LinuxFoundation.KCNA.v2026-06-06.q96

Exam Code:	KCNA
Exam Name:	Kubernetes and Cloud Native Associate
Certification Provider:	Linux Foundation
Free Question Number:	96
Version:	v2026-06-06
# of views:	102
# of Questions views:	1022
https://www.freecram.net/torrent/LinuxFoundation.KCNA.v2026-06-06.q96.html	

NEW QUESTION: 1
What Kubernetes control plane component exposes the programmatic interface used to create, manage and interact with the Kubernetes objects?
A. kube-controller-manager
B. kube-proxy
C. kube-apiserver
D. etcd
Answer: (SHOW ANSWER)
The kube-apiserver is the front door of the Kubernetes control plane and exposes the programmatic interface used to create, read, update, delete, and watch Kubernetes objects-so C is correct. Every interaction with cluster state ultimately goes through the Kubernetes API. Tools like kubectl, client libraries, GitOps controllers, operators, and core control plane components (scheduler and controllers) all communicate with the API server to submit desired state and to observe current state.
The API server is responsible for handling authentication (who are you?), authorization (what are you allowed to do?), and admission control (should this request be allowed and possibly mutated/validated?). After a request passes these gates, the API server persists the object's desired state to etcd (the backing datastore) and returns a response. The API server also provides a watch mechanism so controllers can react to changes efficiently, enabling Kubernetes' reconciliation model.
It's important to distinguish this from the other options. etcd stores cluster data but does not expose the cluster's primary user-facing API; it's an internal datastore. kube-controller-manager runs control loops (controllers) that continuously reconcile resources (like Deployments, Nodes, Jobs) but it consumes the API rather than exposing it. kube-proxy is a node-level component implementing Service networking rules and is unrelated to the control-plane API endpoint.
Because Kubernetes is "API-driven," the kube-apiserver is central: if it is unavailable, you cannot create workloads, update configurations, or even reliably observe cluster state. This is why high

availability architectures prioritize multiple API server instances behind a load balancer, and why securing the API server (RBAC, TLS, audit) is a primary operational concern.

NEW QUESTION: 2
What is the default eviction timeout when the Ready condition of a node is Unknown or False?
A. Thirty seconds.
B. Thirty minutes.
C. One minute.
D. Five minutes.
Answer: D (LEAVE A REPLY)
The verified correct answer is D (Five minutes). In Kubernetes, node health is continuously monitored. When a node stops reporting status (heartbeats from the kubelet) or is otherwise considered unreachable, the Node controller updates the Node's Ready condition to Unknown (or it can become False). From that point, Kubernetes has to balance two risks: acting too quickly might cause unnecessary disruption (e.g., transient network hiccups), but acting too slowly prolongs outage for workloads that were running on the failed node.
The "default eviction timeout" refers to the control plane behavior that determines how long Kubernetes waits before evicting Pods from a node that appears unhealthy/unreachable. After this timeout elapses, Kubernetes begins eviction of Pods so controllers (like Deployments) can recreate them on healthy nodes, restoring the desired replica count and availability.
This is tightly connected to high availability and self-healing: Kubernetes does not "move" Pods from a dead node; it replaces them. The eviction timeout gives the cluster time to confirm the node is truly unavailable, avoiding flapping in unstable networks. Once eviction begins, replacement Pods can be scheduled elsewhere (assuming capacity exists), which is the normal recovery path for stateless workloads.
It's also worth noting that graceful operational handling can be influenced by PodDisruptionBudgets (for voluntary disruptions) and by workload design (replicas across nodes/zones). But the question is testing the default timer value, which is five minutes in this context.
Therefore, among the choices provided, the correct answer is D.

NEW QUESTION: 3
Which of the following is the correct command to run an nginx deployment with 2 replicas?
A. kubectl run deploy nginx --image=nginx --replicas=2
B. kubectl create deploy nginx --image=nginx --replicas=2
C. kubectl create nginx deployment --image=nginx --replicas=2
D. kubectl create deploy nginx --image=nginx --count=2
Answer: (SHOW ANSWER)
The correct answer is B: kubectl create deploy nginx --image=nginx --replicas=2. This uses kubectl create deployment (shorthand create deploy) to generate a Deployment resource named

nginx with the specified container image. The --replicas=2 flag sets the desired replica count, so Kubernetes will create two Pod replicas (via a ReplicaSet) and keep that number stable.
Option A is incorrect because kubectl run is primarily intended to run a Pod (and in older versions could generate other resources, but it's not the recommended/consistent way to create a Deployment in modern kubectl usage). Option C is invalid syntax: kubectl subcommand order is incorrect; you don't say kubectl create nginx deployment. Option D uses a non-existent --count flag for Deployment replicas.
From a Kubernetes fundamentals perspective, this question tests two ideas: (1) Deployments are the standard controller for running stateless workloads with a desired number of replicas, and (2) kubectl create deployment is a common imperative shortcut for generating that resource. After running the command, you can confirm with kubectl get deploy nginx, kubectl get rs, and kubectl get pods -l app=nginx (label may vary depending on kubectl version). You'll see a ReplicaSet created and two Pods brought up.
In production, teams typically use declarative manifests (kubectl apply -f) or GitOps, but knowing the imperative command is useful for quick labs and validation. The key is that replicas are managed by the controller, not by manually starting containers-Kubernetes reconciles the state continuously.
Therefore, B is the verified correct command.

NEW QUESTION: 4
What's the most adopted way of conflict resolution and decision-making for the open-source projects under the CNCF umbrella?
A. Financial Analysis
B. Discussion and Voting
C. Flipism Technique
D. Project Founder Say
Answer: (SHOW ANSWER)
B (Discussion and Voting) is correct. CNCF-hosted open-source projects generally operate with open governance practices that emphasize transparency, community participation, and documented decision-making. While each project can have its own governance model (maintainers, technical steering committees, SIGs, TOC interactions, etc.), a very common and widely adopted approach to resolving disagreements and making decisions is to first pursue discussion (often on GitHub issues/PRs, mailing lists, or community meetings) and then use voting/consensus mechanisms when needed.
This approach is important because open-source communities are made up of diverse contributors across companies and geographies. "Project Founder Say" (D) is not a sustainable or typical CNCF governance norm for mature projects; CNCF explicitly encourages neutral, community-led governance rather than single-person control. "Financial Analysis" (A) is not a conflict resolution mechanism for technical decisions, and "Flipism Technique" (C) is not a real governance practice.

In Kubernetes specifically, community decisions are often made within structured groups (e.g., SIGs) using discussion and consensus-building, sometimes followed by formal votes where governance requires it. The goal is to ensure decisions are fair, recorded, and aligned with the project's mission and contributor expectations. This also reduces risk of vendor capture and builds trust: anyone can review the rationale in meeting notes, issues, or PR threads, and decisions can be revisited with new evidence.
Therefore, the most adopted conflict resolution and decision-making method across CNCF open-source projects is discussion and voting, making B the verified correct answer.

NEW QUESTION: 5
What edge and service proxy tool is designed to be integrated with cloud native applications?
A. CoreDNS
B. CNI
C. gRPC
D. Envoy
Answer: D (LEAVE A REPLY)
The correct answer is D: Envoy. Envoy is a high-performance edge and service proxy designed for cloud-native environments. It is commonly used as the data plane in service meshes and modern API gateways because it provides consistent traffic management, observability, and security features across microservices without requiring every application to implement those capabilities directly.
Envoy operates at Layer 7 (application-aware) and supports protocols like HTTP/1.1, HTTP/2, gRPC, and more. It can handle routing, load balancing, retries, timeouts, circuit breaking, rate limiting, TLS termination, and mutual TLS (mTLS). Envoy also emits rich telemetry (metrics, access logs, tracing) that integrates well with cloud-native observability stacks.
Why the other options are incorrect:
CoreDNS (A) provides DNS-based service discovery within Kubernetes; it is not an edge/service proxy.
CNI (B) is a specification and plugin ecosystem for container networking (Pod networking), not a proxy.
gRPC (C) is an RPC protocol/framework used by applications; it's not a proxy tool. (Envoy can proxy gRPC traffic, but gRPC itself isn't the proxy.) In Kubernetes architectures, Envoy often appears in two places: (1) at the edge as part of an ingress/gateway layer, and (2) sidecar proxies alongside Pods in a service mesh (like Istio) to standardize service-to-service communication controls and telemetry. This is why it is described as "designed to be integrated with cloud native applications": it's purpose-built for dynamic service discovery, resilient routing, and operational visibility in distributed systems.
So the verified correct choice is D (Envoy).

NEW QUESTION: 6
Which are the core features provided by a service mesh?

A. Authentication and authorization
B. Distributing and replicating data
C. Security vulnerability scanning
D. Configuration management
Answer: (SHOW ANSWER)
A is the correct answer because a service mesh primarily focuses on securing and managing service-to-service communication, and a core part of that is authentication and authorization. In microservices architectures, internal ("east-west") traffic can become a complex web of calls. A service mesh introduces a dedicated communication layer-commonly implemented with sidecar proxies or node proxies plus a control plane-to apply consistent security and traffic policies across services.
Authentication in a mesh typically means service identity: each workload gets an identity (often via certificates), enabling mutual TLS (mTLS) so services can verify each other and encrypt traffic in transit. Authorization then builds on identity to enforce "who can talk to whom" via policies (for example: service A can call service B only on certain paths or methods). These capabilities are central because they reduce the need for every development team to implement and maintain custom security libraries correctly.
Why the other answers are incorrect:
B (data distribution/replication) is a storage/database concern, not a mesh function.
C (vulnerability scanning) is typically part of CI/CD and supply-chain security tooling, not service-to-service runtime traffic management.
D (configuration management) is broader (GitOps, IaC, Helm/Kustomize); a mesh does have configuration, but "configuration management" is not the defining core feature tested here.
Service meshes also commonly provide traffic management (timeouts, retries, circuit breaking, canary routing) and telemetry (metrics/traces), but among the listed options, authentication and authorization best matches "core features." It captures the mesh's role in standardizing secure communications in a distributed system.
So, the verified correct answer is A.

NEW QUESTION: 7
Why is Cloud-Native Architecture important?
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: (SHOW ANSWER)
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.

NEW QUESTION: 8
Which two elements are shared between containers in the same pod?
A. Network resources and liveness probes.
B. Storage and container image registry.
C. Storage and network resources.
D. Network resources and Dockerfiles.
Answer: (SHOW ANSWER)
The correct answer is C: Storage and network resources. In Kubernetes, a Pod is the smallest schedulable unit and acts like a "logical host" for its containers. Containers inside the same Pod share a number of namespaces and resources, most notably:
Network: all containers in a Pod share the same network namespace, which means they share a single Pod IP address and the same port space. They can talk to each other via localhost and coordinate tightly without exposing separate network endpoints.
Storage: containers in a Pod can share data through Pod volumes. Volumes (like emptyDir, ConfigMap/Secret volumes, or PVC-backed volumes) are defined at the Pod level and can be mounted into multiple containers within the Pod. This enables common patterns like a sidecar writing logs to a shared volume that the main container generates, or an init/sidecar container producing configuration or certificates for the main container.
Why other options are wrong: liveness probes (A) are defined per container (or per Pod template) but are not a "shared" resource between containers. A container image registry (B) is an external system and not a shared in-Pod element. Dockerfiles (D) are build-time artifacts, irrelevant at runtime, and not shared resources.

This question is a classic test of Pod fundamentals: multi-container Pods work precisely because they share networking and volumes. This is also why the sidecar pattern is feasible-sidecars can intercept traffic on localhost, export metrics, or ship logs while sharing the same lifecycle boundary and scheduling placement.
Therefore, the verified correct choice is C.

NEW QUESTION: 9
What is the reference implementation of the OCI runtime specification?
A. lxc
B. CRI-O
C. runc
D. Docker
Answer: (SHOW ANSWER)
The verified correct answer is C (runc). The Open Container Initiative (OCI) defines standards for container image format and runtime behavior. The OCI runtime specification describes how to run a container (process execution, namespaces, cgroups, filesystem mounts, capabilities, etc.). runc is widely recognized as the reference implementation of that runtime spec and is used underneath many higher-level container runtimes.
In common container stacks, Kubernetes nodes typically run a CRI-compliant runtime such as containerd or CRI-O. Those runtimes handle image management, container lifecycle coordination, and CRI integration, but they usually invoke an OCI runtime to actually create and start containers. In many deployments, that OCI runtime is runc (or a compatible alternative). This layering helps keep responsibilities separated: CRI runtime manages orchestration-facing operations; OCI runtime performs the low-level container creation according to the standardized spec.
Option A (lxc) is an older Linux containers technology and tooling ecosystem, but it is not the OCI runtime reference implementation. Option B (CRI-O) is a Kubernetes-focused container runtime that implements CRI; it uses OCI runtimes (often runc) underneath, so it's not the reference implementation itself. Option D (Docker) is a broader platform/tooling suite; while Docker historically used runc under the hood and helped popularize containers, the OCI reference runtime implementation is runc, not Docker.
Understanding this matters in container orchestration contexts because it clarifies what Kubernetes depends on: Kubernetes relies on CRI for runtime integration, and runtimes rely on OCI standards for interoperability. OCI standards ensure that images and runtime behavior are portable across tools and vendors, and runc is the canonical implementation that demonstrates those standards in practice.
Therefore, the correct answer is C: runc.

NEW QUESTION: 10
At which layer would distributed tracing be implemented in a cloud native deployment?
A. Network

B. Application
C. Database
D. Infrastructure
Answer: (SHOW ANSWER)
Distributed tracing is implemented primarily at the application layer, so B is correct. The reason is simple: tracing is about capturing the end-to-end path of a request as it traverses services, libraries, queues, and databases. That "request context" (trace ID, span ID, baggage) must be created, propagated, and enriched as code executes. While infrastructure components (proxies, gateways, service meshes) can generate or augment trace spans, the fundamental unit of tracing is still tied to application operations (an HTTP handler, a gRPC call, a database query, a cache lookup).
In Kubernetes-based microservices, distributed tracing typically uses standards like OpenTelemetry for instrumentation and context propagation. Application frameworks emit spans for key operations, attach attributes (route, status code, tenant, retry count), and propagate context via headers (e.g., W3C Trace Context). This is what lets you reconstruct "Service A → Service B → Service C" for one user request and identify the slow or failing hop.
Why other layers are not the best answer:
Network focuses on packets/flows, but tracing is not a packet-capture problem; it's a causal request-path problem across services.
Database spans are part of traces, but tracing is not "implemented in the database layer" overall; DB spans are one component.
Infrastructure provides the platform and can observe traffic, but without application context it can't fully represent business operations (and many useful attributes live in app code).
So the correct layer for "where tracing is implemented" is the application layer-even when a mesh or proxy helps, it's still describing application request execution across components.

NEW QUESTION: 11
What component enables end users, different parts of the Kubernetes cluster, and external components to communicate with one another?
A. kubectl
B. AWS Management Console
C. Kubernetes API
D. Google Cloud SDK
Answer: (SHOW ANSWER)
The Kubernetes API is the central interface that enables communication between users, controllers, nodes, and external integrations, so C is correct. Kubernetes is fundamentally an API-driven system: all cluster state is represented as API objects, and all operations-create, update, delete, watch-flow through the API server.
End users typically interact with the Kubernetes API using tools like kubectl, client libraries, or dashboards. But those tools are clients; the shared communication "hub" is the API itself. Inside the cluster, core control plane components (controllers, scheduler) continuously watch the API for

desired state and write status updates back. Worker nodes (via kubelet) also communicate with the API server to receive Pod specs, report node health, and update Pod statuses. External systems-cloud provider integrations, CI/CD pipelines, GitOps controllers, monitoring and policy engines-also integrate primarily through the Kubernetes API.
Option A (kubectl) is a CLI that talks to the Kubernetes API; it is not the underlying component that all parts use to communicate. Options B and D are cloud-provider tools and are not universal to Kubernetes clusters. Kubernetes runs across many environments, and the consistent interoperability layer is the Kubernetes API.
This API-centric architecture is what enables Kubernetes' declarative model: you submit desired state to the API, and controllers reconcile actual state to match. It also enables extensibility: CRDs and admission webhooks expand what the API can represent and enforce. Therefore, the correct answer is C: Kubernetes API.

NEW QUESTION: 12
Which of the following are tasks performed by a container orchestration tool?
A. Schedule, scale, and manage the health of containers.
B. Create images, scale, and manage the health of containers.
C. Debug applications, and manage the health of containers.
D. Store images, scale, and manage the health of containers.
Answer: A (LEAVE A REPLY)
A container orchestration tool (like Kubernetes) is responsible for scheduling, scaling, and health management of workloads, making A correct. Orchestration sits above individual containers and focuses on running applications reliably across a fleet of machines. Scheduling means deciding which node should run a workload based on resource requests, constraints, affinities, taints/tolerations, and current cluster state. Scaling means changing the number of running instances (replicas) to meet demand (manually or automatically through autoscalers). Health management includes monitoring whether containers and Pods are alive and ready, replacing failed instances, and maintaining the declared desired state.
Options B and D include "create images" and "store images," which are not orchestration responsibilities. Image creation is a CI/build responsibility (Docker/BuildKit/build systems), and image storage is a container registry responsibility (Harbor, ECR, GCR, Docker Hub, etc.). Kubernetes consumes images from registries but does not build or store them. Option C includes "debug applications," which is not a core orchestration function. While Kubernetes provides tools that help debugging (logs, exec, events), debugging is a human/operator activity rather than the orchestrator's fundamental responsibility.
In Kubernetes specifically, these orchestration tasks are implemented through controllers and control loops: Deployments/ReplicaSets manage replica counts and rollouts, kube-scheduler assigns Pods to nodes, kubelet ensures containers run, and probes plus controller logic replace unhealthy replicas. This is exactly what makes Kubernetes valuable at scale: instead of manually starting/stopping containers on individual hosts, you declare your intent and let the orchestration

system continually reconcile reality to match. That combination-placement + elasticity + self-healing-is the core of container orchestration, matching option A precisely.

