Complete ArgoCD Tutorial for Kubernetes Beginners

Table of Contents

What is ArgoCD and Why Use It? {#what-is-argocd}

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. Think of it as your deployment automation assistant that:

- **Watches your Git repository** for changes
- **Automatically syncs** your Kubernetes cluster with what's defined in Git
- **Provides a beautiful UI** to visualize your applications
- **Ensures your cluster matches** your Git repository (desired state)

The GitOps Philosophy

- **Git as Single Source of Truth**: All your Kubernetes manifests live in Git
- **Declarative**: You describe what you want, not how to get there
- **Automated**: Changes in Git automatically trigger deployments
- **Auditable**: Every change is tracked in Git history

Prerequisites Setup {#prerequisites}

Since you have `kubectl` and `minikube` already installed, let's verify and set up additional tools:

1. Verify Your Current Setup

```bash
# Check kubectl
kubectl version --client

# Check minikube
minikube version

# Check if minikube is running
minikube status
```

2. Install Additional Tools

```bash
# Install ArgoCD CLI (on macOS)
brew install argocd

# Install git (if not already installed)
brew install git

# Install curl (usually pre-installed)
curl --version
```

3. Start Minikube with Sufficient Resources

```bash
# Stop minikube if running
minikube stop

# Start with more resources for ArgoCD
minikube start --memory=4096 --cpus=2 --driver=docker

# Verify cluster is ready
kubectl get nodes
```

Lab Environment Setup {#lab-setup}

We'll create a realistic scenario where you're deploying a simple web application using GitOps principles.

Scenario Overview

You're a DevOps engineer tasked with:

1. Setting up ArgoCD in your Kubernetes cluster

2. Deploying a sample web application
3. Implementing automated deployments via Git
4. Managing multiple environments (dev, staging)

Create Project Structure

```bash
# Create workspace
mkdir -p ~/argocd-tutorial
cd ~/argocd-tutorial

# Create directory structure
mkdir -p {apps,environments/{dev,staging},argocd-config}

# Your structure should look like:
# ~/argocd-tutorial/
# ├── apps/
# ├── environments/
# │   ├── dev/
# │   └── staging/
# └── argocd-config/
```

Installing ArgoCD {#installing-argocd}

Step 1: Create ArgoCD Namespace

```bash
# Create dedicated namespace for ArgoCD
kubectl create namespace argocd
```

Step 2: Install ArgoCD

```bash
# Install ArgoCD in the argocd namespace
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/m

# Wait for all pods to be ready (this may take 2-3 minutes)
kubectl wait --for=condition=ready pod --all -n argocd --timeout=300s

# Verify installation
kubectl get pods -n argocd
```

You should see pods like:

- `argocd-application-controller`
- `argocd-dex-server`
- `argocd-redis`
- `argocd-repo-server`
- `argocd-server`

Step 3: Access ArgoCD UI

Method 1: Port Forward (Recommended for Tutorial)

```bash
# Forward ArgoCD server port to localhost
kubectl port-forward svc/argocd-server -n argocd 8080:443

# Keep this terminal open and open a new terminal for other commands
```

Method 2: Using Minikube Service (Alternative)

```bash
# Patch the service to NodePort
kubectl patch svc argocd-server -n argocd -p '{"spec":{"type":"NodePort"}}'

# Get the URL
minikube service argocd-server -n argocd --url
```

Step 4: Get ArgoCD Admin Password

```bash
# Get the initial admin password
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}'

# Save this password - you'll need it to login
```

Step 5: Login to ArgoCD

1. Open browser to `https://localhost:8080` (ignore SSL warning)
2. Username: `admin`
3. Password: (the password from step 4)

🎉 **Checkpoint**: You should now see the ArgoCD dashboard!

Your First Application Deployment {#first-deployment}

Now let's deploy your first application using ArgoCD. We'll use a simple nginx web server.

Step 1: Create Application Manifests

Create your first Kubernetes application:

bash

```bash
# Create nginx deployment manifest
cat > ~/argocd-tutorial/apps/nginx-deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
  type: NodePort
EOF
```

Step 2: Initialize Git Repository

```bash
cd ~/argocd-tutorial

# Initialize git repository
git init
git add .
git commit -m "Initial commit with nginx app"

# For this tutorial, we'll use a local git repo
# In real scenarios, you'd push to GitHub/GitLab
```

Step 3: Create ArgoCD Application

You can create an ArgoCD application via UI or CLI. Let's use CLI first:

```bash
# Create ArgoCD application
argocd app create nginx-app \
  --repo file://$(pwd) \
  --path apps \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default

# Login to ArgoCD CLI first
argocd login localhost:8080 --username admin --password $(kubectl -n argocd get secret

# Now create the app
argocd app create nginx-app \
  --repo file://$(pwd) \
  --path apps \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default
```

Step 4: Sync the Application

```bash
# Sync the application (deploy it)
argocd app sync nginx-app

# Check status
argocd app get nginx-app

# Watch the deployment
kubectl get pods -w
```

Step 5: Access Your Application

```bash
# Get the service URL
minikube service nginx-service --url

# Or use port forwarding
kubectl port-forward svc/nginx-service 8081:80
```

Visit the URL and you should see the nginx welcome page!

