Question 1

Question Type: MultipleChoice

Which approach is effective for scalable Kubernetes infrastructure provisioning?

Options:

A- Helm charts with the environment values.yaml

B- Imperative scripts using Kubernetes API

C- Static YAML with kubectl apply

D- Crossplane compositions defining custom CRDs

Answer:

D

Explanation:

The most effective approach for scalable Kubernetes infrastructure provisioning is Crossplane compositions. Option D is correct because compositions let platform teams define custom CRDs (Composite Resources) that abstract infrastructure details while embedding organizational policies and guardrails. Developers then consume these abstractions through simple Kubernetes-native APIs, enabling self-service at scale.

Option A (Helm with values.yaml) is useful for application deployment but not for scalable infrastructure provisioning across multiple clouds. Option B (imperative scripts) lacks scalability, repeatability, and governance. Option C (static YAML with kubectl apply) is manual and not suited for dynamic, multi-team environments.

Crossplane compositions allow platform teams to curate golden paths while giving developers autonomy. This reduces complexity, ensures compliance, and supports multi-cloud provisioning---all key aspects of platform engineering.

--- CNCF Crossplane Project Documentation

--- CNCF Platforms Whitepaper

--- Cloud Native Platform Engineering Study Guide

Question 2

Question Type: MultipleChoice

Which provisioning strategy ensures efficient resource scaling for an application on Kubernetes?

Options:

A- Implementing a fixed resource allocation that does not change regardless of demand.
B- Using a declarative approach with Infrastructure as Code (IaC) tools to define resource requirements.
C- Manual provisioning of resources based on predicted traffic.
D- Using an imperative approach to script resource changes in response to traffic spikes.

Answer:

B

Explanation:

The most efficient and scalable strategy is to use a declarative approach with Infrastructure as Code (IaC). Option B is correct because declarative definitions specify the desired state (e.g., resource requests, limits, autoscaling policies) in code, allowing Kubernetes controllers and autoscalers to reconcile and enforce them dynamically. This ensures that applications can scale efficiently based on actual demand.

Option A (fixed allocation) is inefficient, leading to wasted resources during low usage or insufficient capacity during high demand. Option C (manual provisioning) introduces delays, risk of error, and operational overhead. Option D (imperative scripting) is not sustainable for large-scale or dynamic workloads, as it requires constant manual intervention.

Declarative IaC aligns with GitOps workflows, enabling automated, version-controlled scaling decisions. Combined with Kubernetes' Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler, this approach allows platforms to balance cost efficiency with application reliability.

--- CNCF GitOps Principles

--- Kubernetes Autoscaling Documentation

--- Cloud Native Platform Engineering Study Guide

Question 3

Question Type: MultipleChoice

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

Options:

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:

The fundamental difference between a traditional CI/CD model and a GitOps model lies in how changes are applied to the Kubernetes cluster---whether they are 'pushed' to the cluster by an external system or 'pulled' by an agent running inside the cluster.

CI/CD (Push Model)

In a typical CI/CD pipeline for Kubernetes, the CI/CD server (like Jenkins, GitLab CI, or GitHub Actions) is granted credentials to access the cluster. When a pipeline runs, it executes commands like kubectl apply or helm upgrade to push the new application configuration and image versions directly to the Kubernetes API server.

Actor: The CI/CD pipeline is the active agent initiating the change.

Direction: Changes flow from the CI/CD system to the cluster.

Security: Requires giving cluster credentials to an external system.

In a GitOps model, a Git repository is the single source of truth for the desired state of the application. An agent or controller (like Argo CD or Flux) runs inside the Kubernetes cluster. This controller continuously monitors the Git repository.

When it detects a difference between the desired state defined in Git and the actual state of the cluster, it pulls the changes from the repository and applies them to the cluster to bring it into the desired state. This process is called reconciliation.

Actor: The in-cluster controller is the active agent initiating the change.

Direction: The cluster pulls its desired state from the Git repository.

Security: The cluster's credentials never leave its boundary. The controller only needs read-access to the Git repository.

Question 4

Question Type: MultipleChoice

A development team is struggling to find and connect to various services within a cloud platform. What is the primary benefit of implementing an API-driven service catalog for this team?