NEW QUESTION: 13
Let's assume that an organization needs to process large amounts of data in bursts, on a cloud-based Kubernetes cluster. For instance: each Monday morning, they need to run a batch of 1000 compute jobs of 1 hour each, and these jobs must be completed by Monday night. What's going to be the most cost-effective method?
A. Run a group of nodes with the exact required size to complete the batch on time, and use a combination of taints, tolerations, and nodeSelectors to reserve these nodes to the batch jobs.
B. Leverage the Kubernetes Cluster Autoscaler to automatically start and stop nodes as they're needed.
C. Commit to a specific level of spending to get discounted prices (with e.g. "reserved instances" or similar mechanisms).
D. Use PriorityClasses so that the weekly batch job gets priority over other workloads running on the cluster, and can be completed on time.
Answer: B (LEAVE A REPLY)
Burst workloads are a classic elasticity problem: you need large capacity for a short window, then very little capacity the rest of the week. The most cost-effective approach in a cloud-based Kubernetes environment is to scale infrastructure dynamically, matching node count to current demand. That's exactly what Cluster Autoscaler is designed for: it adds nodes when Pods cannot be scheduled due to insufficient resources and removes nodes when they become underutilized and can be drained safely. Therefore B is correct.
Option A can work operationally, but it commonly results in paying for a reserved "standing army" of nodes that sit idle most of the week-wasteful for bursty patterns unless the nodes are repurposed for other workloads. Taints/tolerations and nodeSelectors are placement tools; they don't reduce cost by themselves and may increase waste if they isolate nodes. Option D (PriorityClasses) affects which Pods get scheduled first given available capacity, but it does not create capacity. If the cluster doesn't have enough nodes, high priority Pods will still remain Pending. Option C (reserved instances or committed-use discounts) can reduce unit price, but it assumes relatively predictable baseline usage. For true bursts, you usually want a smaller baseline plus autoscaling, and optionally combine it with discounted capacity types if your cloud supports them.
In Kubernetes terms, the control loop is: batch Jobs create Pods → scheduler tries to place Pods → if many Pods are Pending due to insufficient CPU/memory, Cluster Autoscaler observes this and increases the node group size → new nodes join and kube-scheduler places Pods → after jobs finish and nodes become empty, Cluster Autoscaler drains and removes nodes. This matches cloud-native principles: elasticity, pay-for-what-you-use, and automation. It minimizes idle capacity while still meeting the completion deadline.

NEW QUESTION: 14

Which mechanism allows extending the Kubernetes API?

A. ConfigMap

B. CustomResourceDefinition

C. MutatingAdmissionWebhook mechanism

D. Kustomize

Answer: (SHOW ANSWER)

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.

NEW QUESTION: 15

What is the telemetry component that represents a series of related distributed events that encode the end-to-end request flow through a distributed system?

A. Metrics

B. Logs

C. Spans

D. Traces

Answer: (SHOW ANSWER)

In observability, traces represent an end-to-end view of a request as it flows through multiple services, so D is correct. Tracing is particularly important in cloud-native microservices architectures because a single user action (like "checkout" or "search") may traverse many services via HTTP/gRPC calls, message queues, and databases. Traces link those related events together so you can see where time is spent, where errors occur, and how dependencies behave.

A trace is typically composed of multiple spans (option C). A span is a single timed operation (e.g., "HTTP GET /orders", "DB query", "call payment service"). Spans include timing, attributes

(tags), status/error information, and parent/child relationships. While spans are essential building blocks, the "series of related distributed events encoding end-to-end request flow" is the trace as a whole, not an individual span.
Metrics (option A) are numeric time series used for aggregation and alerting (rates, latency percentiles when derived, resource usage). Logs (option B) are discrete event records (text or structured) useful for forensic detail and debugging. Both are valuable, but neither inherently provides a stitched, causal, end-to-end request path across services. Traces do exactly that by propagating trace context (trace IDs/span IDs) across service boundaries (often via headers).
In Kubernetes environments, traces are commonly exported via OpenTelemetry instrumentation/collectors and visualized in tracing backends. Tracing enables faster incident resolution by pinpointing the slow hop, the failing downstream dependency, or unexpected fan-out. Therefore, the correct telemetry component for end-to-end distributed request flow is Traces (D).

NEW QUESTION: 16
Which of the following best describes horizontally scaling an application deployment?
A. The act of adding/removing node instances to the cluster to meet demand.
B. The act of adding/removing applications to meet demand.
C. The act of adding/removing application instances of the same application to meet demand.
D. The act of adding/removing resources to application instances to meet demand.
Answer: (SHOW ANSWER)
Horizontal scaling means changing how many instances of an application are running, not changing how big each instance is. Therefore, the best description is C: adding/removing application instances of the same application to meet demand. In Kubernetes, "instances" typically correspond to Pod replicas managed by a controller like a Deployment. When you scale horizontally, you increase or decrease the replica count, which increases or decreases total throughput and resilience by distributing load across more Pods.
Option A is about cluster/node scaling (adding or removing nodes), which is infrastructure scaling typically handled by a cluster autoscaler in cloud environments. Node scaling can enable more Pods to be scheduled, but it's not the definition of horizontal application scaling itself. Option D describes vertical scaling-adding/removing CPU or memory resources to a given instance (Pod/container) by changing requests/limits or using VPA. Option B is vague and not the standard definition.
Horizontal scaling is a core cloud-native pattern because it improves availability and elasticity. If one Pod fails, other replicas continue serving traffic. In Kubernetes, scaling can be manual (kubectl scale deployment ... --replicas=N) or automatic using the Horizontal Pod Autoscaler (HPA). HPA adjusts replicas based on observed metrics like CPU utilization, memory, or custom/external metrics (for example, request rate or queue length). This creates responsive systems that can handle variable traffic.
From an architecture perspective, designing for horizontal scaling often means ensuring your application is stateless (or manages state externally), uses idempotent request handling, and

supports multiple concurrent instances. Stateful workloads can also scale horizontally, but usually with additional constraints (StatefulSets, sharding, quorum membership, stable identity).
So the verified definition and correct choice is C.

NEW QUESTION: 17
Which GitOps engine can be used to orchestrate parallel jobs on Kubernetes?
A. Jenkins X
B. Flagger
C. Flux
D. Argo Workflows
Answer: (SHOW ANSWER)
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).

NEW QUESTION: 18
Which of the following statements is correct concerning Open Policy Agent (OPA)?
A. The policies must be written in Python language.
B. Kubernetes can use it to validate requests and apply policies.
C. Policies can only be tested when published.
D. It cannot be used outside Kubernetes.
Answer: (SHOW ANSWER)
Open Policy Agent (OPA) is a general-purpose policy engine used to define and enforce policy across different systems. In Kubernetes, OPA is commonly integrated through admission control (often via Gatekeeper or custom admission webhooks) to validate and/or mutate requests before they are persisted in the cluster. This makes B correct: Kubernetes can use OPA to validate API requests and apply policy decisions.
Kubernetes' admission chain is where policy enforcement naturally fits. When a user or controller submits a request (for example, to create a Pod), the API server can call external admission webhooks. Those webhooks can evaluate the request against policy-such as "no privileged containers," "images must come from approved registries," "labels must include cost-center," or "Ingress must enforce TLS." OPA's policy language (Rego) allows expressing these rules in a declarative form, and the decision ("allow/deny" and sometimes patches) is returned to the API server. This enforces governance consistently and centrally.
Option A is incorrect because OPA policies are written in Rego, not Python. Option C is incorrect because policies can be tested locally and in CI pipelines before deployment; in fact, testability is a key advantage. Option D is incorrect because OPA is designed to be platform-agnostic-it can be used with APIs, microservices, CI/CD pipelines, service meshes, and infrastructure tools, not only Kubernetes.
From a Kubernetes fundamentals view, OPA complements RBAC: RBAC answers "who can do what to which resources," while OPA-style admission policies answer "even if you can create this resource, does it meet our organizational rules?" Together they help implement defense in depth: authentication + authorization + policy admission + runtime security controls. That is why OPA is widely used to enforce security and compliance requirements in Kubernetes environments.

NEW QUESTION: 19
Which of these components is part of the Kubernetes Control Plane?
A. CoreDNS
B. cloud-controller-manager
C. kube-proxy
D. kubelet
Answer: (SHOW ANSWER)
The Kubernetes control plane is the set of components responsible for making cluster-wide decisions (like scheduling) and detecting and responding to cluster events (like starting new Pods when they fail). In upstream Kubernetes architecture, the canonical control plane components

include kube-apiserver, etcd, kube-scheduler, and kube-controller-manager, and-when running on a cloud provider-the cloud-controller-manager. That makes option B the correct answer: cloud-controller-manager is explicitly a control plane component that integrates Kubernetes with the underlying cloud.
The cloud-controller-manager runs controllers that talk to cloud APIs for infrastructure concerns such as node lifecycle, routes, and load balancers. For example, when you create a Service of type LoadBalancer, a controller in this component is responsible for provisioning a cloud load balancer and updating the Service status. This is clearly control-plane behavior: reconciling desired state into real infrastructure state.
Why the others are not control plane components (in the classic classification): kubelet is a node component (agent) responsible for running and managing Pods on a specific node. kube-proxy is also a node component that implements Service networking rules on nodes. CoreDNS is usually deployed as a cluster add-on for DNS-based service discovery; it's critical, but it's not a control plane component in the strict architectural list.
So, while many clusters run CoreDNS in kube-system, the Kubernetes component that is definitively "part of the control plane" among these choices is cloud-controller-manager (B).

NEW QUESTION: 20

How long should a stable API element in Kubernetes be supported (at minimum) after deprecation?

A. 9 months
B. 24 months
C. 12 months
D. 6 months
Answer: (SHOW ANSWER)
Kubernetes has a formal API deprecation policy to balance stability for users with the ability to evolve the platform. For a stable (GA) API element, Kubernetes commits to supporting that API for a minimum period after it is deprecated. The correct minimum in this question is 12 months, which corresponds to option C.
In practice, Kubernetes releases occur roughly every three to four months, and the deprecation policy is commonly communicated in terms of "releases" as well as time. A GA API that is deprecated in one release is typically kept available for multiple subsequent releases, giving cluster operators and application teams time to migrate manifests, client libraries, controllers, and automation. This matters because Kubernetes is often at the center of production delivery pipelines; abrupt API removals would break deployments, upgrades, and tooling. By guaranteeing a minimum support window, Kubernetes enables predictable upgrades and safer lifecycle management.
This policy also encourages teams to track API versions and plan migrations. For example, workloads might start on a beta API (which can change), but once an API reaches stable, users can expect a stronger compatibility promise. Deprecation warnings help surface risk early. In many clusters, you'll see API server warnings and tooling hints when manifests use deprecated fields/versions, allowing proactive remediation before the removal release.

Options 6 or 9 months would be too short for many enterprises to coordinate changes across multiple teams and environments. 24 months may be true for some ecosystems, but the Kubernetes stated minimum in this exam-style framing is 12 months. The key operational takeaway is: don't ignore deprecation notices-they're your clock for migration planning. Treat API version upgrades as part of routine cluster lifecycle hygiene to avoid being blocked during Kubernetes version upgrades when deprecated APIs are finally removed.

NEW QUESTION: 21
What is the name of the lightweight Kubernetes distribution built for IoT and edge computing?
A. OpenShift
B. k3s
C. RKE
D. k1s
Answer: (SHOW ANSWER)
Edge and IoT environments often have constraints that differ from traditional datacenters: limited CPU/RAM, intermittent connectivity, smaller footprints, and a desire for simpler operations. k3s is a well-known lightweight Kubernetes distribution designed specifically to run in these environments, making B the correct answer.
What makes k3s "lightweight" is that it packages Kubernetes components in a simplified way and reduces operational overhead. It typically uses a single binary distribution and can run with an embedded datastore option for smaller installations (while also supporting external datastores for HA use cases). It streamlines dependencies and is aimed at faster installation and reduced resource consumption, which is ideal for edge nodes, IoT gateways, small servers, labs, and development environments.
By contrast, OpenShift is a Kubernetes distribution focused on enterprise platform capabilities, with additional security defaults, integrated developer tooling, and a larger operational footprint-excellent for many enterprises but not "built for IoT and edge" as the defining characteristic. RKE (Rancher Kubernetes Engine) is a Kubernetes installer/engine used to deploy Kubernetes, but it's not specifically the lightweight edge-focused distribution in the way k3s is. "k1s" is not a standard, widely recognized Kubernetes distribution name in this context.
From a cloud native architecture perspective, edge Kubernetes distributions extend the same declarative and API-driven model to places where you want consistent operations across cloud, datacenter, and edge. You can apply GitOps patterns, standard manifests, and Kubernetes-native controllers across heterogeneous footprints. k3s provides that familiar Kubernetes experience while optimizing for constrained environments, which is why it has become a common choice for edge/IoT Kubernetes deployments.

NEW QUESTION: 22
What native runtime is Open Container Initiative (OCI) compliant?
A. runC
B. runV

C. kata-containers
D. gvisor
Answer: (SHOW ANSWER)
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.

NEW QUESTION: 23
What feature must a CNI support to control specific traffic flows for workloads running in Kubernetes?
A. Border Gateway Protocol
B. IP Address Management
C. Pod Security Policy
D. Network Policies
Answer: (SHOW ANSWER)
To control which workloads can communicate with which other workloads in Kubernetes, you use NetworkPolicy resources-but enforcement depends on the cluster's networking implementation. Therefore, for traffic-flow control, the CNI/plugin must support Network Policies, making D correct. Kubernetes defines the NetworkPolicy API as a declarative way to specify allowed ingress and egress traffic based on selectors (Pod labels, namespaces, IP blocks) and ports/protocols. However, Kubernetes itself does not enforce NetworkPolicy rules; enforcement is provided by the network plugin (or associated dataplane components). If your CNI does not implement NetworkPolicy, the objects may exist in the API but have no effect-Pods will communicate freely by default.
Option B (IP Address Management) is often part of CNI responsibilities, but IPAM is about assigning addresses, not enforcing L3/L4 security policy. Option A (BGP) is used by some CNIs

to advertise routes (for example, in certain Calico deployments), but BGP is not the general requirement for policy enforcement. Option C (Pod Security Policy) is a deprecated/removed Kubernetes admission feature related to Pod security settings, not network flow control.
From a Kubernetes security standpoint, NetworkPolicies are a key tool for implementing least privilege at the network layer-limiting lateral movement, reducing blast radius, and segmenting environments. But they only work when the chosen CNI supports them. Thus, the correct answer is D: Network Policies.

NEW QUESTION: 24
What is the main purpose of a DaemonSet?
A. A DaemonSet ensures that all (or certain) nodes run a copy of a Pod.
B. A DaemonSet ensures that the kubelet is constantly up and running.
C. A DaemonSet ensures that there are as many pods running as specified in the replicas field.
D. A DaemonSet ensures that a process (agent) runs on every node.
Answer: (SHOW ANSWER)
The correct answer is A. A DaemonSet is a workload controller whose job is to ensure that a specific Pod runs on all nodes (or on a selected subset of nodes) in the cluster. This is fundamentally different from Deployments/ReplicaSets, which aim to maintain a certain replica count regardless of node count. With a DaemonSet, the number of Pods is implicitly tied to the number of eligible nodes: add a node, and the DaemonSet automatically schedules a Pod there; remove a node, and its Pod goes away.
DaemonSets are commonly used for node-level services and background agents: log collectors, node monitoring agents, storage daemons, CNI components, or security agents-anything where you want a presence on each node to interact with node resources. This aligns with option D's phrasing ("agent on every node"), but option A is the canonical definition and is slightly broader because it covers "all or certain nodes" (via node selectors/affinity/taints-tolerations) and the fact that the unit is a Pod.
Why the other options are wrong: DaemonSets do not "keep kubelet running" (B); kubelet is a node service managed by the OS. DaemonSets do not use a replicas field to maintain a specific count (C); that's Deployment/ReplicaSet behavior.
Operationally, DaemonSets matter for cluster operations because they provide consistent node coverage and automatically react to node pool scaling. They also require careful scheduling constraints so they land only where intended (e.g., only Linux nodes, only GPU nodes). But the main purpose remains: ensure a copy of a Pod runs on each relevant node-option A.

NEW QUESTION: 25
Services and Pods in Kubernetes are _____ objects.
A. JSON
B. YAML
C. Java
D. REST

Answer: D (LEAVE A REPLY)
In Kubernetes, resources like Pods and Services are represented as API objects that you create, read, update, delete, and watch via the Kubernetes RESTful API. That makes D (REST) the correct answer.
Kubernetes is fundamentally API-driven: the API server exposes endpoints for each resource type (for example, /api/v1/namespaces/{ns}/pods and /api/v1/namespaces/{ns}/services). Clients such as kubectl, controllers, operators, and external systems interact with these resources by making REST-style calls using HTTP verbs (GET, POST, PUT/PATCH, DELETE) and using watch streams for event-driven updates. This API-first design is what enables Kubernetes' declarative model-users submit desired state to the API server, and controllers reconcile the cluster to that desired state.
Options A and B (JSON and YAML) are common serialization formats used to represent Kubernetes objects, but they are not what the objects "are." Kubernetes objects are logical API resources; they can be encoded as JSON (what the API uses) and often authored as YAML for human convenience. YAML is effectively a superset-friendly format that can be converted to JSON. The underlying API object model remains the same regardless of whether you wrote YAML or JSON. Option C (Java) is unrelated; Java is a programming language that can interact with Kubernetes via client libraries, but Kubernetes objects are not "Java objects" in the platform's definition.
So the accurate statement is: Pods and Services are Kubernetes REST API objects (resources) exposed and managed through the Kubernetes API server, which is why REST is the correct fill-in.

