1 CNPE

Generated on: 2026-01-13 15:04:44 **Version:** 1.0

1.1 Table of Contents

1.2 Overview

The **Certified Cloud Native Platform Engineer (CNPE)** exam demonstrates advanced knowledge of platform engineering, including designing, building, and operating cloud native platforms.

> **Note**: This certification is required for Golden Kubestronaut status after March 1st, 2025.

1.3 Exam Overview

Detail	Information
Exam Format	Performance-based (hands-on)
Duration	2 hours
Passing Score	66%
Certification Validity	3 years
Cost	$395 USD
Retake Policy	1 free retake

1.4 Exam Domains & Weights

Domain	Weight
Platform Design	20%
Platform Implementation	30%
Platform Operations	25%
Security and Governance	15%
Developer Experience	10%

1.5 Key Topics

1.5.1 Platform Design

- Architecture patterns
- Multi-tenancy strategies
- API design
- Scalability considerations

1.5.2 Platform Implementation

- Kubernetes operators
- Custom controllers
- GitOps implementation
- Infrastructure as Code

1.5.3 Platform Operations

- Day-2 operations
- Upgrades and migrations
- Disaster recovery
- Capacity planning

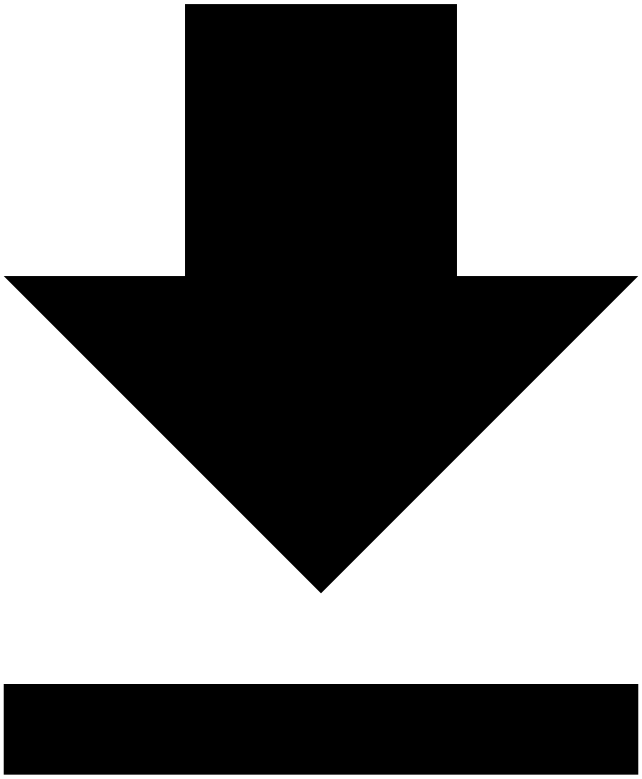
1.6 Study Resources

- Platform Engineering
- CNCF Platforms White Paper
- CNPE Curriculum
- Kubernetes Operators

1.7 Navigation

- Next: Sample Questions →

1.8 Platform Engineering



Download PDF Version

Comprehensive guide to platform engineering for CNPE certification.

1.9 Overview

Platform Engineering focuses on building Internal Developer Platforms (IDPs) that:

- **Abstract complexity** - Hide infrastructure details
- **Enable self-service** - Developers provision resources independently
- **Standardize** - Consistent patterns across teams
- **Automate** - Reduce manual operations

1.10 Key Concepts

1.10.1 Internal Developer Platform (IDP)

An IDP provides:

- Self-service infrastructure provisioning
- Standardized deployment pipelines
- Observability and monitoring
- Security and compliance guardrails

1.10.2 Platform Team Responsibilities

- Build and maintain platform components
- Define golden paths for common workflows
- Provide documentation and support
- Measure and improve developer experience

1.11 Crossplane

Crossplane extends Kubernetes to manage cloud resources:

1.11.1 Installation

```
# Install Crossplane
kubectl create namespace crossplane-system
helm repo add crossplane-stable https://charts.crossplane.io/
        stable
helm install crossplane crossplane-stable/crossplane -n
        crossplane-system

# Install AWS provider
kubectl apply -f - <<EOF
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-aws
spec:
  package: xpkg.upbound.io/crossplane-contrib/provider-aws:v0.40.0
EOF
```

1.11.2 Composite Resources

```
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: databases.platform.example.com
spec:
```

```
  group: platform.example.com
  names:
    kind: Database
    plural: databases
  versions:
  - name: v1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              size:
                type: string
                enum: [small, medium, large]
              engine:
                type: string
                enum: [postgres, mysql]
```