Options:

A- It enables easier service discovery through a consistent interface.
B- It increases the time taken to provision services.
C- It allows the team to bypass security protocols.
D- It requires the development team to manage provisioning details themselves.

Answer:

A

Explanation:

An API-driven service catalog provides a centralized and standardized interface where developers can discover and provision platform services. Option A is correct because it simplifies service discovery, allowing teams to connect to databases, messaging systems, and other infrastructure without needing in-depth platform knowledge. This improves productivity and developer experience by reducing cognitive load and ensuring consistent, governed access.

Option B is the opposite of the benefit---catalogs accelerate provisioning. Option C is incorrect because catalogs do not bypass security; they enforce guardrails and compliance. Option D is also incorrect because service catalogs abstract away provisioning details rather than forcing developers to manage them.

By providing golden paths and API-driven self-service, service catalogs ensure developers focus on building applications while platform teams maintain consistency and compliance.

--- CNCF Platforms Whitepaper

--- CNCF Platform Engineering Maturity Model

--- Cloud Native Platform Engineering Study Guide

Question 5

Question Type: MultipleChoice

Why is centralized configuration management important in a multi-cluster GitOps setup?

Options:

A- It requires all clusters to have the exact same configuration, including secrets and environment variables, to maintain uniformity.

B- It ensures consistent and auditable management of configurations and policies across clusters from a single Git repository or set of coordinated repositories.

C- It eliminates the need for automated deployment tools like Argo CD or Flux since configurations are already stored centrally.

D- It makes it impossible for different teams to customize configurations for specific clusters, reducing flexibility.

Answer:

B

Explanation:

In a GitOps-driven multi-cluster environment, centralized configuration management ensures that platform teams can maintain consistency, governance, and security across multiple clusters, all while leveraging Git as the single source of truth. Option B is correct because centralization allows teams to enforce policies, apply configurations, and audit changes across environments in a traceable and reproducible way. This supports compliance, as every change is version-controlled, peer-reviewed, and automatically reconciled by tools like Argo CD or Flux.

Option A is misleading---centralized management does not mean clusters must have identical configurations; it enables consistent patterns while still allowing environment-specific overlays or customizations (e.g., dev vs. prod). Option C is incorrect because GitOps tools remain essential for continuous reconciliation between desired and actual state. Option D is also incorrect because centralized management does not remove flexibility---it supports parameterization and customization per cluster.

By combining centralization with declarative configuration and GitOps automation, organizations gain operational efficiency, faster recovery from drift, and improved auditability in multi-cluster scenarios.

--- CNCF GitOps Principles for Platforms

--- CNCF Platforms Whitepaper

--- Cloud Native Platform Engineering Study Guide

Question 6

Question Type: MultipleChoice

A team wants to deploy a new feature to production for internal users only and be able to instantly disable it if problems occur, without redeploying code. Which strategy is most suitable?

Options:

A- Use a blue/green deployment to direct internal users to one version and switch as needed.
B- Use feature flags to release the feature to selected users and control its availability through settings.
C- Use a canary deployment to gradually expose the feature to a small group of random users.
D- Deploy the feature to all users and prepare to roll it back manually if an issue is detected.

Answer:

B

Explanation:

Feature flags are the most effective way to control feature exposure to specific users, such as internal testers, while enabling fast rollback without redeployment. Option B is correct because feature flags allow teams to decouple deployment from release, giving precise runtime control over feature availability. This means that once the code is deployed, the team can toggle the feature on or off for different cohorts (e.g., internal users) dynamically.

Option A (blue/green deployment) controls traffic between two environments but does not provide user-level granularity. Option C (canary deployments) gradually expose changes but focus on random subsets of users rather than targeted groups such as internal employees. Option D requires redeployment or rollback, which introduces risk and slows down incident response.

Feature flags are widely recognized in platform engineering as a core continuous delivery practice that improves safety, accelerates experimentation, and enhances resilience by enabling immediate mitigation of issues.