🎉 **Checkpoint**: You've successfully deployed your first application with ArgoCD!

Understanding GitOps Workflow {#gitops-workflow}

Now let's understand the GitOps workflow by making changes and seeing ArgoCD automatically sync them.

Scenario: Update Nginx Version

Let's simulate a real-world scenario where you need to update the nginx version.

Step 1: Make Changes in Git

```bash
# Edit the deployment to update nginx version
sed -i '' 's/nginx:1.21/nginx:1.22/g' ~/argocd-tutorial/apps/nginx-deployment.yaml

# Commit the change
git add apps/nginx-deployment.yaml
git commit -m "Update nginx to version 1.22"
```

Step 2: Configure Auto-Sync (Optional)

```bash
# Enable auto-sync for the application
argocd app set nginx-app --sync-policy automated

# Or do it manually
argocd app sync nginx-app
```

Step 3: Monitor the Update

```bash
# Watch the rollout
kubectl rollout status deployment/nginx-app

# Check the image version
kubectl describe deployment nginx-app | grep Image

# View in ArgoCD UI
# Go to https://localhost:8080 and click on nginx-app
```

Step 4: Understanding ArgoCD States

In the ArgoCD UI, you'll see different states:

- **Synced**: Git repo matches cluster state
- **OutOfSync**: Git repo differs from cluster state
- **Healthy**: Application is running properly
- **Progressing**: Deployment is in progress
- **Degraded**: Application has issues

Advanced Scenarios {#advanced-scenarios}

Scenario 1: Multi-Environment Deployment

Let's set up separate dev and staging environments.

Step 1: Create Environment-Specific Manifests

bash

```
# Create dev environment configuration
cat > ~/argocd-tutorial/environments/dev/kustomization.yaml << 'EOF'
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ../../apps

patchesStrategicMerge:
- replica-patch.yaml

namePrefix: dev-
namespace: dev
EOF

# Create dev-specific patches
cat > ~/argocd-tutorial/environments/dev/replica-patch.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  replicas: 1
EOF

# Create staging environment
cat > ~/argocd-tutorial/environments/staging/kustomization.yaml << 'EOF'
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ../../apps

patchesStrategicMerge:
- replica-patch.yaml

namePrefix: staging-
namespace: staging
EOF

cat > ~/argocd-tutorial/environments/staging/replica-patch.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  replicas: 3
EOF
```

Step 2: Create Namespaces

```bash
kubectl create namespace dev
kubectl create namespace staging
```

Step 3: Create ArgoCD Applications for Each Environment

```bash
# Create dev application
argocd app create nginx-dev \
  --repo file://$(pwd) \
  --path environments/dev \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace dev \
  --sync-policy automated

# Create staging application
argocd app create nginx-staging \
  --repo file://$(pwd) \
  --path environments/staging \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace staging \
  --sync-policy automated

# Sync both applications
argocd app sync nginx-dev
argocd app sync nginx-staging
```

Step 4: Verify Different Configurations

```bash
# Check dev (should have 1 replica)
kubectl get deployments -n dev

# Check staging (should have 3 replicas)
kubectl get deployments -n staging
```

Scenario 2: Application of Applications Pattern

This pattern allows you to manage multiple applications with a single ArgoCD application.

Step 1: Create App of Apps Configuration

```bash
mkdir -p ~/argocd-tutorial/argocd-apps

cat > ~/argocd-tutorial/argocd-apps/dev-apps.yaml << 'EOF'
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-dev
  namespace: argocd
spec:
  project: default
  source:
    repoURL: file://$(pwd)
    targetRevision: HEAD
    path: environments/dev
  destination:
    server: https://kubernetes.default.svc
    namespace: dev
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-staging
  namespace: argocd
spec:
  project: default
  source:
    repoURL: file://$(pwd)
    targetRevision: HEAD
    path: environments/staging
  destination:
    server: https://kubernetes.default.svc
    namespace: staging
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
EOF
```

Step 2: Create the App of Apps

```bash
argocd app create dev-apps \
  --repo file://$(pwd) \
  --path argocd-apps \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace argocd \
  --sync-policy automated

argocd app sync dev-apps
```

Scenario 3: Handling Secrets and ConfigMaps

Let's add configuration to our application.

Step 1: Create ConfigMap and Secret

```bash
cat > ~/argocd-tutorial/apps/nginx-config.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  default.conf: |
    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html;
            index index.html index.htm;
        }

        location /health {
            access_log off;
            return 200 "healthy\n";
            add_header Content-Type text/plain;
        }
    }
---
apiVersion: v1
kind: Secret
metadata:
  name: nginx-secret
type: Opaque
data:
  # echo -n "my-secret-value" | base64
  secret-key: bXktc2VjcmV0LXZhbHVl
EOF
```

Step 2: Update Deployment to Use ConfigMap

bash

```
# Update the deployment to mount the ConfigMap
cat > ~/argocd-tutorial/apps/nginx-deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.22
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-config
          mountPath: /etc/nginx/conf.d
        env:
        - name: SECRET_KEY
          valueFrom:
            secretKeyRef:
              name: nginx-secret
              key: secret-key
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
      volumes:
      - name: nginx-config
```

```
        configMap:
          name: nginx-config
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
  type: NodePort
EOF
```