1.11.3 Composition

```
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: database-aws
spec:
  compositeTypeRef:
    apiVersion: platform.example.com/v1
    kind: Database
  resources:
  - name: rds-instance
    base:
      apiVersion: rds.aws.crossplane.io/v1beta1
      kind: Instance
      spec:
        forProvider:
          region: us-east-1
          dbInstanceClass: db.t3.micro
          engine: postgres
          masterUsername: admin
    patches:
    - fromFieldPath: spec.size
      toFieldPath: spec.forProvider.dbInstanceClass
      transforms:
      - type: map
        map:
          small: db.t3.micro
```

```
          medium: db.t3.small
          large: db.t3.medium
```

1.12 Backstage

Backstage is a developer portal platform:

1.12.1 Features

- **Software Catalog** - Track all services and resources
- **Templates** - Scaffold new projects
- **TechDocs** - Documentation as code
- **Plugins** - Extensible architecture

1.12.2 Catalog Entity

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: my-service
  description: My microservice
  annotations:
    github.com/project-slug: myorg/my-service
    backstage.io/techdocs-ref: dir:.
spec:
  type: service
  lifecycle: production
  owner: team-platform
  system: my-system
  dependsOn:
  - resource:my-database
  providesApis:
  - my-api
```

1.12.3 Software Template

```
apiVersion: scaffolder.backstage.io/v1beta3
kind: Template
metadata:
  name: microservice-template
  title: Microservice Template
spec:
  owner: team-platform
  type: service
  parameters:
  - title: Service Details
    properties:
      name:
        type: string
```

```
        title: Service Name
      owner:
        type: string
        title: Owner Team
  steps:
  - id: fetch
    name: Fetch Template
    action: fetch:template
    input:
      url: ./skeleton
      values:
        name: ${{ parameters.name }}
  - id: publish
    name: Publish to GitHub
    action: publish:github
    input:
      repoUrl: github.com?owner=myorg&repo=${{ parameters.name }}
```

1.13 Platform Patterns

1.13.1 Golden Paths

Pre-defined, well-supported ways to accomplish common tasks:

1. **Service Creation** - Template with CI/CD, monitoring, logging
2. **Database Provisioning** - Self-service with guardrails
3. **Environment Management** - Dev, staging, production workflows

1.13.2 Self-Service Infrastructure

```
# Developer requests database
apiVersion: platform.example.com/v1
kind: Database
metadata:
  name: my-app-db
  namespace: my-team
spec:
  size: medium
  engine: postgres
```

1.13.3 Platform APIs

Expose platform capabilities through Kubernetes CRDs:

- Databases
- Message queues
- Storage buckets
- DNS entries

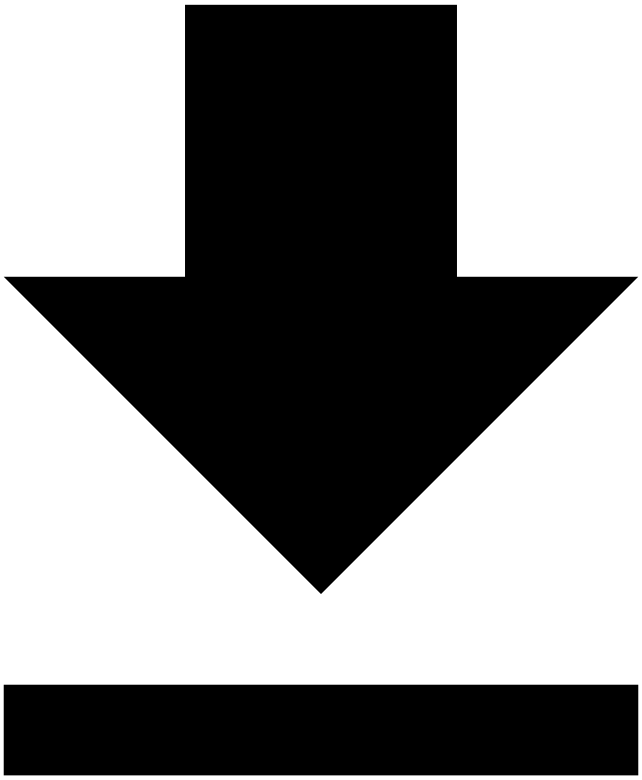
- Certificates

1.14 Best Practices

1. **Start small** - Begin with highest-impact capabilities
2. **Measure adoption** - Track developer satisfaction
3. **Document everything** - Self-service requires good docs
4. **Iterate based on feedback** - Platform is a product
5. **Maintain golden paths** - Keep them up to date

← Back to CNPE Overview

1.15 Sample Practice Questions



Download PDF Version

1.16 Practice Resources

- Platform Engineering
- Kubernetes Operators
- Killercoda Scenarios

1.17 Platform Design (20%)

1.17.1 Question 1

Design a multi-tenant platform architecture for 50 development teams.

Show Solution

Architecture considerations:

1. **Isolation Strategy:**
   - Namespace per team with resource quotas
   - Network policies for isolation
   - RBAC per namespace
2. **Resource Management:**

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-quota
  namespace: team-alpha
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    pods: "50"
```

