

Stratégies de Déploiement dans Kubernetes

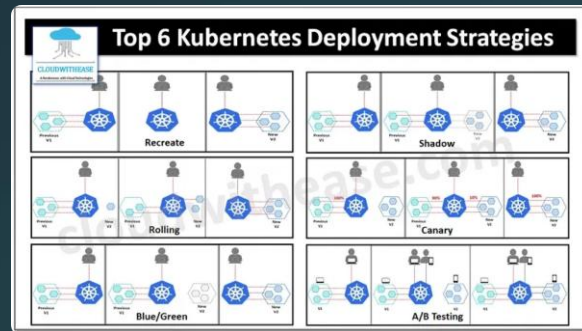
Guide complet pour des déploiements efficaces et sécurisés

Octobre 2025

Introduction aux Stratégies de Déploiement

Pourquoi des stratégies de déploiement ?

- ✓ Minimiser les temps d'arrêt
- ✓ Réduire les risques lors des mises à jour
- ✓ Permettre des tests en production
- ✓ Faciliter le rollback en cas de problème



Vue d'ensemble des stratégies

Stratégie	Disponibilité	Risque	Cas d'utilisation
↻ Rolling Update	✓ Élevée	● Faible	Applications critiques
↔ Blue-Green	✓ Élevée	● Faible	Production stable
🐦 Canary	✓ Élevée	● Moyen	Testing progressif
🔄 Recreate	✗ Interruption	● Élevé	Migrations breaking
🏗️ A/B Testing	✓ Élevée	● Faible	Tests utilisateurs
👻 Shadow	✓ Élevée	● Faible	Tests en production

Rolling Update

Concept

La stratégie **Rolling Update** est la méthode de déploiement par défaut dans Kubernetes. Elle met à jour progressivement les pods un par un.

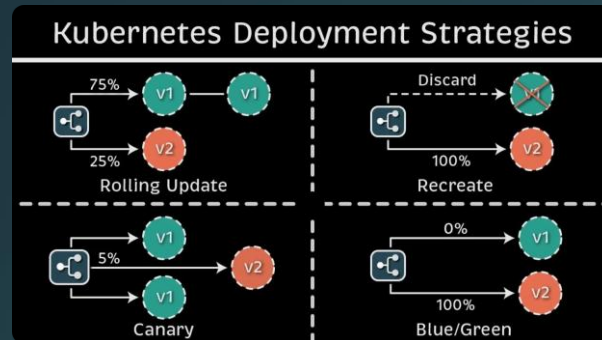
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

Paramètres clés

- **maxSurge** : Pods supplémentaires autorisés
- **maxUnavailable** : Pods indisponibles autorisés

Avantages

- ✓ Zéro downtime
- ✓ Rollback automatique
- ✓ Mise à jour progressive
- ✓ Stratégie par défaut



Commandes utiles

```
# Surveiller un déploiement
kubectl rollout status deployment/my-app

# Rollback
kubectl rollout undo deployment/my-app
```

Inconvénients

- ✗ Version mixte temporaire
- ✗ Complexe pour les migrations DB
- ✗ Tests de compatibilité nécessaires

Blue-Green Deployment

Concept

Deux environnements identiques fonctionnent en parallèle :

Blue : Version actuelle en production

Green : Nouvelle version à déployer

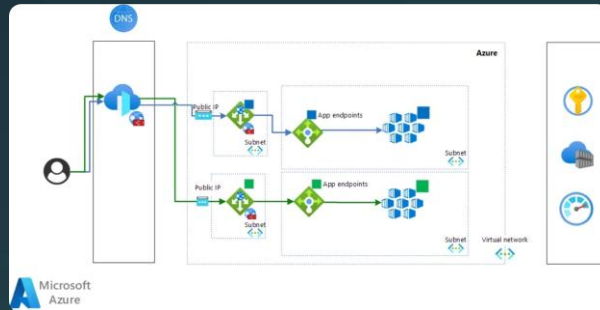
Le trafic est basculé instantanément de Blue vers Green après validation.