NEW QUESTION: 26
What does CNCF stand for?
A. Cloud Native Community Foundation
B. Cloud Native Computing Foundation
C. Cloud Neutral Computing Foundation
D. Cloud Neutral Community Foundation
Answer: (SHOW ANSWER)
CNCF stands for the Cloud Native Computing Foundation, making B correct. CNCF is the foundation that hosts and sustains many cloud-native open source projects, including Kubernetes, and provides governance, neutral stewardship, and community infrastructure to help projects grow and remain vendor-neutral.
CNCF's scope includes not only Kubernetes but also a broad ecosystem of projects across observability, networking, service meshes, runtime security, CI/CD, and application delivery. The foundation defines processes for project incubation and graduation, promotes best practices, organizes community events, and supports interoperability and adoption through reference architectures and education.
In the Kubernetes context, CNCF's role matters because Kubernetes is a massive multi-vendor project. Neutral governance reduces the risk that any single company can unilaterally control

direction. This fosters broad contribution and adoption across cloud providers and enterprises. CNCF also supports the broader "cloud native" definition, often associated with containerization, microservices, declarative APIs, automation, and resilience principles.
The incorrect options are close-sounding but not accurate expansions. "Cloud Native Community Foundation" and the "Cloud Neutral ..." variants are not the recognized meaning. The correct official name is Cloud Native Computing Foundation.
So, the verified answer is B, and understanding CNCF helps connect Kubernetes to its broader ecosystem of standardized, interoperable cloud-native tooling.

NEW QUESTION: 27
Which item is a Kubernetes node component?
A. kube-scheduler
B. kubectl
C. kube-proxy
D. etcd
Answer: (SHOW ANSWER)
A Kubernetes node component is a component that runs on worker nodes to support Pods and node-level networking/operations. Among the options, kube-proxy is a node component, so C is correct.
kube-proxy runs on each node and implements parts of the Kubernetes Service networking model. It watches the API server for Service and endpoint updates and then programs node networking rules (iptables/IPVS, or equivalent) so traffic sent to a Service IP/port is forwarded to one of the backend Pod endpoints. This is essential for stable virtual IPs and load distribution across Pods.
Why the other options are not node components:
kube-scheduler is a control plane component; it assigns Pods to nodes but does not run on every node as part of node functionality.
kubectl is a client CLI tool used by humans/automation; it is not a cluster component.
etcd is the control plane datastore; it stores cluster state and is not a per-node workload component.
Operationally, kube-proxy can be replaced by some modern CNI/eBPF dataplanes, but in classic Kubernetes architecture it remains the canonical node-level component for Service rule programming. Understanding which components are node vs control plane is key for troubleshooting: node issues involve kubelet/runtime/kube-proxy/CNI; control plane issues involve API server/scheduler/controller-manager/etcd.
So, the verified node component in this list is kube-proxy (C).

NEW QUESTION: 28
What is the resource type used to package sets of containers for scheduling in a cluster?
A. Pod
B. ContainerSet

C. ReplicaSet
D. Deployment
Answer: A (LEAVE A REPLY)
The Kubernetes resource used to package one or more containers into a schedulable unit is the Pod, so A is correct. Kubernetes schedules Pods onto nodes; it does not schedule individual containers. A Pod represents a single "instance" of an application component and includes one or more containers that share key runtime properties, including the same network namespace (same IP and port space) and the ability to share volumes.
Pods enable common patterns beyond "one container per Pod." For example, a Pod may include a main application container plus a sidecar container for logging, proxying, or configuration reload. Because these containers share localhost networking and volume mounts, they can coordinate efficiently without requiring external service calls. Kubernetes manages the Pod lifecycle as a unit: the containers in a Pod are started according to container lifecycle rules and are co-located on the same node.
Option B (ContainerSet) is not a standard Kubernetes workload resource. Option C (ReplicaSet) manages a set of Pod replicas, ensuring a desired count is running, but it is not the packaging unit itself. Option D (Deployment) is a higher-level controller that manages ReplicaSets and provides rollout/rollback behavior, again operating on Pods rather than being the container-packaging unit.
From the scheduling perspective, the PodSpec defines container images, commands, resources, volumes, security context, and placement constraints. The scheduler evaluates these constraints and assigns the Pod to a node. This "Pod as the atomic scheduling unit" is fundamental to Kubernetes architecture and explains why Kubernetes-native concepts (Services, selectors, readiness, autoscaling) all revolve around Pods.

NEW QUESTION: 29
What is the Kubernetes object used for running a recurring workload?
A. Job
B. Batch
C. DaemonSet
D. CronJob
Answer: (SHOW ANSWER)
A recurring workload in Kubernetes is implemented with a CronJob, so the correct choice is D. A CronJob is a controller that creates Jobs on a schedule defined in standard cron format (minute, hour, day of month, month, day of week). This makes CronJobs ideal for periodic tasks like backups, report generation, log rotation, and cleanup tasks.
A Job (option A) is run-to-completion but is typically a one-time execution; it ensures that a specified number of Pods successfully terminate. You can use a Job repeatedly, but something else must create it each time-CronJob is that built-in scheduler. Option B ("Batch") is not a standard workload resource type (batch is an API group, not the object name used here). Option C (DaemonSet) ensures one Pod runs on every node (or selected nodes), which is not

"recurring," it's "always present per node." CronJobs include operational controls that matter in real clusters. For example, concurrencyPolicy controls what happens if a scheduled run overlaps with a previous run (Allow, Forbid, Replace). startingDeadlineSeconds can handle missed schedules (e.g., if the controller was down). History limits (successfulJobsHistoryLimit, failedJobsHistoryLimit) help manage cleanup and troubleshooting. Each scheduled execution results in a Job with its own Pods, which can be inspected with kubectl get jobs and kubectl logs. So the correct Kubernetes object for a recurring workload is CronJob (D): it provides native scheduling and creates Jobs automatically according to the defined cadence.

NEW QUESTION: 30

What is CloudEvents?

A. It is a specification for describing event data in common formats for Kubernetes network traffic management and cloud providers.

B. It is a specification for describing event data in common formats in all cloud providers including major cloud providers.

C. It is a specification for describing event data in common formats to provide interoperability across services, platforms and systems.

D. It is a Kubernetes specification for describing events data in common formats for iCloud services, iOS platforms and iMac.

Answer: C (LEAVE A REPLY)

CloudEvents is an open specification for describing event data in a common way to enable interoperability across services, platforms, and systems, so C is correct. In cloud-native architectures, many components communicate asynchronously via events (message brokers, event buses, webhooks). Without a standard envelope, each producer and consumer invents its own event structure, making integration brittle. CloudEvents addresses this by standardizing core metadata fields-like event id, source, type, specversion, and time-and defining how event payloads are carried.

This helps systems interoperate regardless of transport. CloudEvents can be serialized as JSON or other encodings and carried over HTTP, messaging systems, or other protocols. By using a shared spec, you can route, filter, validate, and transform events more consistently.

Option A is too narrow and incorrectly ties CloudEvents to Kubernetes traffic management; CloudEvents is broader than Kubernetes. Option B is closer but still framed incorrectly-CloudEvents is not merely "for all cloud providers," it is an interoperability spec across services and platforms, including but not limited to cloud provider event systems. Option D is clearly incorrect.

In Kubernetes ecosystems, CloudEvents is relevant to event-driven systems and serverless platforms (e.g., Knative Eventing and other eventing frameworks) because it provides a consistent event contract across producers and consumers. That consistency reduces coupling, supports better tooling (schema validation, tracing correlation), and makes event-driven architectures easier to operate at scale.

So, the correct definition is C: a specification for common event formats to enable interoperability across systems.

NEW QUESTION: 31
How is application data maintained in containers?
A. Store data into data folders.
B. Store data in separate folders.
C. Store data into sidecar containers.
D. Store data into volumes.
Answer: (SHOW ANSWER)
Container filesystems are ephemeral: the writable layer is tied to the container lifecycle and can be lost when containers are recreated. Therefore, maintaining application data correctly means storing it in volumes, making D the correct answer. In Kubernetes, volumes provide durable or shareable storage that is mounted into containers at specific paths. Depending on the volume type, the data can persist across container restarts and even Pod rescheduling.
Kubernetes supports many volume patterns. For transient scratch data you might use emptyDir (ephemeral for the Pod's lifetime). For durable state, you typically use PersistentVolumes consumed by PersistentVolumeClaims (PVCs), backed by storage systems via CSI drivers (cloud disks, SAN/NAS, distributed storage). This decouples the application container image from its state and enables rolling updates, rescheduling, and scaling without losing data.
Options A and B ("folders") are incomplete because folders inside the container filesystem do not guarantee persistence. A folder is only as durable as the underlying storage; without a mounted volume, it lives in the container's writable layer and will disappear when the container is replaced.
Option C is incorrect because "sidecar containers" are not a data durability mechanism; sidecars can help ship logs or sync data, but persistent data should still be stored on volumes (or external services like managed databases).
From an application delivery standpoint, the principle is: containers should be immutable and disposable, and state should be externalized. Volumes (and external managed services) make this possible. In Kubernetes, this is a foundational pattern enabling safe rollouts, self-healing, and portability: the platform can kill and recreate Pods freely because data is maintained independently via volumes.
Therefore, the verified correct choice is D: Store data into volumes.

NEW QUESTION: 32
What are the initial namespaces that Kubernetes starts with?
A. default, kube-system, kube-public, kube-node-lease
B. default, system, kube-public
C. kube-default, kube-system, kube-main, kube-node-lease
D. kube-default, system, kube-main, kube-primary
Answer: (SHOW ANSWER)
Kubernetes creates a set of namespaces by default when a cluster is initialized. The standard initial namespaces are default, kube-system, kube-public, and kube-node-lease, making A correct.
default is the namespace where resources are created if you don't specify another namespace. Many quick-start examples deploy here, though production environments typically use dedicated namespaces per app/team.
kube-system contains objects created and managed by Kubernetes system components (control plane add-ons, system Pods, controllers, DNS components, etc.). It's a critical namespace, and access is typically restricted.
kube-public is readable by all users (including unauthenticated users in some configurations) and is intended for public cluster information, though it's used sparingly in many environments.
kube-node-lease holds Lease objects used for node heartbeats. This improves scalability by reducing load on etcd compared to older heartbeat mechanisms and helps the control plane track node liveness efficiently.
The incorrect options contain non-standard namespace names like "system," "kube-main," or "kube-primary," and "kube-default" is not a real default namespace. Kubernetes' built-in namespace set is well-documented and consistent with typical cluster bootstraps.
Understanding these namespaces matters operationally: system workloads and controllers often live in kube-system, and many troubleshooting steps involve inspecting Pods and events there. Meanwhile, kube-node-lease is key to node health tracking, and default is the catch-all if you forget to specify -n.
So, the verified answer is A: default, kube-system, kube-public, kube-node-lease.

NEW QUESTION: 33
Which tools enable Kubernetes HorizontalPodAutoscalers to use custom, application-generated metrics to trigger scaling events?
A. Prometheus and the prometheus-adapter.
B. Graylog and graylog-autoscaler metrics.
C. Graylog and the kubernetes-adapter.
D. Grafana and Prometheus.
Answer: A (LEAVE A REPLY)

To scale on custom, application-generated metrics, the Horizontal Pod Autoscaler (HPA) needs those metrics exposed through the Kubernetes custom metrics (or external metrics) API. A common and Kubernetes-documented approach is Prometheus + prometheus-adapter, making A correct. Prometheus scrapes application metrics (for example, request rate, queue depth, in-flight requests) from /metrics endpoints. The prometheus-adapter then translates selected Prometheus time series into the Kubernetes Custom Metrics API so the HPA controller can fetch them and make scaling decisions.
Why not the other options: Grafana is a visualization tool; it does not provide the metrics API translation layer required by HPA, so "Grafana and Prometheus" is incomplete. Graylog is primarily a log management system; it's not the standard solution for feeding custom metrics into HPA via the Kubernetes metrics APIs. The "kubernetes-adapter" term in option C is not the standard named adapter used in the common Kubernetes ecosystem for Prometheus-backed custom metrics (the recognized component is prometheus-adapter).
This matters operationally because HPA is not limited to CPU/memory. CPU and memory use resource metrics (often from metrics-server), but modern autoscaling often needs application signals: message queue length, requests per second, latency, or business metrics. With Prometheus and prometheus-adapter, you can define HPA rules such as "scale to maintain queue depth under X" or "scale based on requests per second per pod." This can produce better scaling behavior than CPU-based scaling alone, especially for I/O-bound services or workloads with uneven CPU profiles.
So the correct tooling combination in the provided choices is Prometheus and the prometheus-adapter, option A.

NEW QUESTION: 34
Which Kubernetes feature would you use to guard against split brain scenarios with your distributed application?
A. Replication controllers
B. Consensus protocols
C. Rolling updates
D. StatefulSet
Answer: (SHOW ANSWER)
The exam-expected Kubernetes feature here is StatefulSet, so D is the correct answer. StatefulSets are designed for distributed/stateful applications that require stable network identities, stable storage, and ordered deployment/termination. Those properties are commonly required by systems that must avoid "split brain" behaviors-where multiple nodes believe they are the leader/primary due to partitions or identity confusion.
StatefulSets give each Pod a persistent identity (e.g., app-0, app-1) and stable DNS naming (typically via a headless Service), which supports consistent peer discovery and membership. They also commonly pair with PersistentVolumeClaims so that each replica keeps its own data across restarts and reschedules. The ordered rollout semantics help clustered systems bootstrap and expand in controlled sequences, reducing the chance of chaotic membership changes.

Important nuance: StatefulSet alone does not magically prevent split brain. Split brain prevention is primarily a property of the application's own clustering/consensus design (e.g., leader election, quorum, fencing). That's why option B ("consensus protocols") is conceptually the true prevention mechanism-but it's not a Kubernetes feature in the way the question frames it. Kubernetes provides primitives that make it feasible to run such systems safely (stable IDs, stable storage, predictable DNS), and StatefulSet is the Kubernetes workload API designed for that class of distributed stateful apps.
Replication controllers and rolling updates don't address identity/quorum concerns. Therefore, within Kubernetes constructs, StatefulSet is the best verified choice for workloads needing stable identity patterns commonly used to reduce split-brain risk.

NEW QUESTION: 35
What is the purpose of the kube-proxy?
A. The kube-proxy balances network requests to Pods.
B. The kube-proxy maintains network rules on nodes.
C. The kube-proxy ensures the cluster connectivity with the internet.
D. The kube-proxy maintains the DNS rules of the cluster.
Answer: (SHOW ANSWER)
The correct answer is B: kube-proxy maintains network rules on nodes. kube-proxy is a node component that implements part of the Kubernetes Service abstraction. It watches the Kubernetes API for Service and EndpointSlice/Endpoints changes, and then programs the node's dataplane rules (commonly iptables or IPVS, depending on configuration) so that traffic sent to a Service virtual IP and port is correctly forwarded to one of the backing Pod endpoints.
This is how Kubernetes provides stable Service addresses even though Pod IPs are ephemeral. When Pods scale up/down or are replaced during a rollout, endpoints change; kube-proxy updates the node rules accordingly. From the perspective of a client, the Service name and ClusterIP remain stable, while the actual backend endpoints are load-distributed.
Option A is a tempting phrasing but incomplete: load distribution is an outcome of the forwarding rules, but kube-proxy's primary role is maintaining the network forwarding rules that make Services work. Option C is incorrect because internet connectivity depends on cluster networking, routing, NAT, and often CNI configuration-not kube-proxy's job description. Option D is incorrect because DNS is typically handled by CoreDNS; kube-proxy does not "maintain DNS rules."
Operationally, kube-proxy failures often manifest as Service connectivity issues: Pod-to-Service traffic fails, ClusterIP routing breaks, NodePort behavior becomes inconsistent, or endpoints aren't updated correctly. Modern Kubernetes environments sometimes replace kube-proxy with eBPF-based dataplanes, but in the classic architecture the correct statement remains: kube-proxy runs on each node and maintains the rules needed for Service traffic steering.

NEW QUESTION: 36
The Container Runtime Interface (CRI) defines the protocol for the communication between:
A. The kubelet and the container runtime.

B. The container runtime and etcd.
C. The kube-apiserver and the kubelet.
D. The container runtime and the image registry.
Answer: (SHOW ANSWER)
The CRI (Container Runtime Interface) defines how the kubelet talks to the container runtime, so A is correct. The kubelet is the node agent responsible for ensuring containers are running in Pods on that node. It needs a standardized way to request operations such as: create a Pod sandbox, pull an image, start/stop containers, execute commands, attach streams, and retrieve logs. CRI provides that contract so kubelet does not need runtime-specific integrations.
This interface is a key part of Kubernetes' modular design. Different container runtimes implement the CRI, allowing Kubernetes to run with containerd, CRI-O, and other CRI-compliant runtimes. This separation of concerns lets Kubernetes focus on orchestration, while runtimes focus on executing containers according to the OCI runtime spec, managing images, and handling low-level container lifecycle.
Why the other options are incorrect:
etcd is the control plane datastore; container runtimes do not communicate with etcd via CRI.
kube-apiserver and kubelet communicate using Kubernetes APIs, but CRI is not their protocol; CRI is specifically kubelet ↔ runtime.
container runtime and image registry communicate using registry protocols (image pull/push APIs), but that is not CRI. CRI may trigger image pulls via runtime requests, yet the actual registry communication is separate.
Operationally, this distinction matters when debugging node issues. If Pods are stuck in "ContainerCreating" due to image pull failures or runtime errors, you often investigate kubelet logs and the runtime (containerd/CRI-O) logs. Kubernetes administrators also care about CRI streaming (exec/attach/logs streaming), runtime configuration, and compatibility across Kubernetes versions.
So, the verified answer is A: the kubelet and the container runtime.

NEW QUESTION: 37
Which component of the node is responsible to run workloads?
A. The kubelet.
B. The kube-proxy.
C. The kube-apiserver.
D. The container runtime.
Answer: D (LEAVE A REPLY)
The verified correct answer is D (the container runtime). On a Kubernetes node, the container runtime (such as containerd or CRI-O) is the component that actually executes containers-it creates container processes, manages their lifecycle, pulls images, and interacts with the underlying OS primitives (namespaces, cgroups) through an OCI runtime like runc. In that direct sense, the runtime is what "runs workloads." It's important to distinguish responsibilities. The kubelet (A) is the node agent that orchestrates what should run on the node: it watches the API

server for Pods assigned to the node and then asks the runtime to start/stop containers accordingly. Kubelet is essential for node management, but it does not itself execute containers; it delegates execution to the runtime via CRI. kube-proxy (B) handles Service traffic routing rules (or is replaced by other dataplanes) and does not run containers. kube-apiserver (C) is a control plane component that stores and serves cluster state; it is not a node workload runner.
So, in the execution chain: scheduler assigns Pod → kubelet sees Pod assigned → kubelet calls runtime via CRI → runtime launches containers. When troubleshooting "containers won't start," you often inspect kubelet logs and runtime logs because the runtime is the component that can fail image pulls, sandbox creation, or container start operations.
Therefore, the best answer to "which node component is responsible to run workloads" is the container runtime, option D.