3. **Shared Services:**
   - Centralized logging (Loki)
   - Metrics (Prometheus/Thanos)
   - Ingress controller
   - Certificate management
4. **Self-Service:**
   - Namespace provisioning via GitOps
   - Template-based deployments
   - Developer portal (Backstage)

1.17.2 Question 2

How do you design APIs for a platform?

Show Solution

Platform API Design Principles:

1. **Use Kubernetes-native APIs:**

```yaml
apiVersion: platform.example.com/v1
kind: Application
metadata:
  name: my-app
spec:
  image: myapp:v1
  replicas: 3
  ingress:
    enabled: true
    host: myapp.example.com
```

2. **Abstract complexity:**

   - Hide infrastructure details
   - Provide sensible defaults
   - Allow overrides when needed

3. **Versioning:**

   - Use API versioning (v1alpha1, v1beta1, v1)
   - Maintain backward compatibility
   - Deprecation policy

4. **Documentation:**

   - OpenAPI specs
   - Examples for common use cases

1.18 Platform Implementation (30%)

1.18.1 Question 3

Create a Kubernetes Operator that manages a custom Application resource.

Show Solution

```go
// Controller reconcile function
func (r *ApplicationReconciler) Reconcile(ctx
        context.Context, req ctrl.Request) (ctrl.Result, error) {
    var app platformv1.Application
    if err := r.Get(ctx, req.NamespacedName, &app); err != nil {
        return ctrl.Result{}, client.IgnoreNotFound(err)
```

```
    }

    // Create Deployment
    deployment := &appsv1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:      app.Name,
            Namespace: app.Namespace,
        },
        Spec: appsv1.DeploymentSpec{
            Replicas: &app.Spec.Replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: map[string]string{"app": app.Name},
            },
            Template: corev1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: map[string]string{"app": app.Name},
                },
                Spec: corev1.PodSpec{
                    Containers: []corev1.Container{{
                        Name:  app.Name,
                        Image: app.Spec.Image,
                    }},
                },
            },
        },
    }

    if err := ctrl.SetControllerReference(&app, deployment,
        r.Scheme); err != nil {
        return ctrl.Result{}, err
    }

    if err := r.Create(ctx, deployment); err != nil {
        if !errors.IsAlreadyExists(err) {
            return ctrl.Result{}, err
        }
    }

    return ctrl.Result{}, nil
}
```

1.18.2 Question 4

Implement GitOps for platform configuration.

Show Solution

Repository Structure:

```
platform-config/
├── clusters/
│   ├── production/
│   │   ├── flux-system/
```

```
│   │   └── kustomization.yaml
│   └── staging/
├── infrastructure/
│   ├── controllers/
│   ├── configs/
│   └── kustomization.yaml
└── tenants/
    ├── team-alpha/
    │   ├── namespace.yaml
    │   ├── rbac.yaml
    │   └── quota.yaml
    └── team-beta/
```

Flux Kustomization:

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: infrastructure
  namespace: flux-system
spec:
  interval: 10m
  sourceRef:
    kind: GitRepository
    name: platform-config
  path: ./infrastructure
  prune: true
  healthChecks:
  - apiVersion: apps/v1
    kind: Deployment
    name: ingress-nginx-controller
    namespace: ingress-nginx
```

1.18.3 Question 5

Create a Crossplane Composition for provisioning databases.

Show Solution

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: postgresql-aws
spec:
  compositeTypeRef:
    apiVersion: database.platform.io/v1
    kind: PostgreSQLInstance
  resources:
  - name: rds-instance
    base:
      apiVersion: rds.aws.crossplane.io/v1beta1
      kind: Instance
      spec:
```

```
        forProvider:
          engine: postgres
          engineVersion: "14"
          instanceClass: db.t3.medium
          allocatedStorage: 20
          publiclyAccessible: false
    patches:
    - fromFieldPath: spec.storageGB
      toFieldPath: spec.forProvider.allocatedStorage
    - fromFieldPath: spec.size
      toFieldPath: spec.forProvider.instanceClass
      transforms:
      - type: map
        map:
          small: db.t3.small
          medium: db.t3.medium
          large: db.t3.large
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: postgresqlinstances.database.platform.io
spec:
  group: database.platform.io
  names:
    kind: PostgreSQLInstance
    plural: postgresqlinstances
  versions:
  - name: v1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              size:
                type: string
                enum: [small, medium, large]
              storageGB:
                type: integer
```

1.19 Platform Operations (25%)

1.19.1 Question 6

Implement a cluster upgrade strategy with zero downtime.