+ Avantages

- Bascule instantanée
- Rollback très rapide
- Pas de version mixte

- Inconvénients


- Coût en ressources double
- Complexité opérationnelle



Implémentation

```
---
# Version Blue (v1.0)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: "blue"
  template:
    metadata:
      labels:
        app: my-app
        version: "blue"
    spec:
      containers:
        - name: app
          image: my-app:v1.0
```

```
---
# Version Green (v2.0)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: "green"
  template:
    metadata:
      labels:
        app: my-app
        version: "green"
    spec:
      containers:
        - name: app
          image: my-app:v2.0
```

```
---
# Service qui bascule
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: "blue" #  Basculer vers "green" pour déployer
  ports:
    - port: 80
      targetPort: 8080
```

Canary Deployment

🐦 Concept

Le déploiement **Canary** expose progressivement une nouvelle version à un petit pourcentage d'utilisateurs.

Fonctionnement

90% du trafic vers la version stable

10% du trafic vers la nouvelle version

Augmentation progressive si les métriques sont bonnes

Avantages

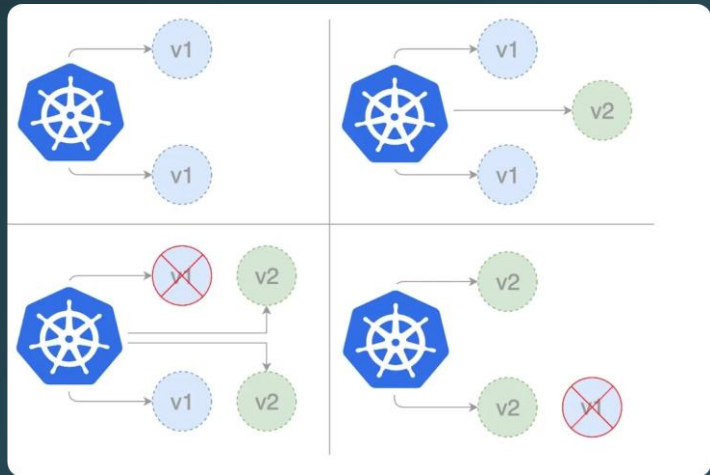
Testing en production sécurisé

Limitation des risques

Inconvénients

Complexité de routage

Monitoring avancé requis



Canary Deployment

Implémentation avec Ingress

```
# Deployment stable (90% du trafic)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-stable
spec:
  replicas: 9
  selector:
    matchLabels:
      app: my-app
      track: stable
  template:
    metadata:
      labels:
        app: my-app
        track: stable
    spec:
      containers:
        - name: app
          image: my-app:v1.0
```

```
---
# Deployment canary (10% du trafic)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      track: canary
  template:
    metadata:
      labels:
        app: my-app
        track: canary
    spec:
      containers:
        - name: app
          image: my-app:v2.0
```

```
# Service avec sélecteur multiple
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app # ➡ Route vers stable ET canary
  ports:
    - port: 80
      targetPort: 8080
```

Avec Ingress pour contrôle fin :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "10" # 10% vers canary
spec:
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-app-service
                port:
                  number: 80
```

Recreate Strategy

Concept ⚠

La stratégie **Recreate** est la plus simple mais la plus risquée : elle arrête complètement l'ancienne version avant de démarrer la nouvelle.

Fonctionnement

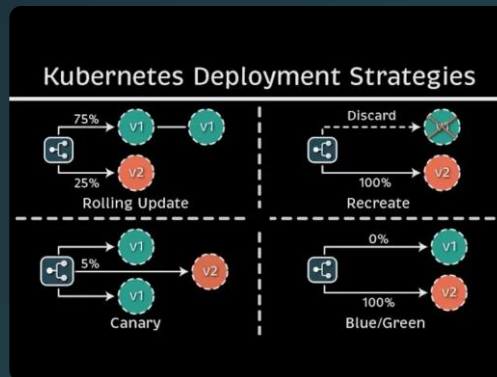
Arrêt complet de tous les pods de l'ancienne version

Période de downtime (indisponibilité du service)

Démarrage des pods de la nouvelle version

Arrêt complet puis redémarrage.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  strategy:
    type: Recreate # ⚠ Stop tout puis redémarre
  template:
    spec:
      containers:
        - name: app
          image: my-app:v2.0
```



Cas d'utilisation

Migrations de base de données
Environnements de test

Changements d'architecture
Applications non critiques

Inconvénients

- ✗ Downtime obligatoire
- ✗ Pas de rollback facile
- ✗ Risque élevé en production
- ✗ Impact utilisateur

Avantages

- ✓ Simple à mettre en œuvre
- ✓ Pas de version mixte
- ✓ Idéal pour migrations DB
- ✓ Ressources optimisées

A/B Testing et Shadow Deployment

A/B Testing

Déploiement basé sur des règles métier pour tester différentes versions auprès de segments d'utilisateurs spécifiques.

