation as a way to build and manage a set of clusters and spread workloads across environments; its original federation design also identifies cross-region availability and placement as key use cases. See the Kubernetes federation overview and the KubeFed project documentation.

QUESTION NO: 9

Which practices support least-privilege access for Kubernetes workloads?

A. Using a dedicated ServiceAccount

B. Granting only required RBAC permissions

C. Disabling automatic token mounting when API access is unnecessary

D. Granting cluster-admin access to every workload

E. Sharing one administrative credential among all applications

ANSWER: A B C

Explanation:

Using a dedicated ServiceAccount is correct because a workload can be assigned an identity separate from the default ServiceAccount. **Granting only required RBAC permissions** supports least privilege by limiting API access to the specific resources and verbs a workload needs. **Disabling automatic ServiceAccount token mounting when API access is unnecessary** reduces unnecessary credentials available to a Pod. Granting cluster-admin access to an application workload violates least privilege because it provides broad control over cluster resources. See the Kubernetes ServiceAccounts documentation and the Kubernetes RBAC documentation.

QUESTION NO: 10

Kubernetes Secrets are specifically intended to hold confidential data. Which API object should be used to hold **non-confidential** data?

A. CNI

B. CSI

C. ConfigMaps

D. RBAC

ANSWER: C

Explanation:

ConfigMaps are the Kubernetes API object intended for non-confidential configuration data. They let you separate environment-specific or operational settings from a container image, so the same application image can be deployed with different configuration in development, testing, and production. Typical ConfigMap content includes service endpoint URLs, ports, feature flags, command-line settings, and complete configuration files.

A Pod can consume ConfigMap values as environment variables, command arguments, or files mounted into the container filesystem. This makes configuration available to workloads without baking it into application code or rebuilding the image for each configuration change. ConfigMaps therefore support the cloud-native practice of externalizing configuration and improve workload portability and maintainability.

ConfigMaps are specifically for data that does not require secrecy. Their contents should be treated as readable configuration rather than as protected credentials or key material. Kubernetes documentation describes ConfigMaps as an API object for storing non-confidential data in key-value pairs and notes that they can be used by Pods or other system components. See the Kubernetes ConfigMap documentation and the Secret documentation for the intended distinction between ordinary configuration and sensitive data.

QUESTION NO: 11

Which type of Service requires manual creation of Endpoints?

A. LoadBalancer

B. Services without selectors

C. NodePort

D. ClusterIP with selectors

ANSWER: B

Explanation:

Services without selectors require manual management of their backend endpoints. In the usual Service pattern, a selector identifies Pods by label, and Kubernetes automatically discovers the matching ready Pods and maintains the corresponding EndpointSlices (and, for compatibility in some cases, Endpoints). This lets the Service provide a stable virtual IP address and DNS name even as Pods are created, removed, or replaced.

A Service that has no selector does not identify any Pods for Kubernetes to track. It is therefore useful when the actual backend is outside the cluster, is not represented by Pods selected through labels, or is managed through a custom mechanism. To direct traffic for such a Service, an administrator or controller must create and maintain the endpoint data explicitly, typically using EndpointSlice resources. The endpoint entries supply the backend IP addresses and ports associated with the Service, allowing in-cluster clients to access those backends through the Service name.

Kubernetes documents this pattern as “Services without selectors” and notes that EndpointSlices must be created manually for them. See the Kubernetes Services without selectors documentation and the EndpointSlices documentation.

QUESTION NO: 12

What are the two major components of service mesh?

