03 K8s Deployment 101

Kubernetes Deployments: A Comprehensive Guide

Introduction to Deployments

Deployments represent the primary workload resource for managing stateless applications in production Kubernetes environments. Unlike Pods and ReplicaSets, which are typically not created directly in production settings, Deployments serve as the standard mechanism for deploying and managing applications at scale.

Architecture and Hierarchy

The Deployment resource operates through a three-tier hierarchy:

```
Deployment → ReplicaSet(s) → Pod(s)
```

Key Characteristics:

- One Deployment typically represents one microservice
- Each Deployment manages a list of ReplicaSets (which may be empty, contain one, or multiple ReplicaSets)
- Each ReplicaSet manages a collection of Pods
- At any given time, only one ReplicaSet is actively serving traffic, while others maintain a replica count of zero

Practical Example

Consider deploying an Order Service microservice built with Spring Boot that requires three instances to handle production load:

1. **Initial Deployment (Version 1):**
   - Create a Deployment YAML representing the Order Service microservice
   - Specify three replicas in the Deployment specification
   - Kubernetes automatically creates one ReplicaSet behind the scenes
   - The ReplicaSet creates three Pods running the application
2. **Deploying New Version (Version 2):**
   - Update the Deployment YAML with the new Docker image version
   - Kubernetes creates a new ReplicaSet for version 2
   - The new ReplicaSet scales up to three replicas
   - The original ReplicaSet scales down to zero replicas
   - Original Pods are terminated

This versioning mechanism enables seamless application updates and rollback capabilities.

Creating a Basic Deployment

Converting ReplicaSet to Deployment

A Deployment can be conceptualized as an extension of a ReplicaSet. The conversion is straightforward:

```yaml
apiVersion: apps/v1
kind: Deployment  # Changed from ReplicaSet
metadata:
  name: my-deploy
spec:
  selector:
    matchLabels:
      app: my-app
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: nginx
        image: nginx
```

Key Points:

- The `kind` field changes from `ReplicaSet` to `Deployment`
- All ReplicaSet properties are inherited and applicable
- The Deployment contains information necessary to create both ReplicaSets and Pods
- Additional Deployment-specific properties extend base ReplicaSet functionality

Deployment Creation and Observation

When applying a Deployment manifest:

```shell
kubectl apply -f deployment.yaml
```

Results:

- Deployment resource is created
- ReplicaSet is automatically generated with an appended hash suffix (e.g., `my-deploy-7d8f9c6b5`)
- Pods are created with unique names derived from the ReplicaSet name
- All resources become operational typically within seconds

kubectl Commands: Create vs Apply

kubectl create

The `create` command operates with the following behavior:

Syntax Options:

```shell
kubectl create -f filename.yaml
kubectl create -f .  # Creates from all YAML files in current directory
kubectl create -f https://example.com/manifest.yaml  # Creates from URL
```

Characteristics:

- Creates resources only if they do not already exist in the cluster
- Subsequent executions with existing resources result in errors
- Non-idempotent operation
- Suitable for initial resource creation

kubectl apply

The `apply` command provides declarative configuration management:

Syntax:

```shell
kubectl apply -f filename.yaml
kubectl apply -f .
```

Characteristics:

- Creates resources if they do not exist
- Updates resources when changes are detected in the manifest
- Ignores requests when no changes exist between current state and manifest
- Idempotent operation—can be executed repeatedly without errors
- Recommended for Deployment lifecycle management
- Enables continuous deployment workflows

Operational Behavior: When executing `kubectl apply` multiple times, the system responds with "configured" when changes are detected or "unchanged" when the manifest matches the current state.

Universal kubectl Command Patterns

Commands previously introduced for Pod management extend to all Kubernetes resources, including Deployments:

General Command Structure

```shell
# Retrieve resources
kubectl get deployment
kubectl get deploy  # Shorthand notation

# Retrieve specific resource
kubectl get deploy <deployment-name>

# Display labels
kubectl get deploy --show-labels

# Query by labels
kubectl get deploy -l <label-selector>

# Output in YAML format
kubectl get deploy <deployment-name> -o yaml

# Describe resource
kubectl describe deploy <deployment-name>

# View logs (from one of the Pods)
kubectl logs deploy/<deployment-name>

# Execute commands in container
kubectl exec -it deploy/<deployment-name> -- bash

# Port forwarding
kubectl port-forward deploy/<deployment-name> 8080:80
```

Practical Advantages

Using Deployment-level commands eliminates the need to track individual Pod names:

- Log retrieval automatically selects one Pod from the Deployment
- Exec commands place you in a container without specifying Pod names
- Port forwarding connects to any available Pod in the Deployment
- Particularly useful when Pod names are dynamically generated

Understanding ReplicaSets and Revisions

Revision Management

Deployments maintain multiple ReplicaSets to manage application versions:

Replica Count Changes: Modifying only the `replicas` field does not create new ReplicaSets. The existing ReplicaSet adjusts its Pod count accordingly, as this represents scaling rather than a version change.