NEW QUESTION: 38
In the Kubernetes platform, which component is responsible for running containers?
A. etcd
B. CRI-O
C. cloud-controller-manager
D. kube-controller-manager
Answer: (SHOW ANSWER)
In Kubernetes, the actual act of running containers on a node is performed by the container runtime. The kubelet instructs the runtime via CRI, and the runtime pulls images, creates containers, and manages their lifecycle. Among the options provided, CRI-O is the only container runtime, so B is correct.
It's important to be precise: the component that "runs containers" is not the control plane and not etcd. etcd (option A) stores cluster state (API objects) as the backing datastore. It never runs containers. cloud-controller-manager (option C) integrates with cloud APIs for infrastructure like load balancers and nodes. kube-controller-manager (option D) runs controllers that reconcile Kubernetes objects (Deployments, Jobs, Nodes, etc.) but does not execute containers on worker nodes.
CRI-O is a CRI implementation that is optimized for Kubernetes and typically uses an OCI runtime (like runc) under the hood to start containers. Another widely used runtime is containerd. The runtime is installed on nodes and is a prerequisite for kubelet to start Pods. When a Pod is scheduled to a node, kubelet reads the PodSpec and asks the runtime to create a "pod sandbox" and then start the container processes. Runtime behavior also includes pulling images, setting up namespaces/cgroups, and exposing logs/stdout streams back to Kubernetes tooling.
So while "the container runtime" is the most general answer, the question's option list makes CRI-O the correct selection because it is a container runtime responsible for running containers in Kubernetes.

NEW QUESTION: 39
What do Deployments and StatefulSets have in common?

A. They manage Pods that are based on an identical container spec.
B. They support the OnDelete update strategy.
C. They support an ordered, graceful deployment and scaling.
D. They maintain a sticky identity for each of their Pods.
Answer: (SHOW ANSWER)
Both Deployments and StatefulSets are Kubernetes workload controllers that manage a set of Pods created from a Pod template, meaning they manage Pods based on an identical container specification (a shared Pod template). That is why A is correct. In both cases, you declare a desired state (replicas, container images, environment variables, volumes, probes, etc.) in spec.template, and the controller ensures the cluster converges toward that state by creating, updating, or replacing Pods.
The differences are what make the other options incorrect. OnDelete update strategy is associated with StatefulSets (it's one of their update strategies), but it is not a shared, defining behavior across both controllers, so B is not "in common." Ordered, graceful deployment and scaling is a hallmark of StatefulSets (ordered pod creation/termination and stable identities) rather than Deployments, so C is not shared. Sticky identity per Pod (stable network identity and stable storage identity per replica, commonly via StatefulSet + headless Service) is specifically a StatefulSet characteristic, not a Deployment feature, so D is not common.
A useful way to think about it is: both controllers manage replicas of a Pod template, but they differ in semantics. Deployments are designed primarily for stateless workloads and typically focus on rolling updates and scalable replicas where any instance is interchangeable. StatefulSets are designed for stateful workloads and add identity and ordering guarantees: each replica gets a stable name (like db-0, db-1) and often stable PersistentVolumeClaims.
So the shared commonality the question is testing is the basic workload-controller pattern: both controllers manage Pods created from a common template (identical container spec). Therefore, A is the verified answer.

NEW QUESTION: 40
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: (SHOW ANSWER)
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.

NEW QUESTION: 41
Which one of the following is an open source runtime security tool?
A. lxd
B. containerd
C. falco
D. gVisor
Answer: (SHOW ANSWER)
The correct answer is C: Falco. Falco is a widely used open-source runtime security tool (originally created by Sysdig and now a CNCF project) designed to detect suspicious behavior at runtime by monitoring system calls and other kernel-level signals. In Kubernetes environments, Falco helps identify threats such as unexpected shell access in containers, privilege escalation attempts, access to sensitive files, anomalous network tooling, crypto-mining patterns, and other behaviors that indicate compromise or policy violations.
The other options are not primarily "runtime security tools" in the detection/alerting sense:
containerd is a container runtime responsible for executing containers; it's not a security detection tool.
lxd is a system container and VM manager; again, not a runtime threat detection tool.
gVisor is a sandboxed container runtime that improves isolation by interposing a user-space kernel; it's a security mechanism, but the question asks for a runtime security tool (monitoring/detection). Falco fits that definition best.
In cloud-native security practice, Falco typically runs as a DaemonSet so it can observe activity on every node. It uses rules to define what "bad" looks like and can emit alerts to SIEM systems, logging backends, or incident response workflows. This complements preventative controls like

RBAC, Pod Security Admission, seccomp, and least privilege configurations. Preventative controls reduce risk; Falco provides visibility and detection when something slips through. Therefore, among the provided choices, the verified runtime security tool is Falco (C).

NEW QUESTION: 42
Imagine there is a requirement to run a database backup every day. Which Kubernetes resource could be used to achieve that?
A. kube-scheduler
B. CronJob
C. Task
D. Job
Answer: (SHOW ANSWER)
To run a workload on a repeating schedule (like "every day"), Kubernetes provides CronJob, making B correct. A CronJob creates Jobs according to a cron-formatted schedule, and then each Job creates one or more Pods that run to completion. This is the Kubernetes-native replacement for traditional cron scheduling, but implemented as a declarative resource managed by controllers in the cluster.
For a daily database backup, you'd define a CronJob with a schedule (e.g., "0 2 * * *" for 2:00 AM daily), and specify the Pod template that performs the backup (invokes backup scripts/tools, writes output to durable storage, uploads to object storage, etc.). Kubernetes will then create a Job at each scheduled time. CronJobs also support operational controls like concurrencyPolicy (Allow/Forbid/Replace) to decide what happens if a previous backup is still running, startingDeadlineSeconds to handle missed schedules, and history limits to retain recent successful/failed Job records for debugging.
Option D (Job) is close but not sufficient for "every day." A Job runs a workload until completion once; you would need an external scheduler to create a Job every day. Option A (kube-scheduler) is a control plane component responsible for placing Pods onto nodes and does not schedule recurring tasks. Option C ("Task") is not a standard Kubernetes workload resource.
This question is fundamentally about mapping a recurring operational requirement (backup cadence) to Kubernetes primitives. The correct design is: CronJob triggers Job creation on a schedule; Job runs Pods to completion. Therefore, the correct answer is B.

NEW QUESTION: 43
Which group of container runtimes provides additional sandboxed isolation and elevated security?
A. rune, cgroups
B. docker, containerd
C. runsc, kata
D. crun, cri-o
Answer: (SHOW ANSWER)
The runtimes most associated with sandboxed isolation are gVisor's runsc and Kata Containers, making C correct. Standard container runtimes (like containerd with runc) rely primarily on Linux

namespaces and cgroups for isolation. That isolation is strong for many use cases, but it shares the host kernel, which can be a concern for multi-tenant or high-risk workloads.
gVisor (runsc) provides a user-space kernel-like layer that intercepts and mediates system calls, reducing the container's direct interaction with the host kernel. Kata Containers takes a different approach: it runs containers inside lightweight virtual machines, providing hardware-virtualization boundaries (or VM-like isolation) while still integrating into container workflows. Both are used to increase isolation compared to traditional containers, and both can be integrated with Kubernetes through compatible CRI/runtime configurations.
The other options are incorrect for the question's intent. "rune, cgroups" is not a meaningful pairing here (cgroups is a Linux resource mechanism, not a runtime). "docker, containerd" are commonly used container platforms/runtimes but are not specifically the "sandboxed isolation" category (containerd typically uses runc for standard isolation). "crun, cri-o" represents a low-level OCI runtime (crun) and a CRI implementation (CRI-O), again not specifically a sandboxed-isolation grouping.
So, when the question asks for the group that provides additional sandboxing and elevated security, the correct, well-established answer is runsc + Kata.

NEW QUESTION: 44
What is the practice of bringing financial accountability to the variable spend model of cloud resources?
A. FaaS
B. DevOps
C. CloudCost
D. FinOps
Answer: (SHOW ANSWER)
The practice of bringing financial accountability to cloud spending-where costs are variable and usage-based-is called FinOps, so D is correct. FinOps (Financial Operations) is an operating model and culture that helps organizations manage cloud costs by connecting engineering, finance, and business teams. Because cloud resources can be provisioned quickly and billed dynamically, traditional budgeting approaches often fail to keep pace. FinOps addresses this by introducing shared visibility, governance, and optimization processes that enable teams to make cost-aware decisions while still moving fast.
In Kubernetes and cloud-native architectures, variable spend shows up in many ways: autoscaling node pools, over-provisioned resource requests, idle clusters, persistent volumes, load balancers, egress traffic, managed services, and observability tooling. FinOps practices encourage tagging/labeling for cost attribution, defining cost KPIs, enforcing budget guardrails, and continuously optimizing usage (right-sizing resources, scaling policies, turning off unused environments, and selecting cost-effective architectures).
Why the other options are incorrect: FaaS (Function as a Service) is a compute model (serverless), not a financial accountability practice. DevOps is a cultural and technical practice

focused on collaboration and delivery speed, not specifically cloud cost accountability (though it can complement FinOps). CloudCost is not a widely recognized standard term in the way FinOps is.
In practice, FinOps for Kubernetes often involves improving resource efficiency: aligning requests/limits with real usage, using HPA/VPA appropriately, selecting instance types that match workload profiles, managing cluster autoscaler settings, and allocating shared platform costs to teams via labels/namespaces. It also includes forecasting and anomaly detection, because cloud-native spend can spike quickly due to misconfigurations (e.g., runaway autoscaling or excessive log ingestion).
So, the correct term for financial accountability in cloud variable spend is FinOps (D).

NEW QUESTION: 45
Which Kubernetes-native deployment strategy supports zero-downtime updates of a workload?
A. Canary
B. Recreate
C. BlueGreen
D. RollingUpdate
Answer: (SHOW ANSWER)
D (RollingUpdate) is correct. In Kubernetes, the Deployment resource's default update strategy is RollingUpdate, which replaces Pods gradually rather than all at once. This supports zero-downtime updates when the workload is properly configured (sufficient replicas, correct readiness probes, and appropriate maxUnavailable / maxSurge settings). As new Pods come up and become Ready, old Pods are terminated in a controlled way, keeping the service available throughout the rollout.
RollingUpdate's "zero downtime" is achieved by maintaining capacity while transitioning between versions. For example, with multiple replicas, Kubernetes can create new Pods, wait for readiness, then scale down old Pods, ensuring traffic continues to flow to healthy instances. Readiness probes are critical: they prevent traffic from being routed to a Pod until it's actually ready to serve.
Why other options are not the Kubernetes-native "strategy" answer here:
Recreate (B) explicitly stops old Pods before starting new ones, causing downtime for most services.
Canary (A) and BlueGreen (C) are real deployment patterns, but in "Kubernetes-native deployment strategy" terms, the built-in Deployment strategies are RollingUpdate and Recreate. Canary/BlueGreen typically require additional tooling/controllers (service mesh, ingress controller features, or progressive delivery operators) to manage traffic shifting between versions.
So, for a Kubernetes-native strategy that supports zero-downtime updates, the correct and verified choice is RollingUpdate (D).

NEW QUESTION: 46
How can you extend the Kubernetes API?

A. Adding a CustomResourceDefinition or implementing an aggregation layer.
B. Adding a new version of a resource, for instance v4beta3.
C. With the command kubectl extend api, logged in as an administrator.
D. Adding the desired API object as a kubelet parameter.
Answer: (SHOW ANSWER)
A is correct: Kubernetes' API can be extended by adding CustomResourceDefinitions (CRDs) and/or by implementing the API Aggregation Layer. These are the two canonical extension mechanisms.
CRDs let you define new resource types (new kinds) that the Kubernetes API server stores in etcd and serves like native objects. You typically pair a CRD with a controller/operator that watches those custom objects and reconciles real resources accordingly. This pattern is foundational to the Kubernetes ecosystem (many popular add-ons install CRDs).
The aggregation layer allows you to add entire API services (aggregated API servers) that serve additional endpoints under the Kubernetes API. This is used when you want custom API behavior, custom storage, or specialized semantics beyond what CRDs provide (or when implementing APIs like metrics APIs historically).
Why the other answers are wrong:
B is not how API extension works. You don't "extend the API" by inventing new versions like v4beta3; versions are defined and implemented by API servers/controllers, not by users arbitrarily.
C is fictional; there is no standard kubectl extend api command.
D is also incorrect; kubelet parameters configure node agent behavior, not API server types and discovery.
So, the verified ways to extend Kubernetes' API surface are CRDs and API aggregation, which is option A.

NEW QUESTION: 47
What is the minimum number of etcd members that are required for a highly available Kubernetes cluster?
A. Two etcd members.
B. Five etcd members.
C. Six etcd members.

D. Three etcd members.

Answer: (SHOW ANSWER)

D (three etcd members) is correct. etcd is a distributed key-value store that uses the Raft consensus algorithm. High availability in consensus systems depends on maintaining a quorum (majority) of members to continue serving writes reliably. With 3 members, the cluster can tolerate 1 failure and still have 2/3 available-enough for quorum.

Two members is a common trap: with 2, a single failure leaves 1/2, which is not a majority, so the cluster cannot safely make progress. That means 2-member etcd is not HA; it is fragile and can be taken down by one node loss, network partition, or maintenance event. Five members can tolerate 2 failures and is a valid HA configuration, but it is not the minimum. Six is even-sized and generally discouraged for consensus because it doesn't improve failure tolerance compared to five (quorum still requires 4), while increasing coordination overhead.

In Kubernetes, etcd reliability directly affects the API server and the entire control plane because etcd stores cluster state: object specs, status, controller state, and more. If etcd loses quorum, the API server will be unable to persist or reliably read/write state, leading to cluster management outages. That's why the minimum HA baseline is three etcd members, often across distinct failure domains (nodes/AZs), with strong disk performance and consistent low-latency networking.

So, the smallest etcd topology that provides true fault tolerance is 3 members, which corresponds to option D.

NEW QUESTION: 48

What factors influence the Kubernetes scheduler when it places Pods on nodes?

A. Pod memory requests, node taints, and Pod affinity.

B. Pod labels, node labels, and request labels.

C. Node taints, node level, and Pod priority.

D. Pod priority, container command, and node labels.

Answer: A (LEAVE A REPLY)

The Kubernetes scheduler chooses a node for a Pod by evaluating scheduling constraints and cluster state. Key inputs include resource requests (CPU/memory), taints/tolerations, and affinity/anti-affinity rules. Option A directly names three real, high-impact scheduling factors-Pod memory requests, node taints, and Pod affinity-so A is correct.

Resource requests are fundamental: the scheduler must ensure the target node has enough allocatable CPU/memory to satisfy the Pod's requests. Requests (not limits) drive placement decisions. Taints on nodes repel Pods unless the Pod has a matching toleration, which is commonly used to reserve nodes for special workloads (GPU nodes, system nodes, restricted nodes) or to protect nodes under certain conditions. Affinity and anti-affinity allow expressing "place me near" or "place me away" rules-e.g., keep replicas spread across failure domains or co-locate components for latency.

Option B includes labels, which do matter, but "request labels" is not a standard scheduler concept; labels influence scheduling mainly through selectors and affinity, not as a direct category called "request labels." Option C mixes a real concept (taints, priority) with "node level," which

isn't a standard scheduling factor term. Option D includes "container command," which does not influence scheduling; the scheduler does not care what command the container runs, only placement constraints and resources.
Under the hood, kube-scheduler uses a two-phase process (filtering then scoring) to select a node, but the inputs it filters/scores include exactly the kinds of constraints in A. Therefore, the verified best answer is A.

NEW QUESTION: 49
What are the 3 pillars of Observability?
A. Metrics, Logs, and Traces
B. Metrics, Logs, and Spans
C. Metrics, Data, and Traces
D. Resources, Logs, and Tracing
Answer: (SHOW ANSWER)
The correct answer is A: Metrics, Logs, and Traces. These are widely recognized as the "three pillars" because together they provide complementary views into system behavior:
Metrics are numeric time series collected over time (CPU usage, request rate, error rate, latency percentiles). They are best for dashboards, alerting, and capacity planning because they are structured and aggregatable. In Kubernetes, metrics underpin autoscaling and operational visibility (node/pod resource usage, cluster health signals).
Logs are discrete event records (often text) emitted by applications and infrastructure components. Logs provide detailed context for debugging: error messages, stack traces, warnings, and business events. In Kubernetes, logs are commonly collected from container stdout/stderr and aggregated centrally for search and correlation.
Traces capture the end-to-end journey of a request through a distributed system, breaking it into spans. Tracing is crucial in microservices because a single user request may cross many services; traces show where latency accumulates and which dependency fails. Tracing also enables root cause analysis when metrics indicate degradation but don't pinpoint the culprit.
Why the other options are wrong: a span is a component within tracing, not a top-level pillar; "data" is too generic; and "resources" are not an observability signal category. The pillars are defined by signal type and how they're used operationally.
In cloud-native practice, these pillars are often unified via correlation IDs and shared context: metrics alerts link to logs and traces for the same timeframe/request. Tooling like Prometheus (metrics), log aggregators (e.g., Loki/Elastic), and tracing systems (Jaeger/Tempo/OpenTelemetry) work together to provide a complete observability story.
Therefore, the verified correct answer is A.

