PW SKILLS | DevOps and Cloud Computing

Kubernetes Controllers, Security Best Practices, and Autoscaling

Objective

- Understand how ReplicaSets, Deployments, DaemonSets, and StatefulSets manage workloads.
- Learn how the Kubernetes scheduler places Pods based on resources and affinities.
- Implement container security best practices, including RBAC, image scanning, and securing Secrets.
- Configure autoscaling policies for optimizing Kubernetes resources dynamically.

Explaining how ReplicaSets ensure Pod availability by maintaining a specified number of replicas.

Let’s see

ReplicaSets ensure Pod availability by maintaining a specified number of replicas through continuous monitoring and automated adjustments. Here's how:

- Desired State Maintenance
- Pod Management via Label Selectors
- Self-Healing Mechanism
- Scaling Capabilities

PW SKILLS

Discussing the role of Deployments in managing application updates and rollbacks.

Let’s discuss

Kubernetes Deployments play a critical role in managing application updates and rollbacks by providing an automated, declarative, and reliable mechanism for maintaining application states. Key functions include:

- Rolling Updates
- Rollback Capability
- Declarative Configuration
- Self-Healing

PW SKILLS

Introducing DaemonSets for running system-level services on every node.

Let’s see

DaemonSets in Kubernetes are used to run system-level services by ensuring that a specific Pod is deployed on every node (or a subset of nodes) within a cluster.

They are ideal for tasks such as log collection, node monitoring, and running background processes that need to operate consistently across all nodes.

Key Features

- Automatic Deployment
- Node Affinity
- Persistence

Pop Quiz

Q. What is the primary purpose of a DaemonSet in Kubernetes?

A

To ensure all nodes run a copy of a specific pod

B

To manage stateful applications

Pop Quiz

Q. What is the primary purpose of a DaemonSet in Kubernetes?

A

To ensure all nodes run a copy of a specific pod

B

To manage stateful applications

PW SKILLS

Explaining StatefulSets for managing stateful applications with persistent storage.

Let’s see

StatefulSets in Kubernetes are specialized controllers designed to manage stateful applications requiring persistent storage and stable identities. Key features include:

- Persistent Storage
- Stable Network Identity
- Ordered Operations

PW SKILLS

Demonstrating how controllers manage Pods automatically instead of using standalone Pod configurations.

Let’s do it

Kubernetes controllers automate Pod management by maintaining the desired state defined in workload resources (like Deployments or StatefulSets), eliminating the need for manual standalone Pod configurations. Here's how they work:

Key Mechanisms of Controller-Based Pod Management:

1. Self-Healing

Controllers continuously monitor Pod health. If a Pod crashes or is deleted, the controller automatically creates a replacement to maintain the desired replica count. For example:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 3  # Controller ensures exactly 3 Pods always exist
```

Let’s do it

2. Automatic Scaling

HorizontalPodAutoscaler adjusts Pod counts based on real-time metrics:

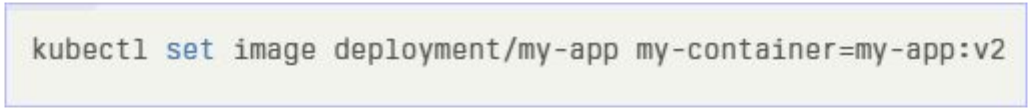
- This automatically scales Pods when CPU usage exceeds 50%.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Let’s do it

3. Rolling Updates

Controllers enable zero-downtime deployments by gradually replacing old Pods with new ones:



```
kubectl set image deployment/my-app my-container=my-app:v2
```

- The Deployment controller coordinates phased rollouts and automatic rollback if failures occur.

Let’s do it

Workflow Example:

1. User defines desired state in a Deployment

2. Controller:

- Creates Pods using the Pod template
- Monitors Pod health via kubelet
- Replaces failed Pods within seconds
- Adjusts replica count during scaling events
- Manages update rollouts

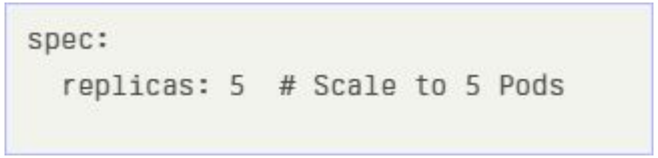
Explaining how Deployments and ReplicaSets handle application scaling and rolling updates.

Let’s do it

Deployments and ReplicaSets in Kubernetes handle application scaling and rolling updates efficiently by automating the management of Pods. Here's how:

Application Scaling

- ReplicaSets: A ReplicaSet ensures a specified number of Pod replicas are running at all times. You can scale applications by adjusting the replicas field in a Deployment or directly in the ReplicaSet. For example:



```yaml
spec:
  replicas: 5  # Scale to 5 Pods
```

Let’s do it

Kubernetes automatically creates or deletes Pods to match the desired replica count.

- Autoscaling: Deployments can integrate with HorizontalPodAutoscaler to dynamically adjust the number of replicas based on metrics like CPU or memory usage.

Let’s do it

Rolling Updates

- Incremental Updates: Deployments use the RollingUpdate strategy to replace old Pods with new ones incrementally, ensuring zero downtime. Kubernetes waits for new Pods to pass readiness checks before terminating old ones.

Configuration Parameters:

- maxUnavailable
- maxSurge
- Rollback Capability: If an update fails, Deployments allow rolling back to a previous stable version quickly, minimizing disruption.

PW SKILLS

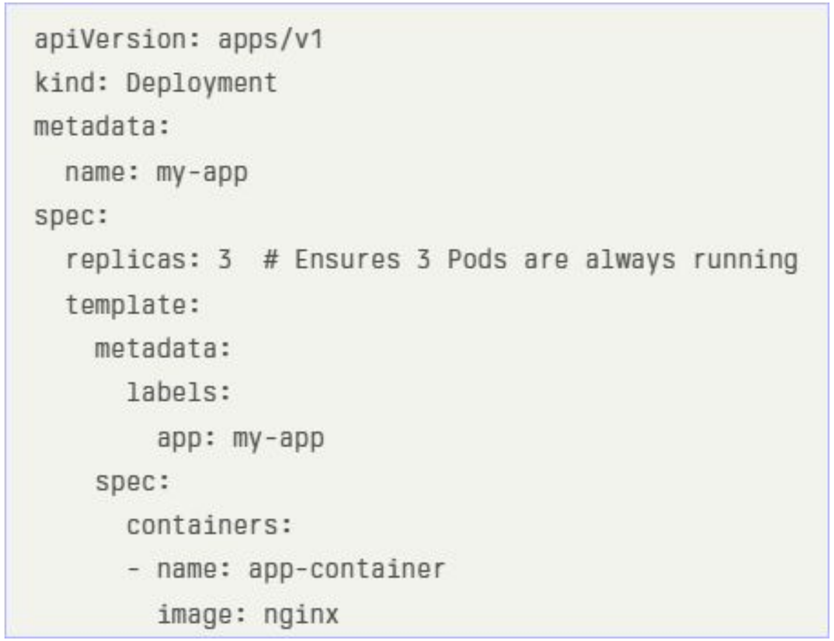
Showing how Pods are recreated when deleted or fail using Kubernetes controllers.

Let’s do it

Kubernetes controllers, such as ReplicaSets or Deployments, automatically recreate Pods when they are deleted or fail. Here's how it works:

1. **Desired State Monitoring:** Controllers continuously monitor the cluster to ensure the actual number of running Pods matches the desired state specified in the controller's configuration (e.g., replicas: 3).
2. **Pod Replacement:** If a Pod is deleted or crashes, the controller detects the discrepancy and immediately creates a new Pod to maintain the desired replica count.

For example:



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3  # Ensures 3 Pods are always running
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: app-container
        image: nginx
```

3. Automatic Rescheduling: If a node hosting a Pod fails, the controller reschedules the replacement Pod on another healthy node in the cluster.

PW SKILLS

Explaining the role of the scheduler in assigning Pods to Nodes based on availability and resources

Let’s see

The Kubernetes scheduler assigns Pods to Nodes by evaluating their resource requirements and constraints. Here's how it works:

1. Monitoring Unscheduled Pods
2. Filtering Nodes
3. Scoring Nodes
4. Binding Pods to Nodes

PW SKILLS

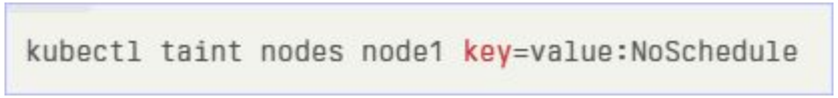
Discussing Pod placement strategies, including taints, tolerations, and affinity rules.