Pod Template Changes: Modifications to the Pod template specification trigger new revision creation:

- Updating container images
- Adding or modifying environment variables
- Changing container ports
- Modifying resource requests or limits
- Updating volume configurations

When template changes occur:

1. New ReplicaSet is created with updated specifications
2. New ReplicaSet scales up to desired replica count
3. Old ReplicaSet scales down to zero
4. Old ReplicaSet is retained for rollback capabilities

Example: Version Progression

```yaml
# Initial deployment (v1)
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: order-service
        image: vinsdocker/k8s-app:v1
```

After updating the image to v2 and applying:

- New ReplicaSet created for v2
- Three new Pods running v2 image are created
- Original ReplicaSet retained with zero replicas
- Original Pods terminated

Observing ReplicaSets:

```shell
kubectl get replicasets
```

Output shows multiple ReplicaSets with different ages and current replica counts, where only the active version maintains non-zero replicas.

Rollout Management

Viewing Rollout History

```shell
kubectl rollout history deployment/<deployment-name>
```

Output Structure:

```
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
```

Recording Change Causes

To maintain meaningful rollout history, use annotations in the Deployment manifest:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service-deploy
  annotations:
    kubernetes.io/change-cause: "deploying v3"
spec:
  # ... rest of specification
```

Enhanced History Output:

```
REVISION  CHANGE-CAUSE
1         deploying v1
2         deploying v2
3         deploying v3
```

Understanding Annotations

Annotations provide metadata to Kubernetes resources and controllers:

- Reserved prefix `kubernetes.io/` is designated for Kubernetes internal use
- Custom annotations can extend resource functionality
- The `change-cause` annotation specifically records deployment reasons
- Particularly valuable in production environments for audit trails

Rollback Operations

Rolling Back to Previous Version:

```shell
kubectl rollout undo deployment/<deployment-name>
```

Behavior:

- Automatically reverts to the immediately preceding version
- Creates new revision number (does not restore old revision number)
- Activates the previous ReplicaSet by setting its replica count
- Deactivates current ReplicaSet by setting its replica count to zero

Caution: Executing multiple consecutive undo operations causes alternation between the two most recent versions rather than progressing backward through history.

Rolling Back to Specific Revision:

```shell
kubectl rollout undo deployment/<deployment-name> --to-revision=<revision-number>
```

This command provides precise control over which version to activate, preventing unintended version oscillation.

Detailed Revision Information

To examine changes introduced in a specific revision:

```shell
kubectl rollout history deployment/<deployment-name> --revision=<revision-number>
```

Output includes:

- Container image versions
- Environment variable configurations
- Port specifications
- Resource requests and limits
- All Pod template modifications

This detailed view assists in understanding the exact changes between versions, particularly valuable when multiple modifications were applied simultaneously.

Deployment Configuration Properties

minReadySeconds

The `minReadySeconds` property defines the minimum duration a Pod must be ready before being considered available:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deploy
spec:
  minReadySeconds: 10
  selector:
    matchLabels:
      app: my-app
  replicas: 3
  template:
    # ... Pod template
```

Purpose:

- Accommodates application startup time (database connections, bean initialization, resource loading)
- Prevents premature traffic routing to applications still initializing
- Provides basic readiness checking without implementing probes

Limitations: This represents a simple, time-based approach. Production environments should implement proper readiness probes for accurate health checking, which are covered in dedicated health check and probe sections.

Observable Behavior: When Pods start, they transition through states:

- Running but not Available (during minReadySeconds period)
- Running and Available (after minReadySeconds expires)

Deployment Strategies

Kubernetes provides two fundamental deployment strategies for managing version transitions.