NEW QUESTION: 50
Which of the following systems is NOT compatible with the CRI runtime interface standard?
(Typo corrected: "CRI-0" → "CRI-O")

A. CRI-O

B. dockershim
C. systemd
D. containerd
Answer: (SHOW ANSWER)
Kubernetes uses the Container Runtime Interface (CRI) to support pluggable container runtimes. The kubelet talks to a CRI-compatible runtime via gRPC, and that runtime is responsible for pulling images and running containers. In this context, containerd and CRI-O are CRI-compatible container runtimes (or runtime stacks) used widely with Kubernetes, and dockershim historically served as a compatibility layer that allowed kubelet to talk to Docker Engine as if it were CRI (before dockershim was removed from kubelet in newer Kubernetes versions). That leaves systemd as the correct "NOT compatible with CRI" answer, so C is correct.
systemd is an init system and service manager for Linux. While it can be involved in how services (like kubelet) are started and managed on the host, it is not a container runtime implementing CRI. It does not provide CRI gRPC endpoints for kubelet, nor does it manage containers in the CRI sense.
The deeper Kubernetes concept here is separation of responsibilities: kubelet is responsible for Pod lifecycle at the node level, but it delegates "run containers" to a runtime via CRI. Runtimes like containerd and CRI-O implement that contract; Kubernetes can swap them without changing kubelet logic. Historically, dockershim translated kubelet's CRI calls into Docker Engine calls. Even though dockershim is no longer part of kubelet, it was still "CRI-adjacent" in purpose and often treated as compatible in older curricula.
Therefore, among the provided options, systemd is the only one that is clearly not a CRI-compatible runtime system, making C correct.

NEW QUESTION: 51
The Kubernetes project work is carried primarily by SIGs. What does SIG stand for?
A. Special Interest Group
B. Software Installation Guide
C. Support and Information Group
D. Strategy Implementation Group
Answer: (SHOW ANSWER)
In Kubernetes governance and project structure, SIG stands for Special Interest Group, so A is correct. Kubernetes is a large open source project under the Cloud Native Computing Foundation (CNCF), and its work is organized into groups that focus on specific domains-such as networking, storage, node, scheduling, security, docs, testing, and many more. SIGs provide a scalable way to coordinate contributors, prioritize work, review design proposals (KEPs), triage issues, and manage releases in their area.
Each SIG typically has regular meetings, mailing lists, chat channels, and maintainers who guide the direction of that part of the project. For example, SIG Network focuses on Kubernetes networking architecture and components, SIG Storage on storage APIs and CSI integration, and

SIG Scheduling on scheduler behavior and extensibility. This structure helps Kubernetes evolve while maintaining quality, review rigor, and community-driven decision making.
The other options are not part of Kubernetes project terminology. "Software Installation Guide" and the others might sound plausible, but they are not how Kubernetes defines SIGs.
Understanding SIGs matters operationally because many Kubernetes features and design changes originate from SIGs. When you read Kubernetes enhancement proposals, release notes, or documentation, you'll often see SIG ownership and references. In short, SIGs are the primary organizational units for Kubernetes engineering and stewardship, and SIG = Special Interest Group.

NEW QUESTION: 52
What is Flux constructed with?
A. GitLab Environment Toolkit
B. GitOps Toolkit
C. Helm Toolkit
D. GitHub Actions Toolkit
Answer: (SHOW ANSWER)
The correct answer is B: GitOps Toolkit. Flux is a GitOps solution for Kubernetes, and in Flux v2 the project is built as a set of Kubernetes controllers and supporting components collectively referred to as the GitOps Toolkit. This toolkit provides the building blocks for implementing GitOps reconciliation: sourcing artifacts (Git repositories, Helm repositories, OCI artifacts), applying manifests (Kustomize/Helm), and continuously reconciling cluster state to match the desired state declared in Git.
This construction matters because it reflects Flux's modular architecture. Instead of being a single monolithic daemon, Flux is composed of controllers that each handle a part of the GitOps workflow: fetching sources, rendering configuration, and applying changes. This makes it more Kubernetes-native: everything is declarative, runs in the cluster, and can be managed like other workloads (RBAC, namespaces, upgrades, observability).
Why the other options are wrong:
"GitLab Environment Toolkit" and "GitHub Actions Toolkit" are not what Flux is built from. Flux can integrate with many SCM providers and CI systems, but it is not "constructed with" those.
"Helm Toolkit" is not the named foundational set Flux is built upon. Flux can deploy Helm charts, but that's a capability, not its underlying construction.
In cloud-native delivery, Flux implements the key GitOps control loop: detect changes in Git (or other declared sources), compute desired Kubernetes state, and apply it while continuously checking for drift. The GitOps Toolkit is the set of controllers enabling that loop.
Therefore, the verified correct answer is B.

NEW QUESTION: 53

Kubernetes ___ allows you to automatically manage the number of nodes in your cluster to meet demand.

A. Node Autoscaler

B. Cluster Autoscaler

C. Horizontal Pod Autoscaler

D. Vertical Pod Autoscaler

Answer: (SHOW ANSWER)

Kubernetes supports multiple autoscaling mechanisms, but they operate at different layers. The question asks specifically about automatically managing the number of nodes in the cluster, which is the role of the Cluster Autoscaler-therefore B is correct.

Cluster Autoscaler monitors the scheduling state of the cluster. When Pods are pending because there are not enough resources (CPU/memory) available on existing nodes-meaning the scheduler cannot place them-Cluster Autoscaler can request that the underlying infrastructure (typically a cloud provider node group / autoscaling group) add nodes. Conversely, when nodes are underutilized and Pods can be rescheduled elsewhere, Cluster Autoscaler can drain those nodes (respecting disruption constraints like PodDisruptionBudgets) and then remove them to reduce cost. This aligns with cloud-native elasticity: scale infrastructure up and down automatically based on workload needs.

The other options are different: Horizontal Pod Autoscaler (HPA) changes the number of Pod replicas for a workload (like a Deployment) based on metrics (CPU utilization, memory, or custom metrics). It scales the application layer, not the node layer. Vertical Pod Autoscaler (VPA) changes resource requests/limits (CPU/memory) for Pods, effectively "scaling up/down" the size of individual Pods. It also does not directly change node count, though its adjustments can influence scheduling pressure. "Node Autoscaler" is not the canonical Kubernetes component name used in standard terminology; the widely referenced upstream component for node count is Cluster Autoscaler.

In real systems, these autoscalers often work together: HPA increases replicas when traffic rises; that may cause Pods to go Pending if nodes are full; Cluster Autoscaler then adds nodes; scheduling proceeds; later, traffic drops, HPA reduces replicas and Cluster Autoscaler removes nodes. This layered approach provides both performance and cost efficiency.

NEW QUESTION: 54

What can be used to create a job that will run at specified times/dates or on a repeating schedule?

A. Job

B. CalendarJob

C. BatchJob

D. CronJob

Answer: (SHOW ANSWER)

The correct answer is D: CronJob. A Kubernetes CronJob is specifically designed for creating Jobs on a schedule-either at specified times/dates (expressed via cron syntax) or on a repeating

interval (hourly, daily, weekly). When the schedule triggers, the CronJob controller creates a Job, and the Job controller creates the Pods that execute the workload to completion.
Option A (Job) is not inherently scheduled. A Job runs when you create it, and it continues until it completes successfully or fails according to its retry/backoff behavior. If you want it to run periodically, you need something else to create the Job each time. CronJob is the built-in mechanism for that scheduling.
Options B and C are not standard Kubernetes workload objects. Kubernetes does not include "CalendarJob" or "BatchJob" as official API kinds. The scheduling primitive is CronJob.
CronJobs also include important operational controls: concurrency policies prevent overlapping runs, deadlines control missed schedules, and history limits manage old Job retention. This makes CronJobs more robust than ad-hoc scheduling approaches and keeps the workload lifecycle visible in the Kubernetes API (status/events/logs). It also means you can apply standard Kubernetes patterns: use a service account with least privilege, mount Secrets/ConfigMaps, run in specific namespaces, and manage resource requests/limits so that scheduled workloads don't destabilize the cluster.
So the correct Kubernetes resource for scheduled and repeating job execution is CronJob (D).

NEW QUESTION: 55
What is the difference between a Deployment and a ReplicaSet?
A. With a Deployment, you can't control the number of pod replicas.
B. A ReplicaSet does not guarantee a stable set of replica pods running.
C. A Deployment is basically the same as a ReplicaSet with annotations.
D. A Deployment is a higher-level concept that manages ReplicaSets.
Answer: (SHOW ANSWER)
A Deployment is a higher-level controller that manages ReplicaSets and provides rollout/rollback behavior, so D is correct. A ReplicaSet's primary job is to ensure that a specified number of Pod replicas are running at any time, based on a label selector and Pod template. It's a fundamental "keep N Pods alive" controller.
Deployments build on that by managing the lifecycle of ReplicaSets over time. When you update a Deployment (for example, changing the container image tag or environment variables), Kubernetes creates a new ReplicaSet for the new Pod template and gradually shifts replicas from the old ReplicaSet to the new one according to the rollout strategy (RollingUpdate by default). Deployments also retain revision history, making it possible to roll back to a previous ReplicaSet if a rollout fails.
Why the other options are incorrect:
A is false: Deployments absolutely control the number of replicas via spec.replicas and can also be controlled by HPA.
B is false: ReplicaSets do guarantee that a stable number of replicas is running (that is their core purpose).
C is false: a Deployment is not "a ReplicaSet with annotations." It is a distinct API resource with additional controller logic for declarative updates, rollouts, and revision tracking.

Operationally, most teams create Deployments rather than ReplicaSets directly because Deployments are safer and more feature-complete for application delivery. ReplicaSets still appear in real clusters because Deployments create them automatically; you'll commonly see multiple ReplicaSets during rollout transitions. Understanding the hierarchy is crucial for troubleshooting: if Pods aren't behaving as expected, you often trace from Deployment → ReplicaSet → Pod, checking selectors, events, and rollout status.
So the key difference is: ReplicaSet maintains replica count; Deployment manages ReplicaSets and orchestrates updates. Therefore, D is the verified answer.

NEW QUESTION: 56
If a Pod was waiting for container images to download on the scheduled node, what state would it be in?
A. Failed
B. Succeeded
C. Unknown
D. Pending
Answer: (SHOW ANSWER)
If a Pod is waiting for its container images to be pulled to the node, it remains in the Pending phase, so D is correct. Kubernetes Pod "phase" is a high-level summary of where the Pod is in its lifecycle. Pending means the Pod has been accepted by the cluster but one or more of its containers has not started yet. That can occur because the Pod is waiting to be scheduled, waiting on volume attachment/mount, or-very commonly-waiting for the container runtime to pull the image.
When image pulling is the blocker, kubectl describe pod <name> usually shows events like "Pulling image ..." and "Successfully pulled image ..." or failures like ImagePullBackOff/ErrImagePull. Even if the node has been assigned (scheduler has set spec.nodeName), the Pod can still be Pending while kubelet and the runtime perform preparation steps.
Why the other phases don't apply:
Succeeded is for run-to-completion Pods that have finished successfully (typical for Jobs).
Failed means the Pod terminated and at least one container terminated in failure (and won't be restarted, depending on restartPolicy).
Unknown is used when the node can't be contacted and the Pod's state can't be reliably determined (rare in healthy clusters).
A subtle but important Kubernetes detail: status "Waiting" reasons like ImagePullBackOff are container states inside .status.containerStatuses, while the Pod phase can still be Pending. So, "waiting for images to download" maps to Pod Pending, with container waiting reasons providing the deeper diagnosis.
Therefore, the verified correct answer is D: Pending.

NEW QUESTION: 57

Which of the following is a definition of Hybrid Cloud?
A. A combination of services running in public and private data centers, only including data centers from the same cloud provider.
B. A cloud native architecture that uses services running in public clouds, excluding data centers in different availability zones.
C. A cloud native architecture that uses services running in different public and private clouds, including on-premises data centers.
D. A combination of services running in public and private data centers, excluding serverless functions.
Answer: C (LEAVE A REPLY)
A hybrid cloud architecture combines public cloud and private/on-premises environments, often spanning multiple infrastructure domains while maintaining some level of portability, connectivity, and unified operations. Option C captures the commonly accepted definition: services run across public and private clouds, including on-premises data centers, so C is correct.
Hybrid cloud is not limited to a single cloud provider (which is why A is too restrictive). Many organizations adopt hybrid cloud to meet regulatory requirements, data residency constraints, latency needs, or to preserve existing investments while still using public cloud elasticity. In Kubernetes terms, hybrid strategies often include running clusters both on-prem and in one or more public clouds, then standardizing deployment through Kubernetes APIs, GitOps, and consistent security/observability practices.
Option B is incorrect because excluding data centers in different availability zones is not a defining property; in fact, hybrid deployments commonly use multiple zones/regions for resilience.
Option D is a distraction: serverless inclusion or exclusion does not define hybrid cloud. Hybrid is about the combination of infrastructure environments, not a specific compute model.
A practical cloud-native view is that hybrid architectures introduce challenges around identity, networking, policy enforcement, and consistent observability across environments. Kubernetes helps because it provides a consistent control plane API and workload model regardless of where it runs. Tools like service meshes, federated identity, and unified monitoring can further reduce fragmentation.
So, the most accurate definition in the given choices is C: hybrid cloud combines public and private clouds, including on-premises infrastructure, to run services in a coordinated architecture.

NEW QUESTION: 58
What helps an organization to deliver software more securely at a higher velocity?
A. Kubernetes
B. apt-get
C. Docker Images
D. CI/CD Pipeline
Answer: (SHOW ANSWER)
A CI/CD pipeline is a core practice/tooling approach that enables organizations to deliver software faster and more securely, so D is correct. CI (Continuous Integration) automates building and

testing code changes frequently, reducing integration risk and catching defects early. CD (Continuous Delivery/Deployment) automates releasing validated builds into environments using consistent, repeatable steps-reducing manual errors and enabling rapid iteration.
Security improves because automation enables standardized checks on every change: static analysis, dependency scanning, container image scanning, policy validation, and signing/verification steps can be integrated into the pipeline. Instead of relying on ad-hoc human processes, security controls become repeatable gates. In Kubernetes environments, pipelines commonly build container images, run tests, publish artifacts to registries, and then deploy via manifests, Helm, or GitOps controllers-keeping deployments consistent and auditable.
Option A (Kubernetes) is a platform that helps run and manage workloads, but by itself it doesn't guarantee secure high-velocity delivery. It provides primitives (rollouts, declarative config, RBAC), yet the delivery workflow still needs automation. Option B (apt-get) is a package manager for Debian-based systems and is not a delivery pipeline. Option C (Docker Images) are artifacts; they improve portability and repeatability, but they don't provide the end-to-end automation of building, testing, promoting, and deploying across environments.
In cloud-native application delivery, the pipeline is the "engine" that turns code changes into safe production releases. Combined with Kubernetes' declarative deployment model (Deployments, rolling updates, health probes), a CI/CD pipeline supports frequent releases with controlled rollouts, fast rollback, and strong auditability. That is exactly what the question is targeting. Therefore, the verified answer is D.

NEW QUESTION: 59
What is the core functionality of GitOps tools like Argo CD and Flux?
A. They track production changes made by a human in a Git repository and generate a human-readable audit trail.
B. They replace human operations with an agent that tracks Git commands.
C. They automatically create pull requests when dependencies are outdated.
D. They continuously compare the desired state in Git with the actual production state and notify or act upon differences.
Answer: (SHOW ANSWER)
The defining capability of GitOps controllers such as Argo CD and Flux is continuous reconciliation: they compare the desired state stored in Git to the actual state in the cluster and then alert and/or correct drift, making D correct. In GitOps, Git becomes the single source of truth for declarative configuration (Kubernetes manifests, Helm charts, Kustomize overlays). The controller watches Git for changes and applies them, and it also watches the cluster for divergence.
This is more than "auditing human changes" (option A). GitOps does provide auditability because changes are made via commits and pull requests, but the core functionality is the reconciliation loop that keeps cluster state aligned with Git, including optional automated sync/remediation.
Option B is not accurate because GitOps is not about tracking user Git commands; it's about reconciling desired state definitions. Option C (automatically creating pull requests for outdated

dependencies) is a useful feature in some tooling ecosystems, but it is not the central defining behavior of GitOps controllers.
In Kubernetes delivery terms, this approach improves reliability: rollouts become repeatable, configuration drift is detected, and recovery is simpler (reapply known-good state from Git). It also supports separation of duties: platform teams can control policies and base layers, while app teams propose changes via PRs.
So the verified statement is: GitOps tools continuously reconcile Git desired state with cluster actual state-exactly option D.

NEW QUESTION: 60

How is application data maintained in containers?

A. Store data into data folders.
B. Store data into volumes.
C. Store data into sidecar containers.
D. Store data in separate folders.
Answer: B (LEAVE A REPLY)
Container filesystems are ephemeral: the writable layer is tied to the container lifecycle and can be lost when containers are recreated. Therefore, maintaining application data correctly means storing it in volumes, making D the correct answer. In Kubernetes, volumes provide durable or shareable storage that is mounted into containers at specific paths. Depending on the volume type, the data can persist across container restarts and even Pod rescheduling.
Kubernetes supports many volume patterns. For transient scratch data you might use emptyDir (ephemeral for the Pod's lifetime). For durable state, you typically use PersistentVolumes consumed by PersistentVolumeClaims (PVCs), backed by storage systems via CSI drivers (cloud disks, SAN/NAS, distributed storage). This decouples the application container image from its state and enables rolling updates, rescheduling, and scaling without losing data.
Options A and B ("folders") are incomplete because folders inside the container filesystem do not guarantee persistence. A folder is only as durable as the underlying storage; without a mounted volume, it lives in the container's writable layer and will disappear when the container is replaced.
Option C is incorrect because "sidecar containers" are not a data durability mechanism; sidecars can help ship logs or sync data, but persistent data should still be stored on volumes (or external services like managed databases).
From an application delivery standpoint, the principle is: containers should be immutable and disposable, and state should be externalized. Volumes (and external managed services) make this possible. In Kubernetes, this is a foundational pattern enabling safe rollouts, self-healing, and portability: the platform can kill and recreate Pods freely because data is maintained independently via volumes.
Therefore, the verified correct choice is D: Store data into volumes.

NEW QUESTION: 61

Which of the following options include resources cleaned by the Kubernetes garbage collection mechanism?

