DUMPSBOSS.

Certified Cloud Native Platform Engineering Associate

Linux Foundation CNPA

Version Demo

Total Demo Questions: 10

Total Premium Questions: 106

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

QUESTION NO: 1

In a cloud native environment, how do policy engines facilitate a unified approach for teams to consume platform services?

A. Enforces strict compliance policies with security standards.

B. Integrates with CI/CD pipelines to streamline service provisioning.

C. Enforces service-level agreements (SLAs) across all teams.

D. Provides centralized reusable policies to ensure security and compliance.

ANSWER: D

Explanation:

Provides centralized reusable policies to ensure security and compliance is correct because policy engines give platform teams a common mechanism for defining and applying governance guardrails wherever platform services are consumed. Rather than requiring every application team to interpret and implement requirements independently, teams can consume approved platform capabilities through consistent rules that are maintained as policy. This establishes a unified operating model while preserving self-service: developers can deploy and use services within clearly defined boundaries, and platform teams can update those boundaries centrally as organizational, regulatory, or security needs evolve.

In Kubernetes-based platforms, policy engines can evaluate resource requests against declarative rules during admission and, depending on the tool and configuration, validate, mutate, or generate resources. These reusable rules can standardize requirements such as permitted container registries, workload security settings, labels, resource limits, and namespace-level controls across clusters and environments. Open Policy Agent describes policy as code that decouples policy decision-making from enforcement, enabling consistent policy decisions across a distributed system. Kyverno similarly provides Kubernetes-native policies for validating, mutating, and generating configuration. This centralized policy layer reduces duplicated effort and makes secure, compliant consumption of platform services predictable for all teams. See Open Policy Agent documentation and Kyverno documentation.

QUESTION NO: 2

What is the fundamental difference between a CI/CD and a GitOps deployment model for Kubernetes application deployments?

A. CI/CD is predominantly a pull model, with the container image providing the desired state.

B. GitOps is predominantly a push model, with an operator reflecting the desired state.

C. GitOps is predominantly a pull model, with a controller reconciling desired state.

D. CI/CD is predominantly a push model, with the user providing the desired state.

ANSWER: C

Explanation:

GitOps is predominantly a pull model, with a controller reconciling desired state. In GitOps, declarative Kubernetes configuration is stored in a Git repository that serves as the source of truth for the intended application and infrastructure state. A GitOps controller, such as Flux or Argo CD, runs in or has controlled access from the cluster environment and regularly checks the repository for changes. It compares the version-controlled desired state with the cluster's live state and takes action to converge the live environment on that declared state. This continuous comparison and correction process is reconciliation.

The pull characteristic is important because the deployment agent initiates retrieval of configuration from Git rather than an external delivery system directly invoking the Kubernetes API to deploy a change. This also supports ongoing drift detection: if a resource is changed manually or otherwise diverges from the repository's declaration, the controller can detect that difference and restore the declared configuration. The OpenGitOps principles describe Git as the source of truth and software agents as the components that continuously reconcile actual state with desired state. Flux likewise describes its controllers as reconciling cluster state from Git-managed configuration. See the OpenGitOps principles and Flux documentation.

QUESTION NO: 3

A platform team is implementing an API-driven approach to enable development teams to consume platform capabilities more effectively. Which of the following examples best illustrates this approach?

A. Providing a documented process for developers to submit feature requests for the platform.

B. Developing a dashboard that visualizes platform usage statistics without exposing any APIs.

C. Allowing developers to request and manage development environments on demand through an internal tool.

D. Implementing a CI/CD pipeline that automatically deploys updates to the platform based on developer requests.

ANSWER: C

Explanation:

Allowing developers to request and manage development environments on demand through an internal tool best represents an API-driven platform approach when that tool provides a self-service interface over platform APIs. Rather than requiring a platform engineer to manually create each environment, developers can use a standardized workflow to provision, update, view, and remove environments when needed. The internal tool may be a developer portal, command-line interface, or another user-facing abstraction, but its value is that it exposes platform capabilities as repeatable, on-demand services.

This model is central to an internal developer platform: the platform team builds and operates reusable “golden paths,” while development teams consume them independently within defined guardrails. Environment creation can therefore consistently apply approved templates, access policies, resource limits, networking settings, and observability defaults. API-backed self-service also makes the process automatable from development workflows and reduces delays caused by ticket-based handoffs. The result is a better developer experience without sacrificing the platform team’s ability to enforce operational and governance standards. The CNCF describes platform engineering as creating internal platforms that provide self-service capabilities and curated developer experiences; see the CNCF overview of platform engineering and the Platform Engineering community’s IDP definition.