Step 3: Commit and Sync

```bash
git add .
git commit -m "Add ConfigMap, Secret, and health checks"

# Sync the application
argocd app sync nginx-app
```

Best Practices {#best-practices}

1. Repository Structure

```
├── applications/           # ArgoCD application definitions
├── environments/
│   ├── dev/
│   ├── staging/
│   └── production/
├── base/                  # Base Kubernetes manifests
└── components/            # Reusable components
```

2. Security Best Practices

```bash
# Create separate ArgoCD projects for different teams/environments
argocd proj create dev-team \
  --description "Development team project" \
  --src "https://github.com/your-org/dev-apps.git" \
  --dest "https://kubernetes.default.svc,dev-*"

# Use RBAC to control access
kubectl create rolebinding dev-team-binding \
  --clusterrole=edit \
  --serviceaccount=argocd:argocd-application-controller \
  --namespace=dev
```

3. Monitoring and Alerts

```bash
# Enable ArgoCD notifications
kubectl create secret generic argocd-notifications-secret \
  --from-literal=slack-token=YOUR_SLACK_TOKEN \
  -n argocd

# Create notification configuration
cat > argocd-notifications-cm.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-notifications-cm
  namespace: argocd
data:
  service.slack: |
    token: $slack-token
  template.app-deployed: |
    message: Application {{.app.metadata.name}} is now running new version.
  trigger.on-deployed: |
    - when: app.status.operationState.phase in ['Succeeded'] and app.status.health.sta
      send: [app-deployed]
EOF

kubectl apply -f argocd-notifications-cm.yaml
```

4. Backup and Disaster Recovery

```bash
# Backup ArgoCD applications
argocd app list -o yaml > argocd-apps-backup.yaml

# Export ArgoCD settings
kubectl get configmap argocd-cm -n argocd -o yaml > argocd-config-backup.yaml
kubectl get secret argocd-secret -n argocd -o yaml > argocd-secret-backup.yaml
```

Troubleshooting {#troubleshooting}

Common Issues and Solutions

1. Application Stuck in "OutOfSync" State

```bash
# Check application status
argocd app get your-app-name

# Force refresh
argocd app get your-app-name --refresh

# Hard refresh (bypass cache)
argocd app get your-app-name --hard-refresh

# Manual sync
argocd app sync your-app-name --prune
```

2. ArgoCD Server Not Accessible

```bash
# Check ArgoCD server pod
kubectl get pods -n argocd | grep argocd-server

# Check service
kubectl get svc -n argocd argocd-server

# Check logs
kubectl logs -n argocd deployment/argocd-server
```

3. Applications Not Syncing Automatically

```bash
# Check if auto-sync is enabled
argocd app get your-app-name | grep "Sync Policy"

# Enable auto-sync
argocd app set your-app-name --sync-policy automated

# Check ArgoCD application controller logs
kubectl logs -n argocd deployment/argocd-application-controller
```

4. Permission Issues

```bash
# Check ArgoCD service account permissions
kubectl auth can-i create deployments --as=system:serviceaccount:argocd:argocd-applica

# Check ArgoCD RBAC configuration
kubectl get configmap argocd-rbac-cm -n argocd -o yaml
```

Useful Commands for Debugging

```bash
# ArgoCD CLI commands
argocd app list                          # List all applications
argocd app get APP_NAME                  # Get application details
argocd app logs APP_NAME                 # Get application logs
argocd app diff APP_NAME                 # Show differences between Git and cluster
argocd app history APP_NAME              # Show deployment history
argocd app rollback APP_NAME ID          # Rollback to specific deployment

# Kubernetes debugging commands
kubectl get events --sort-by=.metadata.creationTimestamp
kubectl describe deployment DEPLOYMENT_NAME
kubectl logs -f deployment/DEPLOYMENT_NAME
kubectl get pods -o wide
```

Next Steps

Congratulations! You've completed the ArgoCD tutorial. Here's what you should explore next:

1. **Set up a real Git repository** (GitHub/GitLab) instead of local files
2. **Implement proper CI/CD pipeline** that updates manifests
3. **Explore ArgoCD Image Updater** for automatic image updates

4. **Learn about ArgoCD ApplicationSets** for managing apps across multiple clusters
5. **Implement proper secret management** with tools like Sealed Secrets or External Secrets Operator
6. **Set up monitoring and observability** with Prometheus and Grafana
7. **Practice with Helm charts** integration with ArgoCD

Cleanup

When you're done with the tutorial:

```bash
# Delete applications
argocd app delete nginx-app
argocd app delete nginx-dev
argocd app delete nginx-staging

# Delete ArgoCD
kubectl delete namespace argocd

# Stop minikube
minikube stop
```

Remember: GitOps is a journey, not a destination. Start small, iterate, and gradually add complexity as your team becomes comfortable with the workflow. The key is consistency and treating your Git repository as the single source of truth for your infrastructure and applications.