A. Stale or expired CertificateSigningRequests (CSRs) and old deployments.

B. Nodes deleted by a cloud controller manager and obsolete logs from the kubelet.

C. Unused container and container images, and obsolete logs from the kubelet.

D. Terminated pods, completed jobs, and objects without owner references.

Answer: (SHOW ANSWER)

Kubernetes garbage collection (GC) is about cleaning up API objects and related resources that are no longer needed, so the correct answer is D. Two big categories it targets are (1) objects that have finished their lifecycle (like terminated Pods and completed Jobs, depending on controllers and TTL policies), and (2) "dangling" objects that are no longer referenced properly-often described as objects without owner references (or where owners are gone), which can happen when a higher-level controller is deleted or when dependent resources are left behind.

A key Kubernetes concept here is OwnerReferences: many resources are created "owned" by a controller (e.g., a ReplicaSet owned by a Deployment, Pods owned by a ReplicaSet). When an owning object is deleted, Kubernetes' garbage collector can remove dependent objects based on deletion propagation policies (foreground/background/orphan). This prevents resource leaks and keeps the cluster tidy and performant.

The other options are incorrect because they refer to cleanup tasks outside Kubernetes GC's scope. Kubelet logs (B/C) are node-level files and log rotation is handled by node/runtime configuration, not the Kubernetes garbage collector. Unused container images (C) are managed by the container runtime's image GC and kubelet disk pressure management, not the Kubernetes API GC. Nodes deleted by a cloud controller (B) aren't "garbage collected" in the same sense; node lifecycle is handled by controllers and cloud integrations, but not as a generic GC cleanup category like ownerRef-based object deletion.

So, when the question asks specifically about "resources cleaned by Kubernetes garbage collection," it's pointing to Kubernetes object lifecycle cleanup: terminated Pods, completed Jobs, and orphaned objects-exactly what option D states.

NEW QUESTION: 62

What is the correct hierarchy of Kubernetes components?

A. Containers → Pods → Cluster → Nodes

B. Nodes → Cluster → Containers → Pods
C. Cluster → Nodes → Pods → Containers
D. Pods → Cluster → Containers → Nodes
Answer: (SHOW ANSWER)
The correct answer is C: Cluster → Nodes → Pods → Containers. This expresses the fundamental structural relationship in Kubernetes. A cluster is the overall system (control plane + nodes) that runs your workloads. Inside the cluster, you have nodes (worker machines-VMs or bare metal) that provide CPU, memory, storage, and networking. The scheduler assigns workloads to nodes.
Workloads are executed as Pods, which are the smallest deployable units Kubernetes schedules. Pods represent one or more containers that share networking (one Pod IP and port space) and can share storage volumes. Within each Pod are containers, which are the actual application processes packaged with their filesystem and runtime dependencies.
The other options are incorrect because they break these containment relationships. Containers do not contain Pods; Pods contain containers. Nodes do not exist "inside" Pods; Pods run on nodes. And the cluster is the top-level boundary that contains nodes and orchestrates Pods.
This hierarchy matters for troubleshooting and design. If you're thinking about capacity, you reason at the node and cluster level (node pools, autoscaling, quotas). If you're thinking about application scaling, you reason at the Pod level (replicas, HPA, readiness probes). If you're thinking about process-level concerns, you reason at the container level (images, security context, runtime user, resources). Kubernetes intentionally uses this layered model so that scheduling and orchestration operate on Pods, while the container runtime handles container execution details.
So the accurate hierarchy from largest to smallest unit is: Cluster → Nodes → Pods → Containers, which corresponds to C.

NEW QUESTION: 63
What is the name of the Kubernetes resource used to expose an application?
A. Port
B. Service
C. DNS
D. Deployment
Answer: (SHOW ANSWER)
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).

NEW QUESTION: 64
Which of the following is a recommended security habit in Kubernetes?
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: (SHOW ANSWER)
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.

NEW QUESTION: 65
Which of the following is the name of a container orchestration software?

A. OpenStack
B. Docker
C. Apache Mesos
D. CRI-O
Answer: (SHOW ANSWER)
C (Apache Mesos) is correct because Mesos is a cluster manager/orchestrator that can schedule and manage workloads (including containerized workloads) across a pool of machines.
Historically, Mesos (often paired with frameworks like Marathon) was used to orchestrate services and batch jobs at scale, similar in spirit to Kubernetes' scheduling and cluster management role.
Why the other answers are not correct as "container orchestration software" in this context:
OpenStack (A) is primarily an IaaS cloud platform for provisioning compute, networking, and storage (VM-focused). It's not a container orchestrator, though it can host Kubernetes or containers.
Docker (B) is a container platform/tooling ecosystem (image build, runtime, local orchestration via Docker Compose/Swarm historically), but "Docker" itself is not the best match for "container orchestration software" in the multi-node cluster orchestration sense that the question implies.
CRI-O (D) is a container runtime implementing Kubernetes' CRI; it runs containers on a node but does not orchestrate placement, scaling, or service lifecycle across a cluster.
Container orchestration typically means capabilities like scheduling, scaling, service discovery integration, health management, and rolling updates across multiple hosts. Mesos fits that definition: it provides resource management and scheduling over a cluster and can run container workloads via supported containerizers. Kubernetes ultimately became the dominant orchestrator for many use cases, but Mesos is clearly recognized as orchestration software in this category.
So, among these choices, the verified orchestration platform is Apache Mesos (C).

NEW QUESTION: 66
What fields must exist in any Kubernetes object (e.g. YAML) file?
A. apiVersion, kind, metadata
B. kind, namespace, data
C. apiVersion, metadata, namespace
D. kind, metadata, data
Answer: (SHOW ANSWER)
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: 67
What's the difference between a security profile and a security context?
A. Security Contexts configure Clusters and Namespaces at runtime. Security profiles are control plane mechanisms to enforce specific settings in the Security Context.
B. Security Contexts configure Pods and Containers at runtime. Security profiles are control plane mechanisms to enforce specific settings in the Security Context.
C. Security Profiles configure Pods and Containers at runtime. Security Contexts are control plane mechanisms to enforce specific settings in the Security Profile.
D. Security Profiles configure Clusters and Namespaces at runtime. Security Contexts are control plane mechanisms to enforce specific settings in the Security Profile.
Answer: (SHOW ANSWER)
The correct answer is B. In Kubernetes, a securityContext is part of the Pod and container specification that configures runtime security settings for that workload-things like runAsUser, runAsNonRoot, Linux capabilities, readOnlyRootFilesystem, allowPrivilegeEscalation, SELinux options, seccomp profile selection, and filesystem group (fsGroup). These settings directly affect how the Pod's containers run on the node.
A security profile, in contrast, is a higher-level policy/standard enforced by the cluster control plane (typically via admission control) to ensure workloads meet required security constraints. In modern Kubernetes, this concept aligns with mechanisms like Pod Security Standards (Privileged, Baseline, Restricted) enforced through Pod Security Admission. The "profile" defines what is allowed or forbidden (for example, disallow privileged containers, disallow hostPath mounts, require non-root, restrict capabilities). The control plane enforces these constraints by validating or rejecting Pod specs that do not comply-ensuring consistent security posture across namespaces and teams.
Option A and D are incorrect because security contexts do not "configure clusters and namespaces at runtime"; security contexts apply to Pods/containers. Option C reverses the

relationship: security profiles don't configure Pods at runtime; they constrain what security context settings (and other fields) are acceptable.
Practically, you can think of it as:
SecurityContext = workload-level configuration knobs (declared in manifests, applied at runtime).
SecurityProfile/Standards = cluster-level guardrails that determine which knobs/settings are permitted.
This separation supports least privilege: developers declare needed runtime settings, and cluster governance ensures those settings stay within approved boundaries. Therefore, B is the verified answer.

NEW QUESTION: 68
What Linux namespace is shared by default by containers running within a Kubernetes Pod?
A. Host Network
B. Network
C. Process ID
D. Process Name
Answer: (SHOW ANSWER)
By default, containers in the same Kubernetes Pod share the network namespace, which means they share the same IP address and port space. Therefore, the correct answer is B (Network).
This shared network namespace is a key part of the Pod abstraction. Because all containers in a Pod share networking, they can communicate with each other over localhost and coordinate tightly, which is the basis for patterns like sidecars (service mesh proxies, log shippers, config reloaders). It also means containers must coordinate port usage: if two containers try to bind the same port on 0.0.0.0, they'll conflict because they share the same port namespace.
Option A ("Host Network") is different: hostNetwork: true is an optional Pod setting that puts the Pod into the node's network namespace, not the Pod's shared namespace. It is not the default and is generally used sparingly due to security and port-collision risks. Option C ("Process ID") is not shared by default in Kubernetes; PID namespace sharing requires explicitly enabling process namespace sharing (e.g., shareProcessNamespace: true). Option D ("Process Name") is not a Linux namespace concept.
The Pod model also commonly implies shared storage volumes (if defined) and shared IPC namespace in some configurations, but the universally shared-by-default namespace across containers in the same Pod is the network namespace. This default behavior is why Kubernetes documentation explains a Pod as a "logical host" for one or more containers: the containers are co-located and share certain namespaces as if they ran on the same host.
So, the correct, verified answer is B: containers in the same Pod share the Network namespace by default.

NEW QUESTION: 69
Which of the following is a responsibility of the governance board of an open source project?
A. Decide about the marketing strategy of the project.

B. Review the pull requests in the main branch.
C. Outline the project's "terms of engagement".
D. Define the license to be used in the project.
Answer: (SHOW ANSWER)
A governance board in an open source project typically defines how the community operates-its decision-making rules, roles, conflict resolution, and contribution expectations-so C ("Outline the project's terms of engagement") is correct. In large cloud-native projects (Kubernetes being a prime example), clear governance is essential to coordinate many contributors, companies, and stakeholders. Governance establishes the "rules of the road" that keep collaboration productive and fair.
"Terms of engagement" commonly includes: how maintainers are selected, how proposals are reviewed (e.g., enhancement processes), how meetings and SIGs operate, what constitutes consensus, how voting works when consensus fails, and what code-of-conduct expectations apply. It also defines escalation and dispute resolution paths so technical disagreements don't become community-breaking conflicts. In other words, governance is about ensuring the project has durable, transparent processes that outlive any individual contributor and support vendor-neutral decision making.
Option B (reviewing pull requests) is usually the responsibility of maintainers and SIG owners, not a governance board. The governance body may define the structure that empowers maintainers, but it generally does not do day-to-day code review. Option A (marketing strategy) is often handled by foundations, steering committees, or separate outreach groups, not governance boards as their primary responsibility. Option D (defining the license) is usually decided early and may be influenced by a foundation or legal process; while governance can shape legal/policy direction, the core governance responsibility is broader community operating rules rather than selecting a license.
In cloud-native ecosystems, strong governance supports sustainability: it encourages contributions, protects neutrality, and provides predictable processes for evolution. Therefore, the best verified answer is C.

NEW QUESTION: 70
What is ephemeral storage?
A. Storage space that need not persist across restarts.
B. Storage that may grow dynamically.
C. Storage used by multiple consumers (e.g., multiple Pods).
D. Storage that is always provisioned locally.
Answer: (SHOW ANSWER)
The correct answer is A: ephemeral storage is non-persistent storage whose data does not need to survive Pod restarts or rescheduling. In Kubernetes, ephemeral storage typically refers to storage tied to the Pod's lifetime-such as the container writable layer, emptyDir volumes, and other temporary storage types. When a Pod is deleted or moved to a different node, that data is generally lost.

This is different from persistent storage, which is backed by PersistentVolumes and PersistentVolumeClaims and is designed to outlive individual Pod instances. Ephemeral storage is commonly used for caches, scratch space, temporary files, and intermediate build artifacts-data that can be recreated and is not the authoritative system of record.
Option B is incorrect because "may grow dynamically" describes an allocation behavior, not the defining characteristic of ephemeral storage. Option C is incorrect because multiple consumers is about access semantics (ReadWriteMany etc.) and shared volumes, not ephemerality. Option D is incorrect because ephemeral storage is not "always provisioned locally" in a strict sense; while many ephemeral forms are local to the node, the definition is about lifecycle and persistence guarantees, not necessarily physical locality.
Operationally, ephemeral storage is an important scheduling and reliability consideration. Pods can request/limit ephemeral storage similarly to CPU/memory, and nodes can evict Pods under disk pressure. Mismanaged ephemeral storage (logs written to the container filesystem, runaway temp files) can cause node disk exhaustion and cascading failures. Best practices include shipping logs off-node, using emptyDir intentionally with size limits where supported, and using persistent volumes for state that must survive restarts.
So, ephemeral storage is best defined as storage that does not need to persist across restarts/rescheduling, matching option A.

NEW QUESTION: 71
Which of the following options includes valid API versions?
A. alpha1v1, beta3v3, v2
B. alpha1, beta3, v2
C. v1alpha1, v2beta3, v2
D. v1alpha1, v2beta3, 2.0
Answer: (SHOW ANSWER)
Kubernetes API versions follow a consistent naming pattern that indicates stability level and versioning. The valid forms include stable versions like v1, and pre-release versions such as v1alpha1, v1beta1, etc. Option C contains valid-looking Kubernetes version strings-v1alpha1, v2beta3, v2-so C is correct.
In Kubernetes, the "v" prefix is part of the standard for API versions. A stable API uses v1, v2, etc. Pre-release APIs include a stability marker: alpha (earliest, most changeable) and beta (more stable but still may change). The numeric suffix (e.g., alpha1, beta3) indicates iteration within that stability stage.
Option A is invalid because strings like alpha1v1 and beta3v3 do not match Kubernetes conventions (the v comes first, and alpha/beta are qualifiers after the version: v1alpha1). Option B is invalid because alpha1 and beta3 are missing the leading version prefix; Kubernetes API versions are not just "alpha1." Option D includes 2.0, which looks like semantic versioning but is not the Kubernetes API version format. Kubernetes uses v2, not 2.0, for API versions.
Understanding this matters because API versions signal compatibility guarantees. Stable APIs are supported for a defined deprecation window, while alpha/beta APIs may change in

incompatible ways and can be removed more easily. When authoring manifests, selecting the correct apiVersion ensures the API server accepts your resource and that controllers interpret fields correctly.
Therefore, among the choices, C is the only option comprised of valid Kubernetes-style API version strings.

NEW QUESTION: 72
In a serverless computing architecture:
A. Users of the cloud provider are charged based on the number of requests to a function.
B. Serverless functions are incompatible with containerized functions.
C. Users should make a reservation to the cloud provider based on an estimation of usage.
D. Containers serving requests are running in the background in idle status.
Answer: (SHOW ANSWER)
Serverless architectures typically bill based on actual consumption, often measured as number of requests and execution duration (and sometimes memory/CPU allocated), so A is correct. The defining trait is that you don't provision or manage servers directly; the platform scales execution up and down automatically, including down to zero for many models, and charges you for what you use.
Option B is incorrect: many serverless platforms can run container-based workloads (and some are explicitly "serverless containers"). The idea is the operational abstraction and billing model, not incompatibility with containers. Option C is incorrect because "making a reservation based on estimation" describes reserved capacity purchasing, which is the opposite of the typical serverless pay-per-use model. Option D is misleading: serverless systems aim to avoid charging for idle compute; while platforms may keep some warm capacity for latency reasons, the customer-facing model is not "containers running idle in the background." In cloud-native architecture, serverless is often chosen for spiky, event-driven workloads where you want minimal ops overhead and cost efficiency at low utilization. It pairs naturally with eventing systems (queues, pub/sub) and can be integrated with Kubernetes ecosystems via event-driven autoscaling frameworks or managed serverless offerings.
So the correct statement is A: charging is commonly based on requests (and usage), which captures the cost and operational model that differentiates serverless from always-on infrastructure.

NEW QUESTION: 73
Which Kubernetes resource uses immutable: true boolean field?
A. Deployment
B. Pod
C. ConfigMap
D. ReplicaSet
Answer: (SHOW ANSWER)

The immutable: true field is supported by ConfigMap (and also by Secrets, though Secret is not in the options), so C is correct. When a ConfigMap is marked immutable, its data can no longer be changed after creation. This is useful for protecting configuration from accidental modification and for improving cluster performance by reducing watch/update churn on frequently referenced configuration objects.
In Kubernetes, ConfigMaps store non-sensitive configuration as key-value pairs. They can be consumed by Pods as environment variables, command-line arguments, or mounted files in volumes. Without immutability, ConfigMap updates can trigger complex runtime behaviors: for example, file-mounted ConfigMap updates can eventually reflect in the volume (with some delay), but environment variables do not update automatically in running Pods. This can cause confusion and configuration drift between expected and actual behavior. Marking a ConfigMap immutable makes the configuration stable and encourages explicit rollout strategies (create a new ConfigMap with a new name and update the Pod template), which is generally more reliable for production delivery.
Why the other options are wrong: Deployments, Pods, and ReplicaSets do not use an immutable: true field as a standard top-level toggle in their API schema for the purpose described. These objects can be updated through the normal API mechanisms, and their updates are part of typical lifecycle operations (rolling updates, scaling, etc.). The immutability concept exists in Kubernetes, but the specific immutable boolean in this context is a recognized field for ConfigMap (and Secret) objects.
Operationally, immutable ConfigMaps help enforce safer practices: instead of editing live configuration in place, teams adopt versioned configuration artifacts and controlled rollouts via Deployments. This fits cloud-native principles of repeatability and reducing accidental production changes.