A. Control plane and Data plane

B. Master plane and Data plane

C. None of the options

D. Controller plane and User plane

E. Master plane and User plane

ANSWER: A

Explanation:

Control plane and Data plane are the two foundational components of a service mesh. The data plane consists of proxies deployed alongside, or sometimes integrated with, application workloads. These proxies intercept service-to-service traffic and apply capabilities such as traffic routing, load balancing, mutual TLS encryption, authentication and authorization enforcement, telemetry collection, and resiliency behaviors. The control plane centrally configures and manages those proxies. It distributes policies, service-discovery information, certificates, routing rules, and other configuration needed for the data plane to handle traffic consistently across the environment. This separation lets application teams focus on business logic while the service mesh provides standardized networking, security, and observability functions for communication between services. Kubernetes documentation describes service meshes as providing a dedicated infrastructure layer for service-to-service communication and commonly using proxies to manage that traffic. The CNCF service-mesh landscape and educational materials also characterize meshes using the distinction between a management-oriented control plane and a traffic-handling data plane. See the Kubernetes overview of service meshes and the CNCF explanation of service mesh architecture.

QUESTION NO: 13

What is the reference implementation of the OCI runtime specification?

A. lxc

B. CRI-O

C. runc

D. Docker

ANSWER: C

Explanation:

The correct answer is **runc**. The Open Container Initiative (OCI) Runtime Specification defines a standard for how a container runtime should create and run a container on an operating system. This includes configuring the container filesystem, process, Linux namespaces, cgroups, capabilities, and other isolation-related settings described in an OCI runtime bundle. runc is the reference implementation maintained for this specification and is designed to create and run containers in accordance with it.

In a typical Kubernetes node architecture, a higher-level Kubernetes Container Runtime Interface implementation manages requests from the kubelet and coordinates container lifecycle operations. That runtime can use an OCI-compliant low-level runtime such as runc to perform the actual creation and execution of the isolated container process. This separation allows OCI-compatible images and runtime bundles to be used consistently across different container tooling and platforms. The OCI Runtime Specification repository identifies runc as its reference implementation, while the runc project describes itself as a CLI tool for spawning and running containers according to the OCI specification.

References: OCI Runtime Specification and runc project.

QUESTION NO: 14

Which of the following are benefits of using a declarative approach to manage Kubernetes resources?

A. Version control of configuration

B. Repeatable deployment across environments

C. Continuous reconciliation of desired state

D. Elimination of access-control requirements

ANSWER: A B C

Explanation:

Version control of configuration is a benefit because manifests can be stored, reviewed, and tracked in a source repository. **Repeatable deployment across environments** is supported because the same declared configuration can be applied consistently, with environment-specific values managed separately when needed. **Continuous reconciliation of desired state** is central to Kubernetes: controllers work to make actual state match the state defined in API objects.

Declarative management does not eliminate the need for access control. Authorization remains necessary to control who can create, modify, or delete resources. See the Kubernetes declarative management documentation.

QUESTION NO: 15

What does “continuous” mean in the context of CI/CD?

A. Frequent releases, manual processes, repeatable, fast processing

B. Periodic releases, manual processes, repeatable, automated processing

C. Frequent releases, automated processes, repeatable, fast processing

D. Periodic releases, automated processes, repeatable, automated processing

ANSWER: C

Explanation:

“Continuous” in CI/CD describes a software delivery approach in which teams make small, frequent changes and use automated, repeatable processes to build, test, validate, and deliver those changes quickly. Therefore, “Frequent releases, automated processes, repeatable, fast processing” best captures the intended meaning. Continuous integration emphasizes integrating code changes into a shared codebase frequently, with automated builds and tests providing rapid feedback that

helps keep the main branch in a deployable state. Continuous delivery extends this practice by ensuring that validated changes can move through a reliable delivery pipeline with minimal delay and a consistent process.

The central goal is not simply releasing on a particular calendar schedule; it is reducing the time and risk between a code change and useful feedback or delivery. Automation makes each pipeline execution consistent, while repeatability ensures that the same defined steps can be used across changes and environments. Fast feedback enables developers to detect and address issues while changes are small and easier to diagnose. The CNCF describes CI/CD as an approach that automates integration, testing, and delivery activities to support frequent, reliable software releases. Kubernetes-oriented workflows commonly pair these pipelines with declarative deployment configuration and health checks to make delivery safer and more predictable. See the CNCF Continuous Integration glossary and the CNCF Continuous Delivery glossary.