Recreate Strategy

The Recreate strategy implements a complete replacement approach:

Configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service-deploy
spec:
  minReadySeconds: 10
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: order-service
  replicas: 3
  template:
    spec:
      containers:
      - name: order-service
        image: vinsdocker/k8s-app:v2
```

Operational Sequence:

1. All existing Pods (version 1) are terminated simultaneously
2. System waits for complete termination
3. New Pods (version 2) are created
4. New Pods initialize and become ready

Characteristics:

- Complete service interruption during transition
- No mixing of old and new versions
- Simplified state management
- Suitable for applications that cannot run multiple versions concurrently
- Faster resource transition but with downtime

Use Cases:

- Applications with database schema migrations requiring exclusive access
- Services incompatible with concurrent version operation
- Development and testing environments where downtime is acceptable

Rolling Update Strategy

Rolling Update implements gradual version transition with zero or minimal downtime:

Configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service-deploy
spec:
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: order-service
  replicas: 3
  template:
    # ... Pod template
```

maxSurge Property

The `maxSurge` parameter controls additional Pod creation during updates:

Accepts:

- Integer value (absolute number of extra Pods)
- Percentage value (relative to desired replica count)

Behavior: Defines how many Pods can run above the desired replica count during deployment.

Example with Percentage (100%):

```yaml
rollingUpdate:
  maxSurge: 100%
```

For a Deployment with 3 replicas:

1. Creates 3 additional Pods (version 2) alongside existing 3 Pods (version 1)
2. Temporarily runs 6 Pods total (double the desired count)
3. After new Pods become available, terminates old Pods
4. Returns to desired count of 3 Pods

Example with Absolute Value:

```yaml
rollingUpdate:
  maxSurge: 1
```

For a Deployment with 3 replicas:

1. Creates 1 additional Pod (version 2)
2. Runs 4 Pods temporarily
3. After new Pod is available, terminates 1 old Pod
4. Repeats process gradually until all Pods are updated
5. Never exceeds 4 Pods at any point

Strategic Considerations:

High `maxSurge` values:

- Accelerate deployment process
- Require additional cluster resources
- May strain dependent services (databases, APIs)

Production Incident Example: In a scenario with 40 replicas and `maxSurge: 100%`:

- Deployment temporarily scaled to 80 Pods
- All 80 Pods simultaneously attempted database connections
- Database connection pool exhausted (designed for 40 connections)
- New Pods failed to start due to connection rejection
- Pods entered CrashLoopBackOff state
- Resolution required 15-20 minutes of investigation and remediation

This illustrates the importance of considering downstream resource capacity when configuring deployment strategies.

maxUnavailable Property

The `maxUnavailable` parameter controls Pod termination during updates:

Accepts:

- Integer value (absolute number of Pods)
- Percentage value (relative to desired replica count)

Behavior: Defines how many Pods can be unavailable below the desired replica count during deployment.

Configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service-deploy
spec:
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 1
  selector:
    matchLabels:
      app: order-service
  replicas: 3
  template:
    # ... Pod template
```

Operational Sequence: For a Deployment with 3 replicas:

1. Terminates 1 existing Pod (leaving 2 available)
2. Creates 1 new Pod with updated version
3. Waits for new Pod to become ready
4. Once ready, terminates another old Pod
5. Creates another new Pod
6. Repeats until all Pods are updated

Key Distinction from maxSurge:

- `maxUnavailable` does not create extra Pods initially
- Maintains or reduces Pod count during transition
- Prevents resource capacity issues with dependent services
- Suitable when additional resource consumption must be avoided

Capacity Management: With `maxUnavailable` set, the available Pod count during deployment ranges from:

- Minimum: `replicas - maxUnavailable`
- Maximum: `replicas`

For 3 replicas with `maxUnavailable: 1`:

- Minimum available: 2 Pods
- Maximum available: 3 Pods

Strategic Selection:

Use `maxSurge` when:

- Minimizing service degradation is critical
- Cluster resources are abundant
- Dependent services can handle increased load

Use `maxUnavailable` when:

- Resource constraints exist
- Dependent services have connection limits
- Gradual capacity reduction is acceptable
- Cost optimization is important

Practical Implementation Example

Application Architecture

Consider a Spring Boot application with Redis dependency:

Components:

1. Application Pod
    - Image: `vinsdocker/k8s-deploy-assignment`
    - Exposes port 8080
    - Requires Redis connection details
2. Redis Pod
    - Image: `redis:6`
    - Exposes port 6379
    - Single replica (in-memory cache)

Redis Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:6
        ports:
        - containerPort: 6379
          name: "redis-port"
```

Deployment Justification: While Redis is stateful, using a Deployment is acceptable for:

- In-memory caching scenarios
- Development and testing environments
- Situations where data loss is acceptable
- Single-replica configurations

Production stateful applications should use StatefulSets, covered in advanced topics.