Let’s discuss

Kubernetes provides several strategies for controlling Pod placement on Nodes, including taints, tolerations, and affinity rules. Here’s how they work:

1. Taints and Tolerations

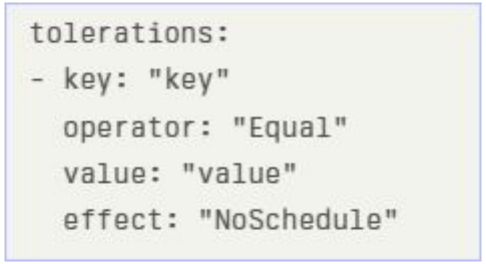
- Taints: Applied to Nodes to repel unwanted Pods. For example, a Node can be tainted to prevent general workloads from being scheduled on it:



```
kubectl taint nodes node1 key=value:NoSchedule
```

Let’s discuss

- Tolerations: Added to Pods to allow them to be scheduled on tainted Nodes. For example:

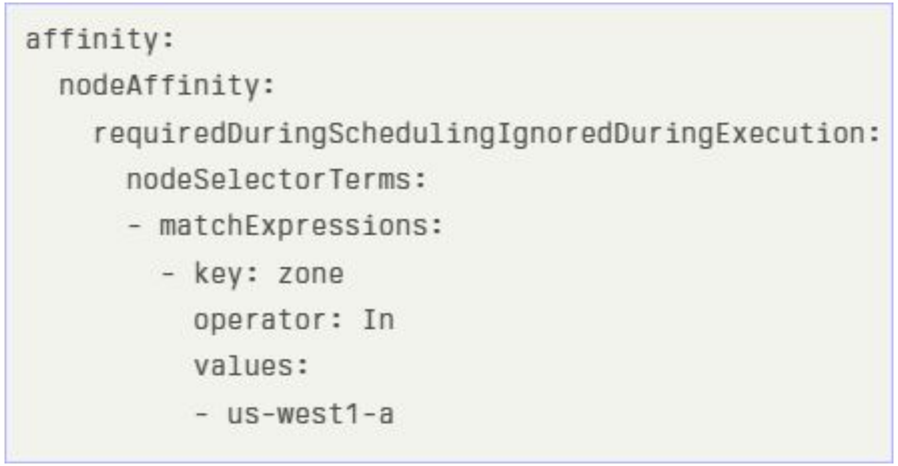


```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

Let’s discuss

2. Node Affinity and Anti-Affinity

- Node Affinity: Directs Pods to Nodes with specific labels. For example, scheduling Pods on Nodes in a specific zone:



```yaml
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: zone
          operator: In
          values:
          - us-west1-a
```

Let’s discuss

- Node Anti-Affinity: Prevents Pods from being scheduled on certain Nodes based on labels.

3. Pod Affinity and Anti-Affinity

- Pod Affinity: Ensures Pods are scheduled close to other Pods with specific labels, useful for co-locating related workloads.
- Pod Anti-Affinity: Ensures Pods are spread out by avoiding Nodes hosting other Pods with specific labels. Example:

Let’s discuss

```yaml
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - database
      topologyKey: "kubernetes.io/hostname"
```

PW SKILLS

Introducing resource requests and limits to ensure fair scheduling across workloads.

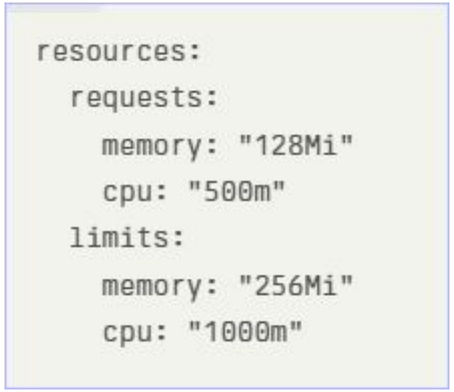
Let’s see

In Kubernetes, resource requests and limits ensure fair scheduling and resource allocation across workloads:

1. Resource Requests: Define the minimum CPU and memory a container needs to function. The Kubernetes scheduler uses these values to assign Pods to Nodes with sufficient resources, ensuring critical workloads always have their required capacity.
2. Resource Limits: Set the maximum CPU and memory a container can consume. This prevents any single container from monopolizing resources, maintaining stability for other workloads. If a container exceeds its CPU limit, it is throttled; if it exceeds its memory limit, it may be terminated (OOMKilled).

Let’s see

Example Configuration