QUESTION NO: 16

What tool allows us to build useful visual representations of prometheus data?

A. Grafana

B. kubectl

C. Distributed system tracing

D. Rook

E. Kibana

ANSWER: A

Explanation:

Grafana is the correct tool because it is commonly used to query, visualize, and create dashboards from Prometheus metrics. Prometheus collects and stores time-series metric data, while Grafana provides the visualization layer that turns those metrics into useful graphs, gauges, tables, heatmaps, and alerting views. In a typical cloud-native monitoring setup, Grafana is configured with Prometheus as a data source; dashboard panels then use PromQL queries to retrieve the relevant metrics from Prometheus. This enables teams to build operational dashboards for information such as resource utilization, application request rates, latency, error rates, and Kubernetes cluster health. Grafana also supports reusable and importable dashboards, making it practical to standardize monitoring views across services and environments. Prometheus documentation specifically lists Grafana as a visualization option and provides guidance for using it with Prometheus. Grafana's documentation likewise describes Prometheus as a supported data source for building dashboards and exploring metric data. See the Prometheus Grafana visualization documentation and the Grafana Prometheus data source documentation.

QUESTION NO: 17

Notary and the update framework leading security projects in CNCF

A. TRUE

B. FALSE

ANSWER: A

Explanation:

TRUE is correct. Notary and The Update Framework (TUF) are CNCF security-focused projects that address an important cloud native security problem: ensuring that software artifacts and updates can be trusted before they are deployed. Notary provides mechanisms for publishing and verifying signed content, helping users establish the authenticity and integrity of container images and other artifacts. TUF provides a secure framework for distributing software updates, with protections designed to reduce risks such as compromised signing keys, replayed metadata, and rollback attacks. These capabilities are especially relevant in Kubernetes environments, where workloads are commonly assembled from container images and deployed through automated delivery pipelines. Together, signed artifact verification and secure update metadata support

software supply-chain security by enabling consumers to validate what they receive and whether it is current and authorized. CNCF recognizes both projects within its cloud native ecosystem, and their roles align directly with the security category represented in the CNCF landscape. See the CNCF TUF project page and the CNCF Cloud Native Landscape for project and ecosystem information.

QUESTION NO: 18

In Kubernetes, which abstraction defines a logical set of Pods and a policy by which to access them?

A. Service Account

B. NetworkPolicy

C. Service

D. Custom Resource Definition

ANSWER: C

Explanation:

Service is the Kubernetes abstraction defined as a logical set of Pods together with a policy for accessing those Pods. Pods are deliberately ephemeral: a Pod may be replaced, rescheduled, scaled, or receive a different IP address during normal cluster operation. A Service decouples clients from those changing Pod addresses by supplying a stable network endpoint, commonly a cluster-internal virtual IP address and DNS name.

Typically, a Service uses a label selector to identify its backing Pods. Kubernetes continuously discovers the matching ready endpoints and represents them through EndpointSlices. Traffic sent to the Service's address and port is routed by the cluster networking implementation to one of the current backend endpoints. This lets applications reliably communicate with a workload through its Service name rather than discovering and calling individual Pod IP addresses.

The definition used in this question is Kubernetes' own description of a Service: it exposes an application running on one or more Pods as a network service. Services can support several exposure models, including internal-only access through ClusterIP and externally reachable configurations such as NodePort or LoadBalancer, while retaining the same core role of stable access to a logical Pod set.

References: Kubernetes Documentation: Service and Kubernetes Documentation: Services, Load Balancing, and Networking.

QUESTION NO: 19

Which of the following is a valid PromQL query?