QUESTION NO: 4

In the context of observability for cloud native platforms, which of the following best describes the role of OpenTelemetry?

A. OpenTelemetry is primarily used for logging data only.

B. OpenTelemetry is a proprietary solution that limits its use to specific cloud providers.

C. OpenTelemetry provides a standardized way to collect and transmit observability data.

D. OpenTelemetry is solely focused on infrastructure monitoring.

ANSWER: C

Explanation:

OpenTelemetry provides a standardized way to collect and transmit observability data. It is a vendor-neutral, open-source observability framework and CNCF project that defines common APIs, SDKs, semantic conventions, and tooling for generating and handling telemetry. Its core telemetry signals are traces, metrics, and logs, enabling teams to instrument cloud native applications consistently rather than adopting incompatible, vendor-specific instrumentation for each backend. OpenTelemetry also includes the OpenTelemetry Collector, which can receive, process, transform, and export telemetry to one or more observability backends. This separation of instrumentation from the destination makes application telemetry more portable and supports flexible platform architectures. For example, a platform team can use common instrumentation across services while routing data to different monitoring or tracing systems as operational requirements evolve. The project’s goal is therefore to standardize the production, collection, and export of telemetry data, providing a foundational layer for observability in distributed cloud native environments. See the OpenTelemetry overview and the OpenTelemetry Collector documentation.

QUESTION NO: 5

In the context of Agile methodology, which principle aligns best with DevOps practices in platform engineering?

A. Customer involvement should be limited during the development process to avoid disruptions.

B. Teams should continuously gather feedback and iterate on their work to improve outcomes.

C. Teams should strictly adhere to initial project plans without making adjustments during development.

D. Development and operations teams should remain separate to maintain clear responsibilities.

ANSWER: B

Explanation:

Teams should continuously gather feedback and iterate on their work to improve outcomes. This principle directly connects Agile's empirical approach with DevOps and platform-engineering practices. Agile relies on frequent inspection of outcomes and adaptation based on what is learned, rather than treating delivery as a one-time execution of a fixed plan. In a platform context, feedback can come from application developers using platform self-service capabilities, operational signals such as reliability and deployment performance, security findings, and user-facing service outcomes. Platform teams use that information to iteratively improve workflows, templates, documentation, observability, automation, and the overall developer experience.

DevOps reinforces these feedback loops across the full software lifecycle. Continuous integration and delivery shorten the time between a change and useful evidence about its quality and behavior. Monitoring and observability provide production feedback that helps teams prioritize improvements and operate services reliably. The Agile Manifesto explicitly values responding to change, while the DevOps approach emphasizes collaboration, automation, measurement, and sharing to enable continual learning and improvement. These complementary practices help platform engineering deliver capabilities that are useful, reliable, and responsive to the needs of internal developers and the organization. See the Agile Manifesto principles and the Cloud Native Computing Foundation project landscape for cloud-native technologies that support iterative delivery and operational feedback.

QUESTION NO: 6

A platform team wants to let developers provision cloud services like S3 buckets and databases using Kubernetes-native APIs, without exposing cloud-specific details. Which tool is best suited for this?

A. Cluster API

B. Crossplane

C. Helm

D. OpenTofu

ANSWER: B

Explanation:

Crossplane is purpose-built for delivering cloud infrastructure through Kubernetes-native APIs. It extends the Kubernetes control plane with custom resource definitions and providers, allowing cloud resources—including object storage buckets, managed databases, networking, and other services—to be declared, reconciled, and managed using Kubernetes-style manifests. This means developers can request a service through a familiar API rather than needing direct knowledge of a particular cloud provider's console, SDK, or resource model.

For platform engineering use cases, Crossplane supports higher-level abstractions through composite resources and compositions. A platform team can define an opinionated API such as a database claim, then have Crossplane compose the underlying managed resources, configuration, access policies, and provider-specific settings. Developers consume the simplified interface, while the platform team retains control over security, compliance, naming, lifecycle defaults, and implementation details. The same abstraction can also be backed by different providers where needed, helping avoid exposing cloud-specific details to application teams.

This declarative model fits naturally with Kubernetes operations and GitOps workflows because the desired infrastructure state is represented as Kubernetes resources and continuously reconciled. See the Crossplane documentation and its overview of compositions.

QUESTION NO: 7

A team configures a HorizontalPodAutoscaler to scale a Deployment when average CPU utilization exceeds 70 percent. The Pods have CPU limits but no CPU requests. Why can this prevent CPU-utilization-based scaling from working as intended?