```
resources:
  requests:
    memory: "128Mi"
    cpu: "500m"
  limits:
    memory: "256Mi"
    cpu: "1000m"
```

- Requests ensure guaranteed resources for scheduling.
- Limits cap resource usage during runtime to prevent overconsumption.

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Explaining how least privilege access minimizes attack surfaces in containers

Let’s see

The principle of least privilege access minimizes attack surfaces in containers by ensuring that users, processes, and systems are granted only the minimum permissions necessary to perform their tasks. This approach significantly reduces security vulnerabilities in the following ways:

- Reduced Attack Surface
- Mitigation of Privilege Escalation
- Isolation of Containers
- Enhanced Fault Tolerance

PW SKILLS

Introducing container image scanning tools such as Clair and Trivy to detect vulnerabilities.

Let’s see

Clair

Clair is an open-source project developed by CoreOS (now part of Red Hat) that specializes in static analysis of vulnerabilities in application containers. Key features of Clair include:

- Layer-by-layer analysis of container images.
- Utilization of multiple vulnerability databases (e.g., CVE, Debian Security Tracker).
- Integration with popular platforms like Quay.io.
- Detailed vulnerability reporting.

Let’s see

Trivy

Trivy, developed by Aqua Security, is a versatile and fast vulnerability scanner for container images, file systems, and application dependencies. Notable features of Trivy include:

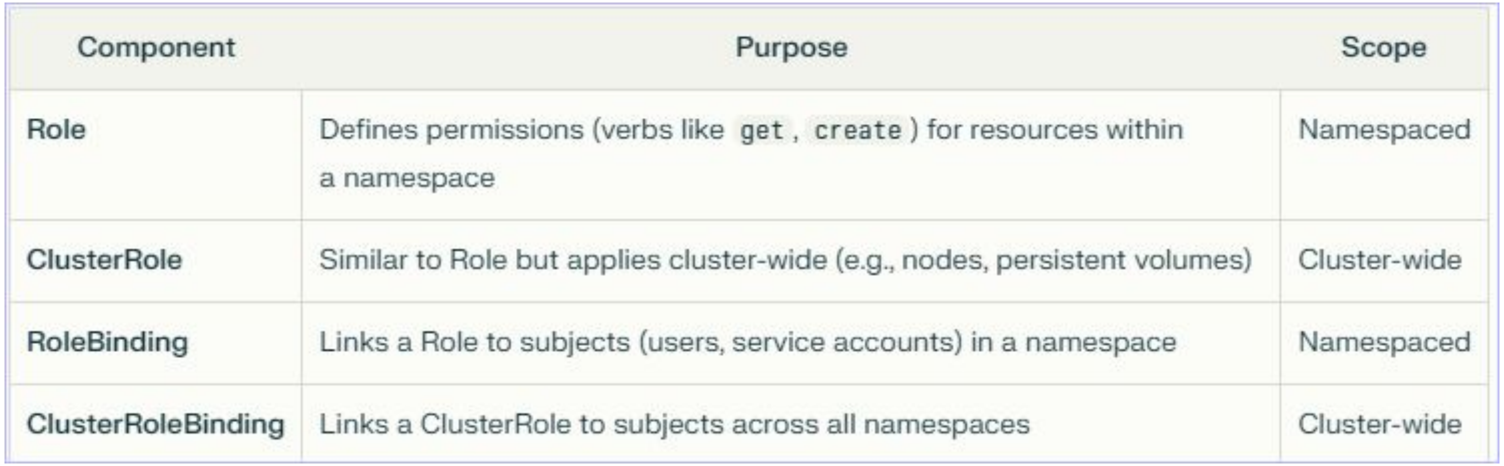
- Comprehensive scanning for vulnerabilities, misconfigurations, and secrets.
- Fast scanning speed, ideal for CI/CD pipelines.
- Simple installation and usage.
- Wide compatibility with various container registries and programming languages.

Discussing Role-Based Access Control (RBAC) and how to limit permissions in Kubernetes

Let’s discuss

Role-Based Access Control (RBAC) in Kubernetes governs permissions for users, groups, and service accounts by defining what actions they can perform on specific resources.

Key RBAC Components:



Component	Purpose	Scope
Role	Defines permissions (verbs like `get`, `create`) for resources within a namespace	Namespaced
ClusterRole	Similar to Role but applies cluster-wide (e.g., nodes, persistent volumes)	Cluster-wide
RoleBinding	Links a Role to subjects (users, service accounts) in a namespace	Namespaced
ClusterRoleBinding	Links a ClusterRole to subjects across all namespaces	Cluster-wide

Let’s discuss

Limiting Permissions: Best Practices

- Avoid cluster-admin
- Prefer RoleBindings
- Reject wildcards
- Audit third-party components
- Use groups for bulk assignments
- Restrict LIST permissions

Let’s discuss

Example Risky Configuration