A. SELECT * from http_requests_total WHERE job=apiserver

B. http_requests_total WHERE (job="apiserver")

C. SELECT * from http_requests_total

D. http_requests_total(job="apiserver")

E. http_requests_total{job="apiserver"}

ANSWER: E

Explanation:

`http_requests_total{job="apiserver"}` is a valid PromQL instant-vector selector. In PromQL, a metric name such as `http_requests_total` can be followed by a set of label matchers enclosed in curly braces. The matcher `job="apiserver"` selects only those time series whose `job` label has the exact value `apiserver`. The resulting query returns the current sample value for every matching time series, preserving any other labels that distinguish those series.

This selector format is fundamental to Prometheus because metrics are multidimensional: one metric name can represent many separate time series, identified by label sets. Curly-brace label matchers let users narrow that set before applying PromQL functions, range selectors, aggregations, recording rules, dashboard visualizations, or alert expressions. For example, this selector could be used as the input to a rate calculation: `rate(http_requests_total{job="apiserver"}[5m])`.

Prometheus documents instant vector selectors as a metric name optionally followed by label matchers in curly braces, and documents equality matching with the = operator. See the Prometheus querying basics and the PromQL operators documentation.

QUESTION NO: 20

Which statements about Kubernetes NetworkPolicies are correct?

A. They can control ingress traffic to selected Pods

B. They can control egress traffic from selected Pods

C. They require a network plugin that supports enforcement

D. They replace Services for Pod discovery

ANSWER: A B C

Explanation:

NetworkPolicies can control ingress traffic and **can control egress traffic** for selected Pods. They use label selectors to identify the Pods to which a policy applies and can define allowed traffic based on peers, ports, and protocols. A NetworkPolicy can also implement a default-deny approach by selecting Pods while defining no allowed ingress or egress rules.

NetworkPolicies require a network plugin that implements NetworkPolicy enforcement. They do not replace Kubernetes Services, which provide stable service discovery and traffic routing to Pods. See the Kubernetes NetworkPolicy documentation.

QUESTION NO: 21

What are valid uses of Kubernetes labels?

A. Organizing and selecting objects

B. Grouping objects for queries

C. Storing confidential credentials

D. Granting API permissions to users

E. Associating a Service with backend Pods

ANSWER: A B E

Explanation:

Organizing and selecting objects is correct because labels are key-value metadata that can identify and categorize Kubernetes objects. Kubernetes selectors use labels to associate resources, such as a Service selecting its backend Pods or a Deployment managing Pods matching its selector. **Grouping objects for queries** is also correct because users can filter resources with commands such as `kubectl get pods -l app=web`. Labels are intended to identify attributes that are meaningful for users and tooling, such as application name, environment, or release version. Labels do not store sensitive configuration data and do not directly provide authorization. See the Kubernetes labels and selectors documentation.

QUESTION NO: 22

What is Helm?

A. An open source dashboard for Kubernetes.

B. A package manager for Kubernetes applications.

C. A custom scheduler for Kubernetes.

D. An end-to-end testing project for Kubernetes applications.

ANSWER: B

Explanation:

Helm is a package manager for Kubernetes applications. It packages the Kubernetes resource definitions and configuration needed to deploy an application into a reusable unit called a *chart*. A chart can contain templates for resources such as Deployments, Services, ConfigMaps, and Ingresses, along with default configurable values. This approach lets teams install, upgrade, roll back, share, and consistently manage complex applications across Kubernetes clusters using Helm commands. For example, rather than manually applying many separate YAML manifests, an operator can install a chart as a named release and provide environment-specific configuration through a values file or command-line settings. Helm also supports chart repositories, which make versioned application packages available for discovery and installation. The Helm project describes itself as “the package manager for Kubernetes,” and its documentation explains that charts are the packaging format Helm uses to define, install, and upgrade Kubernetes applications. Learn more in the official Helm documentation and the Helm charts documentation.

QUESTION NO: 23

What are container runtimes with Kubernetes?