NEW QUESTION: 74
The cloud native architecture centered around microservices provides a strong system that ensures _____.
A. fallback
B. resiliency
C. failover
D. high reachability
Answer: (SHOW ANSWER)
The best answer is B (resiliency). A microservices-centered cloud-native architecture is designed to build systems that continue to operate effectively under change and failure. "Resiliency" is the umbrella concept: the ability to tolerate faults, recover from disruptions, and maintain acceptable service levels through redundancy, isolation, and automated recovery.
Microservices help resiliency by reducing blast radius. Instead of one monolith where a single defect can take down the entire application, microservices separate concerns into independently deployable components. Combined with Kubernetes, you get resiliency mechanisms such as replication (multiple Pod replicas), self-healing (restart and reschedule on failure), rolling updates,

health probes, and service discovery/load balancing. These enable the platform to detect and replace failing instances automatically, and to keep traffic flowing to healthy backends.
Options C (failover) and A (fallback) are resiliency techniques but are narrower terms. Failover usually refers to switching to a standby component when a primary fails; fallback often refers to degraded behavior (cached responses, reduced features). Both can exist in microservice systems, but the broader architectural guarantee microservices aim to support is resiliency overall. Option D ("high reachability") is not the standard term used in cloud-native design and doesn't capture the intent as precisely as resiliency.
In practice, achieving resiliency also requires good observability and disciplined delivery: monitoring/alerts, tracing across service boundaries, circuit breakers/timeouts/retries, and progressive delivery patterns. Kubernetes provides platform primitives, but resilient microservices also need careful API design and failure-mode thinking.
So the intended and verified completion is resiliency, option B.

NEW QUESTION: 75
What is the API that exposes resource metrics from the metrics-server?
A. custom.k8s.io
B. resources.k8s.io
C. metrics.k8s.io
D. cadvisor.k8s.io
Answer: (SHOW ANSWER)
The correct answer is C: metrics.k8s.io. Kubernetes' metrics-server is the standard component that provides resource metrics (primarily CPU and memory) for nodes and pods. It aggregates this information (sourced from kubelet/cAdvisor) and serves it through the Kubernetes aggregated API under the group metrics.k8s.io. This is what enables commands like kubectl top nodes and kubectl top pods, and it is also a key data source for autoscaling with the Horizontal Pod Autoscaler (HPA) when scaling on CPU/memory utilization.
Why the other options are wrong:
custom.k8s.io is not the standard API group for metrics-server resource metrics. Custom metrics are typically served through the custom metrics API (commonly custom.metrics.k8s.io) via adapters (e.g., Prometheus Adapter), not metrics-server.
resources.k8s.io is not the metrics-server API group.
cadvisor.k8s.io is not exposed as a Kubernetes aggregated metrics API. cAdvisor is a component integrated into kubelet that provides container stats, but metrics-server is the thing that exposes the aggregated Kubernetes metrics API, and the canonical group is metrics.k8s.io.
Operationally, it's important to understand the boundary: metrics-server provides basic resource metrics suitable for core autoscaling and "top" views, but it is not a full observability system (it does not store long-term metrics history like Prometheus). For richer metrics (SLOs, application metrics, long-term trending), teams typically deploy Prometheus or a managed monitoring backend. Still, when the question asks specifically which API exposes metrics-server data, the answer is definitively metrics.k8s.io.

NEW QUESTION: 76
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: (SHOW ANSWER)
The true statement is C: the default namespace exists when a new cluster is created.
Namespaces are a Kubernetes mechanism for partitioning cluster resources into logical groups. When you set up a cluster, Kubernetes creates some initial namespaces (including default, and commonly kube-system, kube-public, and kube-node-lease). The default namespace is where resources go if you don't specify a namespace explicitly.
Option A is false because namespaces are not hierarchical; Kubernetes does not support "namespaces inside namespaces." Option B is false because within a given namespace, resource names must be unique per resource kind. You can't have two Deployments with the same name in the same namespace. You can have a Deployment named web in one namespace and another Deployment named web in a different namespace-namespaces provide that scope boundary. Option D is false because not all objects are namespaced. Many resources are cluster-scoped (for example, Nodes, PersistentVolumes, ClusterRoles, ClusterRoleBindings, and StorageClasses). Namespaces apply only to namespaced resources.
Operationally, namespaces support multi-tenancy and environment separation (dev/test/prod), RBAC scoping, resource quotas, and policy boundaries. For example, you can grant a team access only to their namespace and enforce quotas that prevent them from consuming excessive CPU/memory. Namespaces also make organization and cleanup easier: deleting a namespace removes most namespaced resources inside it (subject to finalizers).
So, the verified correct statement is C: the default namespace exists upon cluster creation.

NEW QUESTION: 77
Which control plane component is responsible for updating the node Ready condition if a node becomes unreachable?
A. The kube-proxy

B. The node controller
C. The kubectl
D. The kube-apiserver
Answer: (SHOW ANSWER)
The correct answer is B: the node controller. In Kubernetes, node health is monitored and reflected through Node conditions such as Ready. The Node Controller (a controller that runs as part of the control plane, within the controller-manager) is responsible for monitoring node heartbeats and updating node status when a node becomes unreachable or unhealthy.
Nodes periodically report status (including kubelet heartbeats) to the API server. The Node Controller watches these updates. If it detects that a node has stopped reporting within expected time windows, it marks the node condition Ready as Unknown (or otherwise updates conditions) to indicate the control plane can't confirm node health. This status change then influences higher-level behaviors such as Pod eviction and rescheduling: after grace periods and eviction timeouts, Pods on an unhealthy node may be evicted so the workload can be recreated on healthy nodes (assuming a controller manages replicas).
Option A (kube-proxy) is a node component for Service traffic routing and does not manage node health conditions. Option C (kubectl) is a CLI client; it does not participate in control plane health monitoring. Option D (kube-apiserver) stores and serves Node status, but it doesn't decide when a node is unreachable; it persists what controllers and kubelets report. The "decision logic" for updating the Ready condition in response to missing heartbeats is the Node Controller's job.
So, the component that updates the Node Ready condition when a node becomes unreachable is the node controller, which is option B.

NEW QUESTION: 78
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: (SHOW ANSWER)
Helm is best described as a package manager for Kubernetes applications, making B correct. Helm packages Kubernetes resource manifests (Deployments, Services, ConfigMaps, Ingress, RBAC, etc.) into a unit called a chart. A chart includes templates and default values, allowing teams to parameterize deployments for different environments (dev/stage/prod) without rewriting YAML.
From an application delivery perspective, Helm solves common problems: repeatable installation, upgrade management, versioning, and sharing of standardized application definitions. Instead of copying and editing raw YAML, users install a chart and supply a values.yaml file (or CLI overrides) to configure image tags, replica counts, ingress hosts, resource requests, and other settings. Helm then renders templates into concrete Kubernetes manifests and applies them to the cluster.

Helm also manages releases: it tracks what has been installed and supports upgrades and rollbacks. This aligns with cloud native delivery practices where deployments are automated, reproducible, and auditable. Helm is commonly integrated into CI/CD pipelines and GitOps workflows (sometimes with charts stored in Git or Helm repositories).
The other options are incorrect: a dashboard is a UI like Kubernetes Dashboard; a scheduler is kube-scheduler (or custom scheduler implementations, but Helm is not that); end-to-end testing projects exist in the ecosystem, but Helm's role is packaging and lifecycle management of Kubernetes app definitions.
So the verified, standard definition is: Helm = Kubernetes package manager.

NEW QUESTION: 79
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: (SHOW ANSWER)
B is correct: the OCI's main purpose is to create open, vendor-neutral industry standards for container image formats and container runtimes. Standardization is critical in container orchestration because portability is a core promise: you should be able to build an image once and run it across different environments and runtimes without rewriting packaging or execution logic.
OCI defines (at a high level) two foundational specs:
Image specification: how container images are packaged (layers, metadata, manifests).
Runtime specification: how to run a container (filesystem setup, namespaces/cgroups behavior, lifecycle).
These standards enable interoperability across tooling. For example, higher-level runtimes (like containerd or CRI-O) rely on OCI-compliant components (often runc or equivalents) to execute containers consistently.
Why the other options are not the best answer:
A (accelerating adoption) might be an indirect outcome, but it's not the OCI's core charter.
C is contradictory ("industry standards" but "for private purposes")-OCI is explicitly about open standards.
D (improving security) can be helped by standardization and best practices, but OCI is not primarily a security standards body; its central function is format and runtime interoperability.
In Kubernetes specifically, OCI is part of the "plumbing" that makes runtimes replaceable. Kubernetes talks to runtimes via CRI; runtimes execute containers via OCI. This layering helps Kubernetes remain runtime-agnostic while still benefiting from consistent container behavior everywhere.

Therefore, the correct choice is B: OCI creates open standards around container formats and runtimes.

NEW QUESTION: 80

In a cloud native world, what does the IaC abbreviation stand for?

A. Infrastructure and Code

B. Infrastructure as Code

C. Infrastructure above Code

D. Infrastructure across Code

Answer: (SHOW ANSWER)

IaC stands for Infrastructure as Code, which is option B. In cloud native environments, IaC is a core operational practice: infrastructure (networks, clusters, load balancers, IAM roles, storage classes, DNS records, and more) is defined using code-like, declarative configuration rather than manual, click-driven changes. This approach mirrors Kubernetes' own declarative model-where you define desired state in manifests and controllers reconcile the cluster to match.

IaC improves reliability and velocity because it makes infrastructure repeatable, version-controlled, reviewable, and testable. Teams can store infrastructure definitions in Git, use pull requests for change review, and run automated checks to validate formatting, policies, and safety constraints. If an environment must be recreated (disaster recovery, test environments, regional expansion), IaC enables consistent reproduction with fewer human errors.

In Kubernetes-centric workflows, IaC often covers both the base platform and the workloads layered on top. For example, provisioning might include the Kubernetes control plane, node pools, networking, and identity integration, while Kubernetes manifests (or Helm/Kustomize) define Deployments, Services, RBAC, Ingress, and storage resources. GitOps extends this further by continuously reconciling cluster configuration from a Git source of truth.

The incorrect options (Infrastructure and Code / above / across) are not standard terms. The key idea is "infrastructure treated like software": changes are made through code commits, go through CI checks, and are rolled out in controlled ways. This aligns with cloud native goals: faster iteration, safer operations, and easier auditing. In short, IaC is the operational backbone that makes Kubernetes and cloud platforms manageable at scale, enabling consistent environments and reducing configuration drift.

You're right - my previous 16-30 were not taken from your PDF. Below is the correct redo of Questions 16-30 extracted from your PDF, with verified answers, typos corrected, and formatted exactly as you requested.

NEW QUESTION: 81

How does dynamic storage provisioning work?

A. A user requests dynamically provisioned storage by including an existing StorageClass in their PersistentVolumeClaim.

B. An administrator creates a StorageClass and includes it in their Pod YAML definition file without creating a PersistentVolumeClaim.
C. A Pod requests dynamically provisioned storage by including a StorageClass and the Pod name in their PersistentVolumeClaim.
D. An administrator creates a PersistentVolume and includes the name of the PersistentVolume in their Pod YAML definition file.
Answer: (SHOW ANSWER)
Dynamic provisioning is the Kubernetes mechanism where storage is created on-demand when a user creates a PersistentVolumeClaim (PVC) that references a StorageClass, so A is correct. In this model, the user does not need to pre-create a PersistentVolume (PV). Instead, the StorageClass points to a provisioner (typically a CSI driver) that knows how to create a volume in the underlying storage system (cloud disk, SAN, NAS, etc.). When the PVC is created with storageClassName: <class>, Kubernetes triggers the provisioner to create a new volume and then binds the resulting PV to that PVC.
This is why option B is incorrect: you do not put a StorageClass "in the Pod YAML" to request provisioning. Pods reference PVCs, not StorageClasses directly. Option C is incorrect because the PVC does not need the Pod name; binding is done via the PVC itself. Option D describes static provisioning: an admin pre-creates PVs and users claim them by creating PVCs that match the PV (capacity, access modes, selectors). Static provisioning can work, but it is not dynamic provisioning.
Under the hood, the StorageClass can define parameters like volume type, replication, encryption, and binding behavior (e.g., volumeBindingMode: WaitForFirstConsumer to delay provisioning until the Pod is scheduled, ensuring the volume is created in the correct zone). Reclaim policies (Delete/Retain) define what happens to the underlying volume after the PVC is deleted.
In cloud-native operations, dynamic provisioning is preferred because it improves developer self-service, reduces manual admin work, and makes scaling stateful workloads easier and faster. The essence is: PVC + StorageClass → automatic PV creation and binding.

NEW QUESTION: 82
What best describes cloud native service discovery?
A. It's a mechanism for applications and microservices to locate each other on a network.
B. It's a procedure for discovering a MAC address, associated with a given IP address.
C. It's used for automatically assigning IP addresses to devices connected to the network.
D. It's a protocol that turns human-readable domain names into IP addresses on the Internet.
Answer: (SHOW ANSWER)
Cloud native service discovery is fundamentally about how services and microservices find and connect to each other reliably in a dynamic environment, so A is correct. In cloud native systems (especially Kubernetes), instances are ephemeral: Pods can be created, destroyed, rescheduled, and scaled at any time. Hardcoding IPs breaks quickly. Service discovery provides stable names

and lookup mechanisms so that one component can locate another even as underlying endpoints change.
In Kubernetes, service discovery is commonly achieved through Services (stable virtual IP + DNS name) and cluster DNS (CoreDNS). A Service selects a group of Pods via labels, and Kubernetes maintains the set of endpoints behind that Service. Clients connect to the Service name (DNS) and Kubernetes routes traffic to the current healthy Pods. For some workloads, headless Services provide DNS records that map directly to Pod IPs for per-instance discovery.
The other options describe different networking concepts: B is ARP (MAC discovery), C is DHCP (IP assignment), and D is DNS in a general internet sense. DNS is often used as a mechanism for service discovery, but cloud native service discovery is broader: it's the overall mechanism enabling dynamic location of services, often implemented via DNS and/or environment variables and sometimes enhanced by service meshes.
So the best description remains A: a mechanism that allows applications and microservices to locate each other on a network in a dynamic environment.

NEW QUESTION: 83
Which kubectl command is useful for collecting information about any type of resource that is active in a Kubernetes cluster?
A. describe
B. list
C. expose
D. explain
Answer: (SHOW ANSWER)
The correct answer is A (describe), used as kubectl describe <resource> <name>. kubectl describe is a troubleshooting-focused command that provides a rich, human-readable view of a specific live object in the cluster, including key fields, status, and-crucially-Events related to that object. This makes it extremely useful for "collecting information" about almost any active resource: Pods, Deployments, Nodes, Services, PersistentVolumeClaims, and more.
kubectl get (not listed) is typically used for listing objects and their summary fields, but kubectl describe goes deeper: for a Pod it will show container images, resource requests/limits, probes, mounted volumes, node assignment, IPs, conditions, and recent scheduling/pulling/starting events. For a Node it shows capacity/allocatable resources, labels/taints, conditions, and node events. Those event details often explain why something is Pending, failing to pull images, failing readiness checks, or being evicted.
Option B ("list") is not a standard kubectl subcommand for retrieving resource information (you would use get for listing). Option C (expose) is for creating a Service to expose a resource (like a Deployment). Option D (explain) is for viewing API schema/field documentation (e.g., kubectl explain deployment.spec.replicas) and does not report what is currently happening in the cluster.
So, for gathering detailed live diagnostics about a resource in the cluster, the best kubectl command is kubectl describe, which corresponds to option A.

NEW QUESTION: 84
What is a best practice to minimize the container image size?
A. Use a DockerFile.
B. Use multistage builds.
C. Build images with different tags.
D. Add a build.sh script.
Answer: (SHOW ANSWER)
A proven best practice for minimizing container image size is to use multi-stage builds, so B is correct. Multi-stage builds allow you to separate the "build environment" from the "runtime environment." In the first stage, you can use a full-featured base image (with compilers, package managers, and build tools) to compile your application or assemble artifacts. In the final stage, you copy only the resulting binaries or necessary runtime assets into a much smaller base image (for example, a distroless image or a slim OS image). This dramatically reduces the final image size because it excludes compilers, caches, and build dependencies that are not needed at runtime.
In cloud-native application delivery, smaller images matter for several reasons. They pull faster, which speeds up deployments, rollouts, and scaling events (Pods become Ready sooner). They also reduce attack surface by removing unnecessary packages, which helps security posture and scanning results. Smaller images tend to be simpler and more reproducible, improving reliability across environments.
Option A is not a size-minimization practice: using a Dockerfile is simply the standard way to define how to build an image; it doesn't inherently reduce size. Option C (different tags) changes image identification but not size. Option D (a build script) may help automation, but it doesn't guarantee smaller images; the image contents are determined by what ends up in the layers.
Multi-stage builds are commonly paired with other best practices: choosing minimal base images, cleaning package caches, avoiding copying unnecessary files (use .dockerignore), and reducing layer churn. But among the options, the clearest and most directly correct technique is multi-stage builds.
Therefore, the verified answer is B.

NEW QUESTION: 85
What does "Continuous Integration" mean?
A. The continuous integration and testing of code changes from multiple sources manually.
B. The continuous integration and testing of code changes from multiple sources via automation.
C. The continuous integration of changes from one environment to another.
D. The continuous integration of new tools to support developers in a project.
Answer: (SHOW ANSWER)
The correct answer is B: Continuous Integration (CI) is the practice of frequently integrating code changes from multiple contributors and validating them through automated builds and tests. The "continuous" part is about doing this often (ideally many times per day) and consistently, so

integration problems are detected early instead of piling up until a painful merge or release window.
Automation is essential. CI typically includes steps like compiling/building artifacts, running unit and integration tests, executing linters, checking formatting, scanning dependencies for vulnerabilities, and producing build reports. This automation creates fast feedback loops that help developers catch regressions quickly and maintain a releasable main branch.
Option A is incorrect because manual integration/testing does not scale and undermines the reliability and speed that CI is meant to provide. Option C confuses CI with deployment promotion across environments (which is more aligned with Continuous Delivery/Deployment). Option D is unrelated: adding tools can support CI, but it isn't the definition.
In cloud-native application delivery, CI is tightly coupled with containerization and Kubernetes: CI pipelines often build container images from source, run tests, scan images, sign artifacts, and push to registries. Those validated artifacts then flow into CD processes that deploy to Kubernetes using manifests, Helm, or GitOps controllers. Without CI, Kubernetes rollouts become riskier because you lack consistent validation of what you're deploying.
So, CI is best defined as automated integration and testing of code changes from multiple sources, which matches option B.