```
# Dangerous: Grants cluster-admin to default service account in default namespace
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: risky-binding
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Explaining network policies for securing intra-cluster communication.

Let’s discuss

Kubernetes Network Policies secure intra-cluster communication by controlling how pods interact with each other and external endpoints. They act as internal firewalls, allowing administrators to define rules for ingress (incoming) and egress (outgoing) traffic at the pod level.

Key Features of Network Policies:

- Traffic Control
- Granular Rules
- Isolation

PW SKILLS

Demonstrating best practices for securing Kubernetes Secrets and avoiding hardcoding sensitive data.

Let’s do it

To secure Kubernetes Secrets and avoid hardcoding sensitive data, follow these best practices:

1. Use Kubernetes Secrets for Sensitive Data
2. Enable Encryption at Rest
3. Implement Role-Based Access Control (RBAC)
4. Use External Secret Management Tools
5. Rotate Secrets Regularly
6. Avoid Exposing Secrets in Logs
7. Use Short-Lived Secrets
8. Monitor and Audit Secret Access
9. Segregate Secrets

Explaining the need for automatic scaling in Kubernetes to handle varying workloads.

Let’s see

Automatic scaling in Kubernetes is essential for handling varying workloads because it ensures efficient resource utilization, optimal performance, and cost savings.

Key Benefits of Kubernetes Autoscaling:

1. Performance Optimization
2. Cost Efficiency
3. Enhanced Reliability
4. Operational Simplicity

PW SKILLS

Discussing how Horizontal Pod Autoscaler (HPA) adjusts the number of Pods based on CPU and memory usage.

Let’s discuss

The Kubernetes Horizontal Pod Autoscaler (HPA) dynamically adjusts the number of pods in a deployment, replica set, or StatefulSet based on observed metrics like CPU and memory usage.

How HPA Works:

1. Metrics Collection: HPA continuously monitors pod metrics, such as CPU utilization or custom metrics.
2. Target Comparison: It compares the observed metrics to predefined target values set by the user.

Let’s discuss

3. Scaling Decision: Using a control loop, HPA calculates the required number of pods to meet the target metric value using a formula:

i.e Desired Replicas=Current Replicas × Target Metric/Observed Metric

4. Scaling Action: The HPA updates the pod count via the Kubernetes API, scaling up when resource usage exceeds targets or scaling down during low demand.

PW SKILLS

Introducing custom metrics scaling, allowing autoscaling based on application-specific metrics.

Let’s see

Custom metrics scaling in Kubernetes allows for autoscaling based on application-specific metrics, providing more flexibility and precision in managing workloads compared to standard CPU and memory-based scaling.

Key aspects of custom metrics scaling include:

1. Metric Sources
2. Custom Metrics API
3. Horizontal Pod Autoscaler (HPA):
4. Metric Configuration
5. Scaling Behavior

PW SKILLS

Explaining how to configure autoscaling policies to balance performance and cost.

Let’s see

To balance performance and cost in Kubernetes autoscaling, you can configure autoscaling policies that optimize resource allocation while avoiding overprovisioning. Here are the best practices:

1. Use Scaling Policies

Example:

```
behavior:
  scaleUp:
    policies:
    - type: Percent
      value: 50
      periodSeconds: 60
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
```

Let’s see

2. Combine HPA and VPA

3. Enable Cluster Autoscaler

4. Set Resource Requests and Limits

5. Monitor and Optimize

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar consult the teaching assistants

Thanks !