A. CRI-O

B. lxd

C. containerd

D. Dockershim

ANSWER: A C

Explanation:

CRI-O and **containerd** are container runtimes used with Kubernetes. Kubernetes does not directly create and run containers itself; instead, the kubelet communicates with a runtime on each node through the Container Runtime Interface (CRI). That runtime is responsible for pulling container images, creating and starting containers, stopping them when requested, and reporting container and image status back to the kubelet.

containerd is an industry-standard runtime that implements CRI support and is commonly deployed as the runtime for Kubernetes worker nodes. It manages the complete container lifecycle and uses lower-level runtime components to execute containers. **CRI-O** is another CRI-compatible runtime designed specifically for Kubernetes. It focuses on running Open Container Initiative-compatible container images and integrates directly with Kubernetes through the standard CRI protocol.

Because both runtimes implement the interface Kubernetes expects, cluster operators can configure kubelets to use either one without changing application manifests or workload definitions. Kubernetes documentation identifies containerd and CRI-O among the CRI-compatible runtimes available for nodes. See the Kubernetes container runtimes documentation and the CRI-O project site.

QUESTION NO: 24

Various Container Orchestrator Systems (COS)?

A. Apache Mesos

B. None of the options

C. Docker Swarm

D. Kubernetes

ANSWER: A C D

Explanation:

Apache Mesos, Docker Swarm, and Kubernetes are container orchestration systems because each provides mechanisms to deploy and manage containerized workloads across a group of machines rather than treating containers as isolated, manually operated processes. Apache Mesos is a distributed-systems kernel that can pool compute resources and schedule workloads, including container-based workloads, across a cluster. Docker Swarm provides Docker's native clustering and orchestration capabilities, allowing services to be deployed with desired replica counts and managed across swarm nodes. Kubernetes is an open-source container orchestration platform that automates deployment, scaling, service discovery, workload scheduling, and management of containerized applications. These systems represent different approaches and feature sets, but all address the core orchestration concerns of cluster resource use, workload placement, availability, and lifecycle management. Kubernetes documentation describes Kubernetes as a platform for managing containerized workloads and services that supports declarative configuration and automation. The Cloud Native Computing Foundation's landscape also identifies Kubernetes as a foundational orchestration technology in cloud-native environments. See the Kubernetes Overview and the Apache Mesos documentation.

QUESTION NO: 25

In a cloud native environment, who is usually responsible for maintaining the workloads running across the different platforms?

A. The cloud provider.

B. The Site Reliability Engineering (SRE) team.

C. The team of developers.

D. The Support Engineering team (SE).

ANSWER: B

Explanation:

The Site Reliability Engineering (SRE) team is usually responsible for maintaining workloads across cloud-native platforms because SRE applies software-engineering practices to operating production services reliably at scale. This includes establishing and monitoring service-level objectives, managing incident response, automating repetitive operational activities, planning capacity, and improving the reliability and performance of services as they run in distributed environments. Cloud-native workloads commonly span orchestrators, clusters, cloud services, networking components, and observability systems, so maintaining them requires an operational view that crosses individual application teams and underlying platforms. SRE teams use automation and measurable reliability targets to make this work repeatable and sustainable, rather than relying on manual intervention. This operational ownership also creates a feedback loop: reliability data and incidents drive engineering improvements to deployment processes, resilience, scaling, and service design. Google's foundational SRE guidance describes SRE as treating operations as a software problem and focusing on availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity planning. The CNCF's cloud-native definition likewise emphasizes automated management and operation of scalable applications across dynamic environments. See Google's Site Reliability Engineering book and the CNCF Cloud Native Definition.

QUESTION NO: 26

Which of the following are common telemetry signals used for observability?

A. Metrics

B. Logs

C. Traces

D. Container images

ANSWER: A B C

Explanation:

Metrics, **logs**, and **traces** are the primary telemetry signals used in observability. Metrics provide numerical measurements over time, such as request rate, error rate, and CPU usage. Logs provide timestamped records of events produced by applications and infrastructure. Traces follow an individual request as it moves through distributed services, helping teams understand latency and dependencies.