Show Solution

Blue-Green Cluster Upgrade:

1. **Preparation:**

```
# Create new cluster with updated version
eksctl create cluster -f new-cluster.yaml

# Install platform components
flux bootstrap github --context=new-cluster ...
```

2. **Migration:**

```
# Sync workloads via GitOps
# Update DNS weights gradually

# Route 10% traffic to new cluster
aws route53 change-resource-record-sets \
  --hosted-zone-id $ZONE_ID \
  --change-batch file://weighted-10.json
```

3. **Validation:**

```
# Monitor error rates
# Check application health
# Verify data consistency
```

4. **Cutover:**

```
# Route 100% to new cluster
# Decommission old cluster
```

1.19.2 Question 7

Design a disaster recovery plan for a platform.

Show Solution

DR Strategy:

1. **Backup:**

```
# Velero backup schedule
apiVersion: velero.io/v1
kind: Schedule
metadata:
  name: daily-backup
spec:
  schedule: "0 2 * * *"
  template:
    includedNamespaces:
    - "*"
    excludedNamespaces:
```

```
    - kube-system
  storageLocation: aws-backup
  ttl: 720h
```

2. **Recovery Targets:**
   - RPO: 1 hour (data loss tolerance)
   - RTO: 4 hours (recovery time)
3. **DR Runbook:**
   - Restore cluster from backup
   - Verify DNS failover
   - Test application functionality
   - Notify stakeholders
4. **Regular Testing:**
   - Monthly DR drills
   - Documented procedures
   - Post-mortem reviews

1.20 Security and Governance (15%)

1.20.1 Question 8

Implement policy-as-code for platform governance.

Show Solution

Kyverno Policies:

```
# Require resource limits
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-limits
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-limits
    match:
      any:
      - resources:
          kinds: [Pod]
    validate:
      message: "Resource limits required"
      pattern:
        spec:
          containers:
          - resources:
```

```
              limits:
                memory: "?*"
                cpu: "?*"

---
# Require approved registries
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: allowed-registries
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-registry
    match:
      any:
      - resources:
          kinds: [Pod]
    validate:
      message: "Images must be from approved registries"
      pattern:
        spec:
          containers:
          - image: "gcr.io/myorg/* | docker.io/myorg/*"
```

1.20.2 Question 9

Implement RBAC for platform teams.

Show Solution

```
# Platform Admin Role
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: platform-admin
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]

---
# Team Developer Role
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: team-developer
  namespace: team-alpha
rules:
- apiGroups: ["", "apps", "batch"]
  resources: ["pods", "deployments", "services", "configmaps",
        "secrets", "jobs"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]
```

```yaml
- apiGroups: [""]
  resources: ["pods/log", "pods/exec"]
  verbs: ["get", "create"]

---
# Bind to team
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: team-alpha-developers
  namespace: team-alpha
subjects:
- kind: Group
  name: team-alpha-devs
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: team-developer
  apiGroup: rbac.authorization.k8s.io
```

1.21 Developer Experience (10%)

1.21.1 Question 10

Design a self-service namespace provisioning system.

Show Solution

GitOps-based Provisioning:

1. **Request Template:**

   ```yaml
   # teams/team-gamma/namespace-request.yaml
   apiVersion: platform.io/v1
   kind: TeamNamespace
   metadata:
     name: team-gamma
   spec:
     team: gamma
     owners:
     - user1@example.com
     - user2@example.com
     resourceQuota:
       cpu: "20"
       memory: 40Gi
     networkPolicy: restricted
   ```

2. **Controller generates:**

   - Namespace
   - ResourceQuota

- NetworkPolicy
- RBAC bindings
- Default LimitRange

3. **Workflow:**

   - Developer submits PR
   - Automated validation
   - Approval from platform team
   - Merge triggers provisioning

1.22 Exam Tips

1. **Know Kubernetes deeply** - Operators, controllers, CRDs
2. **Practice GitOps** - Flux, Argo CD configurations
3. **Understand IaC** - Crossplane, Terraform
4. **Know policy tools** - Kyverno, OPA/Gatekeeper
5. **Practice troubleshooting** - Logs, events, debugging

← Back to CNPE Overview