--- CNCF Platforms Whitepaper

--- Cloud Native Platform Engineering Study Guide

--- Continuous Delivery Foundation Guidance

Question 7

Question Type: MultipleChoice

Which of the following is a primary benefit of adopting a platform approach for managing application environments with diverse needs?

Options:

A- It enables self-service infrastructure provisioning while supporting app-specific requirements and organizational standards.
B- It isolates application environments completely to maximize security and avoid shared resources.
C- It enforces one infrastructure setup for all applications to reduce management complexity.
D- It centralizes all deployments in one environment to improve control and visibility.

Answer:

A

Explanation:

The main advantage of a platform engineering approach is balancing self-service for developers with organizational governance and standardization. Option A is correct because platforms enable developers to provision infrastructure and application environments independently while embedding security, compliance, and operational guardrails. This ensures that applications with diverse needs (e.g., different scaling patterns, compliance requirements, or environments) can still operate within a unified governance framework.

Option B (isolation only) is sometimes required for compliance but does not address the broader benefit of balancing flexibility and standardization. Option C forces uniformity, which reduces adaptability for varied workloads. Option D (centralized deployments) reduces developer autonomy and scalability.

The platform approach enables golden paths, curated abstractions, and reusable services, allowing diverse applications to thrive while maintaining control. This balance is central to platform engineering's goal of reducing cognitive load and improving developer productivity.

--- CNCF Platforms Whitepaper

--- CNCF Platform Engineering Maturity Model

--- Cloud Native Platform Engineering Study Guide

Question 8

Question Type: MultipleChoice

In a cloud native environment, which factor most critically influences the need for customized CI pipeline configurations across different application types?

Options:

A- The organizational practice of assigning unique pipeline configurations based on application priority levels.

B- The technical differences in build tools, testing frameworks, and artifact formats across programming languages.

C- The need to accommodate varying team sizes and developer expertise levels within the organization.

D- The requirement to visually distinguish between different application pipelines in monitoring dashboards.

Answer:

B

Explanation:

The biggest driver for customizing CI pipeline configurations across application types is technical differences between programming languages, frameworks, and artifact formats. Option B is correct because applications written in Java, Python, Go, or Node.js require different build tools (e.g., Maven, pip, go build, npm), testing frameworks, and packaging mechanisms. These differences must be reflected in the CI pipeline to ensure successful builds, tests, and artifact generation.

Option A (priority-based pipelines) is more of an organizational practice, not a technical necessity. Option C (team sizes and expertise) may influence usability but does not drive pipeline configuration. Option D (visual distinction) relates to dashboards and observability, not pipeline functionality.

Platform engineers often provide pipeline templates or abstractions that encapsulate these differences while standardizing security and compliance checks. This balances customization with consistency, enabling developers to use pipelines suited to their technology stack without fragmenting governance.

--- CNCF Platforms Whitepaper

--- Continuous Delivery Foundation Guidance

--- Cloud Native Platform Engineering Study Guide

Question 9

Question Type: MultipleChoice

Which platform component enables one-click provisioning of sandbox environments, including both infrastructure and application code?

Options:

A- CI/CD pipeline

B- Service mesh

C- Service bus

D- Observability pipeline

Answer:

A

Explanation:

A CI/CD pipeline is the platform component that enables automated provisioning of sandbox environments with both infrastructure and application code. Option A is correct because modern pipelines integrate Infrastructure as Code (IaC) with application deployment, enabling ''one-click'' or self-service provisioning of complete environments. This capability is central to platform engineering because it empowers developers to spin up temporary or permanent sandbox environments quickly for testing, experimentation, or demos.

Option B (service mesh) focuses on secure, observable service-to-service communication but does not provision environments. Option C (service bus) is used for asynchronous communication between services, not environment provisioning. Option D (observability pipeline) deals with collecting telemetry data, not provisioning.

By leveraging CI/CD pipelines integrated with GitOps and IaC tools (such as Terraform, Crossplane, or Kubernetes manifests), platform teams ensure consistency, compliance, and automation. Developers benefit from reduced friction, faster feedback cycles, and a better overall developer experience.

--- CNCF Platforms Whitepaper

--- CNCF GitOps Principles

--- Cloud Native Platform Engineering Study Guide