A container image is a packaged application artifact, not a telemetry signal. Together, metrics, logs, and traces help operators understand the behavior and health of cloud-native systems. See the OpenTelemetry signals documentation.

QUESTION NO: 27

Which Kubernetes probe determines whether a running container should be restarted?

A. Readiness probe

B. Liveness probe

C. Startup probe

D. Resource probe

ANSWER: B

Explanation:

Liveness probe is correct because Kubernetes uses it to determine whether a container is still functioning. When a liveness probe repeatedly fails, the kubelet restarts the container according to the Pod restart policy. This is useful when an application is running but has entered a failed state from which it cannot recover on its own. A readiness probe has a different purpose: it determines whether a container is ready to receive traffic. See the Kubernetes probes documentation.

QUESTION NO: 28

How to get the logs of the previously terminated nginx container from the web pod?

A. kubectl logs -p -c nginx web

B. kubectl logs nginx

C. kubectl logs -p -c web nginx

D. kubectl logs -f -c nginx web

ANSWER: A

Explanation:

`kubectl logs -p -c nginx web` is the correct command because it identifies the Pod as `web`, selects its `nginx` container with `-c nginx`, and requests logs from that container's prior terminated instance with `-p`. The short `-p` flag is equivalent to `--previous`. This is specifically useful when a container has restarted and the administrator needs output from the earlier instance rather than the currently running container.

Kubernetes maintains container logs on the node, and `kubectl logs` retrieves the log stream associated with the requested Pod and container. When a container has restarted, the default command targets the current instance; adding `--previous` changes the target to the immediately preceding terminated container. Specifying the container name is important for Pods that contain more than one container, since it ensures the logs belong to `nginx`. The Kubernetes task documentation demonstrates this pattern using `kubectl logs <pod-name> --previous`, while the command reference documents `-p, --previous` as printing logs from the previous instance of the container. See the kubectl logs reference and the Kubernetes debugging running Pods guide.

QUESTION NO: 29

Which characteristics are associated with cloud-native applications?

A. Declarative APIs

B. Automation

C. Containers

D. Manual recovery as the primary operations model

E. Running all application components on one server

ANSWER: A B C

Explanation:

Declarative APIs are associated with cloud-native systems because desired state can be described and continuously reconciled by automation. **Automation** supports consistent deployment, scaling, recovery, and operations in dynamic environments. **Containers** provide a portable packaging format for application workloads and their dependencies. Cloud-native architecture does not require every application to run on a single server or prohibit the use of managed services. The CNCF describes cloud native technologies as enabling organizations to build and run scalable applications in modern, dynamic environments using approaches that include containers, declarative APIs, and automation. See the CNCF Cloud Native Definition.

QUESTION NO: 30

How can persistent volume be provisioned?

A. Automatically

B. Bootstrap

C. Dynamically

ANSWER: C

Explanation:

Dynamically is correct because Kubernetes supports dynamic provisioning of PersistentVolumes in response to PersistentVolumeClaims (PVCs). An administrator defines one or more StorageClasses that describe available storage and identify the provisioner responsible for creating the backing storage. When a user creates a PVC that requests storage from a particular StorageClass, Kubernetes uses that provisioner to create a matching PersistentVolume automatically and binds it to the claim. This avoids requiring an administrator to create every PersistentVolume in advance and lets workloads request storage through a standard Kubernetes API abstraction.

Dynamic provisioning is especially useful in cloud-native environments because the underlying provisioner can allocate storage from the relevant platform or storage system, while Kubernetes tracks the resulting PersistentVolume and its lifecycle. A StorageClass can also specify parameters such as the reclaim policy, volume expansion behavior, binding mode, and provisioner-specific settings. Kubernetes documentation describes dynamic volume provisioning as the mechanism that creates storage volumes on demand when users create PersistentVolumeClaims, using a StorageClass and provisioner. See the Kubernetes Dynamic Volume Provisioning documentation and the Persistent Volumes documentation.

QUESTION NO: 31

Which statements about PersistentVolumes and PersistentVolumeClaims are correct?