NEW QUESTION: 86
Kubernetes ___ protect you against voluntary interruptions (such as deleting Pods, draining nodes) to run applications in a highly available manner.
A. Pod Topology Spread Constraints
B. Pod Disruption Budgets
C. Taints and Tolerations
D. Resource Limits and Requests
Answer: (SHOW ANSWER)
The correct answer is B: Pod Disruption Budgets (PDBs). A PDB is a policy object that limits how many Pods of an application can be voluntarily disrupted at the same time. "Voluntary disruptions" include actions such as draining a node for maintenance (kubectl drain), cluster upgrades, or an administrator deleting Pods. The core purpose is to preserve availability by ensuring that a minimum number (or percentage) of replicas remain running and ready while those planned disruptions occur.
A PDB is typically defined with either minAvailable (e.g., "at least 3 Pods must remain available") or maxUnavailable (e.g., "no more than 1 Pod can be unavailable"). Kubernetes uses this budget when performing eviction operations. If evicting a Pod would violate the PDB, the eviction is blocked (or delayed), which forces maintenance workflows to proceed more safely-either by draining more slowly, scaling up first, or scheduling maintenance in stages.
Why the other options are not correct: topology spread constraints (A) influence scheduling distribution across failure domains but don't directly protect against voluntary disruptions. Taints and tolerations (C) control where Pods can schedule, not how many can be disrupted. Resource

requests/limits (D) control CPU/memory allocation and do not guard availability during drains or deletions.
PDBs also work best when paired with Deployments/StatefulSets that maintain replicas and with readiness probes that accurately represent whether a Pod can serve traffic. PDBs do not prevent involuntary disruptions (node crashes), but they materially reduce risk during planned operations-exactly what the question is targeting.

NEW QUESTION: 87
Which is an industry-standard container runtime with an "emphasis" on simplicity, robustness, and portability?
A. CRI-O
B. LXD
C. containerd
D. kata-runtime
Answer: (SHOW ANSWER)
containerd is a widely adopted, industry-standard container runtime known for simplicity, robustness, and portability, so C is correct. containerd originated as a core component extracted from Docker and has become a common runtime across Kubernetes distributions and managed services. It implements container lifecycle management (image pull, unpack, container execution, snapshotting) and typically delegates low-level container execution to an OCI runtime like runc.
In Kubernetes, kubelet communicates with container runtimes through CRI. containerd provides a CRI plugin (or can be integrated via CRI implementations) that makes it a first-class choice for Kubernetes nodes. This aligns with the runtime landscape after dockershim removal: Kubernetes users commonly run containerd or CRI-O as the node runtime.
Option A (CRI-O) is also a CRI-focused runtime and is valid in Kubernetes contexts, but the phrasing "industry-standard ... emphasis on simplicity, robustness, and portability" is strongly associated with containerd's positioning and broad cross-platform adoption beyond Kubernetes.
Option B (LXD) is a system container manager (often associated with LXC) and not the standard Kubernetes runtime in mainstream CRI discussions. Option D (kata-runtime) is associated with Kata Containers, which focuses on stronger isolation by running containers inside lightweight VMs; that is a security-oriented sandbox approach rather than a simplicity/portability "industry standard" baseline runtime.
From a cloud-native operations point of view, containerd's popularity comes from its stable API, strong ecosystem support, and alignment with OCI standards. It integrates cleanly with image registries, supports modern snapshotters, and is heavily used in production by many Kubernetes providers. Therefore, the best verified answer is C: containerd.

NEW QUESTION: 88
How many different Kubernetes service types can you define?
A. 2
B. 3

C. 4
D. 5
Answer: (SHOW ANSWER)
Kubernetes defines four primary Service types, which is why C (4) is correct. The commonly recognized Service spec.type values are:
ClusterIP: The default type. Exposes the Service on an internal virtual IP reachable only within the cluster. This supports typical east-west traffic between workloads.
NodePort: Exposes the Service on a static port on each node. Traffic to <NodeIP>:<NodePort> is forwarded to the Service endpoints. This is often used for simple external access in environments without load balancers, or as a building block for other systems.
LoadBalancer: Integrates with a cloud provider (or load balancer implementation) to provision an external load balancer and route traffic to the Service. This is common in managed Kubernetes.
ExternalName: Maps the Service name to an external DNS name via a CNAME record, allowing in-cluster clients to use a consistent Service DNS name to reach an external dependency.
Some people also talk about "Headless Services," but headless is not a separate type; it's a behavior achieved by setting clusterIP: None. Headless Services still use the Service API object but change DNS and virtual-IP behavior to return endpoint IPs directly rather than a ClusterIP. That's why the canonical count of "Service types" is four.
This question tests understanding of the Service abstraction: Service type controls how a stable service identity is exposed (internal VIP, node port, external LB, or DNS alias), while selectors/endpoints control where traffic goes (the backend Pods). Different environments will favor different types: ClusterIP for internal microservices, LoadBalancer for external exposure in cloud, NodePort for bare-metal or simple access, ExternalName for bridging to outside services. Therefore, the verified answer is C (4).

NEW QUESTION: 89
If kubectl is failing to retrieve information from the cluster, where can you find Pod logs to troubleshoot?
A. /var/log/pods/
B. ~/.kube/config
C. /var/log/k8s/
D. /etc/kubernetes/
Answer: (SHOW ANSWER)
The correct answer is A: /var/log/pods/. When kubectl logs can't retrieve logs (for example, API connectivity issues, auth problems, or kubelet/API proxy issues), you can often troubleshoot directly on the node where the Pod ran. Kubernetes nodes typically store container logs on disk, and a common location is under /var/log/pods/, organized by namespace, Pod name/UID, and container. This directory contains symlinks or files that map to the underlying container runtime log location (often under /var/log/containers/ as well, depending on distro/runtime setup).
Option B (~/.kube/config) is your local kubeconfig file; it contains cluster endpoints and credentials, not Pod logs. Option D (/etc/kubernetes/) contains Kubernetes component

configuration/manifests on some installations (especially control plane), not application logs. Option C (/var/log/k8s/) is not a standard Kubernetes log path.
Operationally, the node-level log locations depend on the container runtime and logging configuration, but the Kubernetes convention is that kubelet writes container logs to a known location and exposes them through the API so kubectl logs works. If the API path is broken, node access becomes your fallback. This is also why secure node access is sensitive: anyone with node root access can potentially read logs (and other data), which is part of the threat model.
So, the best answer for where to look on the node for Pod logs when kubectl can't retrieve them is /var/log/pods/, option A.

NEW QUESTION: 90

What happens with a regular Pod running in Kubernetes when a node fails?

A. A new Pod with the same UID is scheduled to another node after a while.
B. A new, near-identical Pod but with different UID is scheduled to another node.
C. By default, a Pod can only be scheduled to the same node when the node fails.
D. A new Pod is scheduled on a different node only if it is configured explicitly.
Answer: (SHOW ANSWER)
B is correct: when a node fails, Kubernetes does not "move" the same Pod instance; instead, a new Pod object (new UID) is created to replace it-assuming the Pod is managed by a controller (Deployment/ReplicaSet, StatefulSet, etc.). A Pod is an API object with a unique identifier (UID) and is tightly associated with the node it's scheduled to via spec.nodeName. If the node becomes unreachable, that original Pod cannot be restarted elsewhere because it was bound to that node.
Kubernetes' high availability comes from controllers maintaining desired state. For example, a Deployment desires N replicas. If a node fails and the replicas on that node are lost, the controller will create replacement Pods, and the scheduler will place them onto healthy nodes. These replacement Pods will be "near-identical" in spec (same template), but they are still new instances with new UIDs and typically new IPs.
Why the other options are wrong:
A is incorrect because the UID does not remain the same-Kubernetes creates a new Pod object rather than reusing the old identity.
C is incorrect; pods are not restricted to the same node after failure. The whole point of orchestration is to reschedule elsewhere.
D is incorrect; rescheduling does not require special explicit configuration for typical controller-managed workloads. The controller behavior is standard. (If it's a bare Pod without a controller, it will not be recreated automatically.) This also ties to the difference between "regular Pod" vs controller-managed workloads: a standalone Pod is not self-healing by itself, while a Deployment/ReplicaSet provides that resilience. In typical production design, you run workloads under controllers specifically so node failure triggers replacement and restores replica count.
Therefore, the correct outcome is B.

NEW QUESTION: 91

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: (SHOW ANSWER)
The Kubernetes-native mechanism to run setup steps before the main application containers start is an init container, so A is correct. Init containers run sequentially and must complete successfully before the regular containers in the Pod are started. This makes them ideal for preparing configuration, downloading artifacts, performing migrations, generating files, or waiting for dependencies.
The question specifically asks how to "load and generate data required before Pod startup." The most common pattern is: an init container writes files into a shared volume (like an emptyDir volume) mounted by both the init container and the app container. When the init container finishes, the app container starts and reads the generated files. This is deterministic and aligns with Kubernetes Pod lifecycle semantics.
A sidecar container (option C) runs concurrently with the main container, so it is not guaranteed to complete work before startup. Sidecars are great for ongoing concerns (log shipping, proxies, config reloaders), but they are not the primary "before startup" mechanism. A PVC volume (option B) is just storage; it doesn't itself perform generation or ensure ordering. "Another Pod with a PVC" (option D) introduces coordination complexity and still does not guarantee the data is prepared before this Pod starts unless you build additional synchronization.
Init containers are explicitly designed for this kind of pre-flight work, and Kubernetes guarantees ordering: all init containers complete in order, then the app containers begin. That guarantee is why A is the best and verified answer.

NEW QUESTION: 92
What is the role of a NetworkPolicy in Kubernetes?
A. The ability to cryptic and obscure all traffic.
B. The ability to classify the Pods as isolated and non isolated.
C. The ability to prevent loopback or incoming host traffic.

D. The ability to log network security events.

Answer: (SHOW ANSWER)

A Kubernetes NetworkPolicy defines which traffic is allowed to and from Pods by selecting Pods and specifying ingress/egress rules. A key conceptual effect is that it can make Pods "isolated" (default deny except what is allowed) versus "non-isolated" (default allow). This aligns best with option B, so B is correct.

By default, Kubernetes networking is permissive: Pods can typically talk to any other Pod. When you apply a NetworkPolicy that selects a set of Pods, those selected Pods become "isolated" for the direction(s) covered by the policy (ingress and/or egress). That means only traffic explicitly allowed by the policy is permitted; everything else is denied (again, for the selected Pods and direction). This classification concept-isolated vs non-isolated-is a common way the Kubernetes documentation explains NetworkPolicy behavior.

Option A is incorrect: NetworkPolicy does not encrypt ("cryptic and obscure") traffic. Encryption is typically handled by mTLS via a service mesh or application-layer TLS. Option C is not the primary role; loopback and host traffic handling depend on the network plugin and node configuration, and NetworkPolicy is not a "prevent loopback" mechanism. Option D is incorrect because NetworkPolicy is not a logging system; while some CNIs can produce logs about policy decisions, logging is not NetworkPolicy's role in the API.

One critical Kubernetes detail: NetworkPolicy enforcement is performed by the CNI/network plugin. If your CNI doesn't implement NetworkPolicy, creating these objects won't change runtime traffic. In CNIs that do support it, NetworkPolicy becomes a foundational security primitive for segmentation and least privilege: restricting database access to app Pods only, isolating namespaces, and reducing lateral movement risk.

So, in the language of the provided answers, NetworkPolicy's role is best captured as the ability to classify Pods into isolated/non-isolated by applying traffic-allow rules-option B.

NEW QUESTION: 93

Which of the following resources helps in managing a stateless application workload on a Kubernetes cluster?

A. DaemonSet

B. StatefulSet

C. kubectl

D. Deployment

Answer: (SHOW ANSWER)

The correct answer is D: Deployment. A Deployment is the standard Kubernetes controller for managing stateless applications. It provides declarative updates, replica management, and rollout/rollback functionality. You define the desired state (container image, environment variables, ports, replica count) in the Deployment spec, and Kubernetes ensures the specified number of Pods are running and updated according to strategy (RollingUpdate by default). Stateless workloads are ideal for Deployments because each replica is interchangeable. If a Pod dies, a new one can be created anywhere; if traffic increases, replicas can be increased; if you

need to update the app, a new ReplicaSet is created and traffic shifts gradually to new Pods. Deployments integrate naturally with Services for stable networking and load balancing.
Why the other options are incorrect:
A DaemonSet ensures one Pod per node (or selected nodes). It's for node-level agents, not generic stateless service replicas.
A StatefulSet is for workloads needing stable identity, ordered rollout, and persistent storage per replica (databases, quorum systems). That's not the typical stateless app case.
kubectl is a CLI tool; it doesn't "manage" workloads as a controller resource.
In real cluster operations, almost every stateless microservice is represented as a Deployment plus a Service (and often an Ingress/Gateway for edge routing). Deployments also support advanced delivery patterns (maxSurge/maxUnavailable tuning) and easy integration with HPA for horizontal scaling. Because the question is specifically "managing a stateless application workload," the Kubernetes resource designed for that is clearly the Deployment.

NEW QUESTION: 94
Which authorization-mode allows granular control over the operations that different entities can perform on different objects in a Kubernetes cluster?
A. Webhook Mode Authorization Control
B. Role Based Access Control
C. Node Authorization Access Control
D. Attribute Based Access Control
Answer: B (LEAVE A REPLY)
Role Based Access Control (RBAC) is the standard Kubernetes authorization mode that provides granular control over what users and service accounts can do to which resources, so B is correct. RBAC works by defining Roles (namespaced) and ClusterRoles (cluster-wide) that contain sets of rules. Each rule specifies API groups, resource types, resource names (optional), and allowed verbs such as get, list, watch, create, update, patch, and delete. You then attach these roles to identities using RoleBindings or ClusterRoleBindings.
This gives fine-grained, auditable access control. For example, you can allow a CI service account to create and patch Deployments only in a specific namespace, while restricting it from reading Secrets. You can allow developers to view Pods and logs but prevent them from changing cluster-wide networking resources. This is exactly the "granular control over operations on objects" described by the question.
Why other options are not the best answer: "Webhook mode" is an authorization mechanism where Kubernetes calls an external service to decide authorization. While it can be granular depending on the external system, Kubernetes' common built-in answer for granular object-level control is RBAC. "Node authorization" is a specialized authorizer for kubelets/nodes to access resources they need; it's not the general-purpose system for all cluster entities. ABAC (Attribute-Based Access Control) is an older mechanism and is not the primary recommended authorization model; it can be expressive but is less commonly used and not the default best-practice for Kubernetes authorization today.

In Kubernetes security practice, RBAC is typically paired with authentication (certs/OIDC), admission controls, and namespaces to build a defense-in-depth security posture. RBAC policy is also central to least privilege: granting only what is necessary for a workload or user role to function. This reduces blast radius if credentials are compromised.
Therefore, the verified answer is B: Role Based Access Control.

NEW QUESTION: 95
Which of the following is a correct definition of a Helm chart?
A. A Helm chart is a collection of YAML files bundled in a tar.gz file and can be applied without decompressing it.
B. A Helm chart is a collection of JSON files and contains all the resource definitions to run an application on Kubernetes.
C. A Helm chart is a collection of YAML files that can be applied on Kubernetes by using the kubectl tool.
D. A Helm chart is similar to a package and contains all the resource definitions to run an application on Kubernetes.
Answer: (SHOW ANSWER)
A Helm chart is best described as a package for Kubernetes applications, containing the resource definitions (as templates) and metadata needed to install and manage an application-so D is correct. Helm is a package manager for Kubernetes; the chart is the packaging format. Charts include a Chart.yaml (metadata), a values.yaml (default configuration values), and a templates/ directory containing Kubernetes manifests written as templates. When you install a chart, Helm renders those templates into concrete Kubernetes YAML manifests by substituting values, then applies them to the cluster.
Option A is misleading/incomplete. While charts are often distributed as a compressed tarball (.tgz), the defining feature is not "YAML bundled in tar.gz" but the packaging and templating model that supports install/upgrade/rollback. Option B is incorrect because Helm charts are not "collections of JSON files" by definition; Kubernetes resources can be expressed as YAML or JSON, but Helm charts overwhelmingly use templated YAML. Option C is incorrect because charts are not simply YAML applied by kubectl; Helm manages releases, tracks installed resources, and supports upgrades and rollbacks. Helm uses Kubernetes APIs under the hood, but the value of Helm is the lifecycle and packaging system, not "kubectl apply." In cloud-native application delivery, Helm helps standardize deployments across environments (dev/stage/prod) by externalizing configuration through values. It reduces copy/paste and supports reuse via dependencies and subcharts. Helm also supports versioning of application packages, allowing teams to upgrade predictably and roll back if needed-critical for production change management.
So, the correct and verified definition is D: a Helm chart is like a package containing the resource definitions needed to run an application on Kubernetes.

NEW QUESTION: 96
What framework does Kubernetes use to authenticate users with JSON Web Tokens?

A. OpenID Connect
B. OpenID Container
C. OpenID Cluster
D. OpenID CNCF
Answer: (SHOW ANSWER)
Kubernetes commonly authenticates users using OpenID Connect (OIDC) when JSON Web Tokens (JWTs) are involved, so A is correct. OIDC is an identity layer on top of OAuth 2.0 that standardizes how clients obtain identity information and how JWTs are issued and validated.
In Kubernetes, authentication happens at the API server. When OIDC is configured, the API server validates incoming bearer tokens (JWTs) by checking token signature and claims against the configured OIDC issuer and client settings. Kubernetes can use OIDC claims (such as sub, email, groups) to map the authenticated identity to Kubernetes RBAC subjects. This is how enterprises integrate clusters with identity providers such as Okta, Dex, Azure AD, or other OIDC-compliant IdPs.
Options B, C, and D are fabricated phrases and not real frameworks. Kubernetes documentation explicitly references OIDC as a supported method for token-based user authentication (alongside client certificates, bearer tokens, static token files, and webhook authentication). The key point is that Kubernetes does not "invent" JWT auth; it integrates with standard identity providers through OIDC so clusters can participate in centralized SSO and group-based authorization.
Operationally, OIDC authentication is typically paired with:
RBAC for authorization ("what you can do")
Audit logging for traceability
Short-lived tokens and rotation practices for security
Group claim mapping to simplify permission management
So, the verified framework Kubernetes uses with JWTs for user authentication is OpenID Connect.