Application Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: assignment
spec:
  selector:
    matchLabels:
      app: assignment
  replicas: 1
  template:
    metadata:
      labels:
        app: assignment
    spec:
      containers:
      - name: assignment
        image: vinsdocker/k8s-deploy-assignment
        ports:
        - containerPort: 8080
          name: "app-port"
        env:
        - name: "spring.redis.host"
          value: "10.244.1.2"  # Pod IP address - temporary solution
        - name: "spring.redis.port"
          value: "6379"
```

Important Note: Using direct Pod IP addresses represents a temporary approach for demonstration purposes. This method has significant limitations:

- Pod IPs change when Pods are recreated
- No automatic service discovery
- Requires manual configuration updates

Production environments should use Kubernetes Services for stable networking, which provide:

- Stable virtual IP addresses
- DNS-based service discovery
- Load balancing across Pods
- Automatic endpoint management

Deployment Sequence

```shell
# Deploy Redis first
kubectl apply -f redis-deployment.yaml

# Retrieve Redis Pod IP
kubectl get pod -o wide

# Update application manifest with Redis IP
# Then deploy application
kubectl apply -f app-deployment.yaml

# Verify deployments
kubectl get deployments

# Access application
kubectl port-forward deploy/assignment 8080:8080
```

This example demonstrates basic Deployment usage while highlighting the need for proper service networking covered in subsequent sections.

Summary of Key Concepts

Deployment Hierarchy

- **Deployment** represents one microservice or application
- **ReplicaSet** represents each version deployed
- **Pod** represents the actual running application instance

Resource Relationships

- Deployments manage ReplicaSets
- ReplicaSets manage Pods
- Only one ReplicaSet actively serves traffic at any time
- Historical ReplicaSets are retained with zero replicas for rollback capability

Version Management

- Updating Pod template specifications creates new ReplicaSets
- Updating replica counts modifies existing ReplicaSet
- Each ReplicaSet corresponds to one application revision
- Revision history enables rapid rollback operations

Command Patterns

All kubectl commands applicable to Pods extend to Deployments:

- `kubectl get deploy`

- `kubectl describe deploy <name>`
- `kubectl logs deploy/<name>`
- `kubectl exec -it deploy/<name> -- <command>`
- `kubectl port-forward deploy/<name> <local-port>:<container-port>`

Rollout Operations

View History:

```shell
kubectl rollout history deployment/<name>
kubectl rollout history deployment/<name> --revision=<number>
```

Rollback:

```shell
kubectl rollout undo deployment/<name>  # Previous version
kubectl rollout undo deployment/<name> --to-revision=<number>  # Specific version
```

Deployment Strategies

Recreate:

- Complete replacement strategy
- Service interruption during transition
- No version mixing
- Use when versions cannot coexist

Rolling Update:

- Gradual transition strategy
- Zero or minimal downtime
- Controlled by `maxSurge` and `maxUnavailable`
- Suitable for most production scenarios

Configuration Best Practices

1. **Always use `kubectl apply`** for Deployment management
2. **Record change causes** using annotations for audit trails
3. **Configure minReadySeconds** to accommodate application startup
4. **Choose appropriate deployment strategy** based on application requirements
5. **Consider resource constraints** when setting maxSurge values
6. **Test rollback procedures** before production deployment
7. **Use Services for networking** rather than direct Pod IPs
8. **Monitor dependent service capacity** when scaling or deploying

Production Considerations

- Evaluate downstream service capacity before aggressive rollout strategies

- Implement proper health checks (readiness and liveness probes)
- Maintain detailed revision history through annotations
- Test rollback procedures in non-production environments
- Consider resource quotas and limits in multi-tenant clusters
- Plan for gradual rollouts using appropriate maxSurge and maxUnavailable values

Deployments provide robust, production-ready mechanisms for managing application lifecycles in Kubernetes, offering powerful version control, rollback capabilities, and flexible deployment strategies that accommodate diverse operational requirements.