A. A PersistentVolume is a cluster resource

B. A PersistentVolumeClaim requests storage

C. A Pod can mount storage through a PersistentVolumeClaim

D. A ConfigMap requests persistent storage

ANSWER: A B C

Explanation:

A PersistentVolume is a cluster resource that represents storage made available to the cluster. **A PersistentVolumeClaim requests storage** for use by a workload, including attributes such as requested capacity and access modes. Kubernetes can bind a claim to a suitable PersistentVolume, allowing a Pod to consume storage without needing to know the underlying storage implementation.

A Pod can mount storage through a PersistentVolumeClaim by referencing the claim in its volume specification. A ConfigMap stores configuration data and is not a persistent storage request. See the Kubernetes Persistent Volumes documentation.

QUESTION NO: 32

What are benefits of continuous integration?

A. Automated builds and tests

B. Early detection of integration problems

C. Rapid feedback on changes

D. Eliminating the need for testing

E. Guaranteeing production availability

ANSWER: A B C

Explanation:

Automated builds and tests are a benefit of continuous integration because each proposed change can be built and validated consistently. **Early detection of integration problems** is also a benefit because developers integrate changes frequently, reducing the size and complexity of conflicts or failures. **Rapid feedback** helps teams identify failures shortly after a change is submitted. Continuous integration does not eliminate the need for testing or guarantee that an application is ready for production; instead, it establishes frequent integration and automated validation as part of the delivery process. See the CNCF overview of cloud-native CI/CD.

QUESTION NO: 33

Which Kubernetes-native deployment strategy supports zero-downtime updates of a workload?

A. Canary

B. Recreate

C. BlueGreen

D. RollingUpdate

ANSWER: D

Explanation:

RollingUpdate is the Kubernetes Deployment strategy designed to update an application incrementally rather than replacing every Pod simultaneously. During a rollout, Kubernetes creates Pods for the new revision and removes Pods from the prior revision in controlled proportions. The `maxSurge` and `maxUnavailable` settings control how much temporary capacity may be added and how many Pods may be unavailable during this process. With appropriate values, sufficient replicas, and a correctly configured readiness probe, newly created Pods do not receive Service traffic until they are ready. Kubernetes can therefore retain healthy, available endpoints while it transitions the workload to the new version, supporting a zero-downtime rollout.

This behavior is built into the `Deployment` API: the default deployment strategy is `RollingUpdate`. The Kubernetes documentation describes rolling updates as gradually scaling down old ReplicaSets while scaling up new ReplicaSets, and it documents how readiness and availability affect Deployment progress. In practical production use, zero downtime also depends on application-level graceful shutdown and enough capacity to meet traffic needs throughout the transition. See the Kubernetes Deployment documentation and the documentation on maintaining application availability.

QUESTION NO: 34

How to load and generate data required before the Pod startup?

A. Use an init container with shared file storage.

B. Use a PVC volume.

C. Use a sidecar container with shared volume.

D. Use another Pod with a PVC.

ANSWER: A

Explanation:

Use an init container with shared file storage. Init containers are purpose-built for initialization tasks that must finish before a Pod's application containers begin running. Kubernetes runs the Pod's init containers in their declared order and starts the regular application containers only after every init container has completed successfully. This lifecycle guarantee makes an init container the appropriate mechanism for loading input, downloading artifacts, rendering configuration, generating files, or performing other prerequisite preparation.

For file-based data, the standard pattern is to mount the same shared volume into both the init container and the application container. The init container writes or generates the required content in that volume, such as an `emptyDir` volume, and then exits successfully. Kubernetes subsequently starts the application container, which can safely consume the prepared data from the shared mount. This keeps setup logic separate from the main application image and makes startup dependencies explicit in the Pod specification. Kubernetes documents that init containers run and complete before app containers are started, and that volumes can be shared between them. See the Kubernetes Init Containers documentation and the Kubernetes Volumes documentation.