Avantages

- Tests utilisateurs réels
- Décision data-driven

Inconvénients

- Complexité avancée
- Analytics requis

Shadow Deployment

Copie du trafic live vers la nouvelle version sans impact sur les utilisateurs.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-app
spec:
  http:
  - route:
    - destination:
        host: my-app-v1
      mirror:
        host: my-app-v2 # Copie le trafic vers v2
      mirror_percent: 100
```

Avantages

- Testing sans impact
- Données réelles

Inconvénients

- Complexité élevée
- Coût supplémentaire

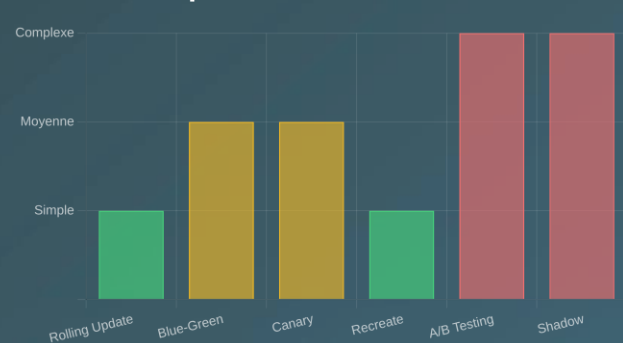
Tableau Comparatif des Stratégies

Stratégie	Downtime	Risque	Complexité	Rollback	Use Case
↻ Rolling Update	✖ Aucun	● Faible	● Simple	● Rapide	Applications standards
↔ Blue-Green	✖ Aucun	● Faible	● Moyenne	● Instant	Production critique
🐦 Canary	✖ Aucun	● Moyen	● Moyenne	● Rapide	Nouveaux features
🔄 Recreate	✔ Oui	● Élevé	● Simple	● Difficile	Migrations DB
🧪 A/B Testing	✖ Aucun	● Faible	● Complexe	● Rapide	Tests utilisateurs
👻 Shadow	✖ Aucun	● Faible	● Complexe	● Facile	Performance testing

Comparaison des risques



Comparaison de la complexité



Outils Recommandés

Outils natifs Kubernetes



kubectl rollout

Gestion des déploiements et rollbacks natifs



Services + Ingress

Routage et équilibrage de charge pour Blue-Green/Canary



ConfigMaps + Secrets

Configuration externalisée pour faciliter les mises à jour

Commandes essentielles

```
# Surveiller un déploiement
kubectl rollout status deployment/my-app

# Historique des déploiements
kubectl rollout history deployment/my-app

# Rollback
kubectl rollout undo deployment/my-app
```

Outils avancés



Istio

Service mesh pour Canary, Shadow et A/B testing avancés



Argo Rollouts

Contrôle avancé des déploiements progressifs



Flagger

Automatisation des déploiements progressifs



Prometheus

Monitoring pour décisions data-driven

Critères de choix

Taille de l'équipe et expertise disponible

Complexité des applications déployées

Besoins en automatisation et intégration CI/CD

Exigences de monitoring et d'observabilité

Conclusion

Points clés à retenir

- ✓ **Pas de solution universelle** : Chaque stratégie répond à des besoins spécifiques
- ✓ **Évaluer les critères** : Disponibilité, risque, complexité et facilité de rollback
- ✓ **Considérer le contexte** : Type d'application, environnement et criticité
- ✓ **Automatiser** : Utiliser des outils spécialisés pour simplifier les déploiements
- ✓ **Tester** : Valider la stratégie dans des environnements non-critiques avant la production

Recommandation générale

Commencez avec **Rolling Update** (stratégie par défaut) pour la plupart des applications, puis évoluez vers des stratégies plus avancées selon vos besoins spécifiques de fiabilité et de test.

Choisir la bonne stratégie



Rolling Update

Applications standards



Blue-Green

Production critique



Canary

Nouveaux features



Recreate

Migrations DB



A/B Testing

Tests utilisateurs



Shadow

Performance testing

Outils recommandés

✂ kubectl rollout

✂ Istio

✂ Flagger

✂ Argo Rollouts

"Choisissez votre stratégie selon : criticité de l'application, tolérance au risque, et complexité opérationnelle !"