A. CPU utilization is calculated relative to requested CPU, so the autoscaler lacks the required baseline.

B. CPU limits prevent the Metrics Server from collecting CPU usage for the Pods.

C. The Deployment must use a StatefulSet before it can be scaled by CPU utilization.

D. The autoscaler can calculate CPU utilization only when every Pod runs on a dedicated node.

ANSWER: A

Explanation:

CPU utilization is calculated relative to requested CPU, so the autoscaler lacks the required baseline is correct. For resource-metric utilization targets, the HorizontalPodAutoscaler uses the resource request as the denominator when calculating average utilization. Without a CPU request for the relevant containers, Kubernetes cannot determine the target utilization percentage in the normal way.

A CPU limit constrains maximum CPU use but does not establish the scheduling request used for this utilization calculation. Increasing the replica count manually or using only node-level metrics does not correct the missing workload resource request.

QUESTION NO: 8

A platform team measures deployment frequency and lead time for changes after launching a new internal developer platform. Both measures improve, but application teams report repeated production incidents caused by untested configuration changes. Which improvement should the platform team prioritize next?

A. Increase deployment frequency so configuration failures are corrected more quickly.

B. Add automated validation and promotion gates for configuration changes before production.

C. Require platform engineers to manually review every production deployment.

D. Prioritize cloud-cost reporting for each application namespace.

ANSWER: B

Explanation:

Add automated validation and promotion gates for configuration changes before production is correct because faster delivery metrics alone do not establish release quality or reliability. Configuration should be validated through repeatable checks, such as schema validation, policy checks, integration testing, and environment-specific verification, before it is promoted to production.

Increasing deployment frequency could increase risk if validation remains inadequate. Measuring cloud cost may be useful but does not directly prevent configuration-related incidents. Requiring platform engineers to manually review every deployment recreates a bottleneck rather than providing scalable, consistent controls.

QUESTION NO: 9

What is the primary advantage of using a declarative approach to Infrastructure as Code (IaC) over an imperative approach?

A. Declarative IaC focuses on the “what” rather than the “how,” simplifying the management of infrastructure.

B. Declarative IaC is less suitable for dynamic environments compared to imperative IaC.

C. Declarative IaC allows for more granular control over resource provisioning.

D. Declarative IaC requires more coding effort compared to imperative IaC.

ANSWER: A

Explanation:

Declarative IaC focuses on the “what” rather than the “how,” simplifying the management of infrastructure. With a declarative model, an engineer specifies the intended end state—for example, the required networks, compute capacity, access policies, or Kubernetes objects—rather than writing every command and procedural step needed to create them. The IaC tool is responsible for determining the actions needed to converge actual infrastructure toward that declared state.

This approach makes infrastructure definitions easier to review, version, reproduce, and apply consistently across environments. It also supports safe, repeatable change management: teams can compare the current state with the desired configuration, plan changes, and use automation to reduce configuration drift. In cloud-native platforms, declarative resource definitions are especially valuable because controllers continuously reconcile observed state with the desired state. Kubernetes documents this model through its API objects and control loops, while Terraform describes configuration as the desired infrastructure resources and their relationships. These characteristics make declarative IaC a natural fit for collaborative operations and GitOps-style workflows.

References: Kubernetes Controllers and Terraform Documentation.

QUESTION NO: 10

What is a key cultural aspect that drives successful platform adoption in an organization?

A. Mandating that all teams must use the platform without exceptions

B. Keeping platform development separate from application teams.

C. Prioritizing platform security over usability.

D. Encouraging platform feedback loops from developers to improve usability.

ANSWER: D

Explanation:

Encouraging platform feedback loops from developers to improve usability is a key cultural driver of successful platform adoption because an internal platform succeeds when it is treated as a product for its developer users. Platform engineers need regular, actionable input from application teams about friction points, missing capabilities, documentation gaps, and the quality of the developer experience. This feedback enables the platform team to prioritize improvements that reduce cognitive load and make the preferred path the easiest path for developers to take.

A collaborative feedback culture also creates trust and shared ownership. Rather than viewing the platform as a centrally imposed set of tools, developers can see that their needs shape its roadmap. Teams are consequently more likely to adopt the platform voluntarily and consistently, while the platform team can validate whether its services solve real delivery problems. The CNCF’s platform engineering guidance emphasizes platform-as-a-product thinking and continuous engagement with platform users. Similarly, Team Topologies describes the importance of reducing cognitive load and enabling teams through effective interaction modes. See the CNCF Platforms Whitepaper and Team Topologies key concepts.