Devops Shack

MASTERING ARGOCD

BY DEVOPS SHACK

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Table of Contents

1. Introduction to ArgoCD

What is ArgoCD?

ArgoCD (Argo Continuous Delivery) is a declarative, GitOps-based continuous delivery tool for Kubernetes. It enables developers and DevOps teams to manage and deploy Kubernetes resources using Git repositories as the single source of truth. With ArgoCD, any changes committed to Git can automatically be synchronized to the target Kubernetes cluster.

Why Use ArgoCD?

ArgoCD brings several benefits to Kubernetes-based environments:

- **Declarative GitOps Workflow**: Define the desired state of your infrastructure and applications in Git. ArgoCD continuously monitors and syncs that state to your Kubernetes cluster.
- **Automatic Synchronization**: Automatically apply changes from your Git repo to Kubernetes without manual kubectl commands.
- **Version Control**: Rollback or audit deployments through Git commit history.
- **Multi-cluster Support**: Manage multiple Kubernetes clusters from a single ArgoCD instance.
- **Web UI and CLI Access**: Use the intuitive ArgoCD UI or CLI (argocd) to manage and monitor deployments.

Key Features

- **Application Lifecycle Management**: Handles creation, updating, and deletion of Kubernetes resources.
- **Health Monitoring**: Built-in resource health checks.
- **Self-healing**: Detects and reverts out-of-sync resources automatically if enabled.
- **RBAC and SSO Support**: Secure and configurable access control.
- **Integration Friendly**: Works well with Helm, Kustomize, K8s manifests, and custom plugins.

Use Cases

- Continuous delivery in Kubernetes environments
- GitOps-based infrastructure management
- Safe and automated rollback using Git history
- Multi-team and multi-environment deployments (dev/staging/prod)

Comparison with Traditional CI/CD

Feature	Traditional CI/CD	ArgoCD (GitOps)
Source of truth	CI server	Git repository
Deployment trigger	CI events/scripts	Git changes
State storage	Ephemeral	Git-managed
Rollback method	Manual	Git revert
Sync monitoring	Limited	Continuous

2. Installation and Setup

This section walks you through installing ArgoCD on a Kubernetes cluster and accessing the ArgoCD UI and CLI.

Prerequisites

Before installing ArgoCD, ensure the following:

- A Kubernetes cluster (minikube, kind, EKS, GKE, etc.)
- kubectl installed and configured
- helm (optional but useful)
- argocd CLI (we'll install it shortly)

Step 1: Install ArgoCD

You can install ArgoCD using either raw manifests or Helm. Here's the simplest way using kubectl:

```
# Create the ArgoCD namespace
kubectl create namespace argocd

# Install ArgoCD core components
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

🔍 This installs all ArgoCD components like the API server, UI, repo server, and controller.

Step 2: Expose the ArgoCD API Server

By default, ArgoCD runs internally. To access the UI or CLI from your local machine, expose it using a port-forward:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Now open your browser and go to https://localhost:8080

Step 3: Get the Initial Admin Password

The initial login username is admin. To get the default password (which is the pod name):

```
kubectl get secret argocd-initial-admin-secret -n argocd \
  -o jsonpath="{.data.password}" | base64 -d && echo
```

Step 4: Install the ArgoCD CLI

Download the CLI for your operating system. For Linux/macOS:

```
# macOS
brew install argocd


# or using curl
VERSION=$(curl -s https://api.github.com/repos/argoproj/argo-cd/releases/latest | grep tag_name | cut -d '"' -f 4)
curl -sSL -o argocd "https://github.com/argoproj/argo-cd/releases/download/${VERSION}/argocd-linux-amd64"
chmod +x argocd
sudo mv argocd /usr/local/bin/
```

You can check it installed correctly with:

```
argocd version
```

Step 5: Login to ArgoCD via CLI

```
argocd login localhost:8080 --username admin --password <password> --insecure
```

--insecure skips TLS verification when using localhost.

Optional: Install Using Helm

```
helm repo add argo https://argoproj.github.io/argo-helm
```

```
helm repo update
```

```
helm install argocd argo/argo-cd -n argocd --create-namespace
```

Verify Installation

To confirm all components are up and running:

```
kubectl get pods -n argocd
```

You should see pods like argocd-server, argocd-repo-server, argocd-application-controller, etc.

3. Connecting ArgoCD to Git Repositories

ArgoCD uses Git repositories as the **source of truth** for your application manifests. Once a repository is connected, ArgoCD continuously monitors it and syncs any changes to your Kubernetes cluster.

Step 1: Add a Git Repository to ArgoCD

You can register both public and private Git repositories. Here's how to add one using the CLI:

For a Public Repository:

```
argocd repo add https://github.com/<your-org>/<your-repo>.git
```

For a Private Repository:

You need to provide credentials (username/password or SSH key).

Example using HTTPS with username/password or token:

```
argocd repo add https://github.com/<your-org>/<your-private-repo>.git \
 --username your-username \
 --password your-password
```

Example using SSH:

1. Ensure your local SSH key is registered with your Git provider.
2. Add the SSH repo:

```
argocd repo add git@github.com:<your-org>/<your-repo>.git \
 --ssh-private-key-path ~/.ssh/id_rsa
```

Use --insecure-skip-server-verification if your Git server uses a self-signed certificate.

Step 2: Verify the Repository Was Added

```
argocd repo list
```

Output

example:

```
TYPE  NAME                    REPO                    INSECURE
```

```
git <your-repo-name> https://github.com/<your-org>/<your-repo>.git false
```

Step 3: File Structure of Git Repository

ArgoCD supports applications defined using:

- **Raw YAML manifests**
- **Helm charts**
- **Kustomize directories**
- **Jsonnet**
- Or even a mix using **App of Apps**

Example of a valid repo structure using Kustomize:

```
.
├── base
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
└── overlays
    └── production
        ├── kustomization.yaml
        └── values-prod.yaml
```

Step 4: Application Manifest in Git (Optional)

You can optionally store the ArgoCD Application CR itself in Git, which can be used to bootstrap itself:

```
# app.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
```

```
  name: my-app
  namespace: argocd
spec:
  project: default
  source:
    repoURL: 'https://github.com/your-org/your-repo.git'
    targetRevision: HEAD
    path: overlays/production
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Then apply it:

```
kubectl apply -f app.yaml -n argocd
```

Next Steps

Once your repo is connected, ArgoCD can now **monitor**, **sync**, and **deploy** Kubernetes manifests automatically or on demand.

4. Defining Applications in ArgoCD

In ArgoCD, an **Application** is a custom Kubernetes resource (kind: Application) that defines how ArgoCD should deploy and manage a specific app from your Git repository.

Key Components of an ArgoCD Application

An Application CRD includes:

- **Source**: Where to pull manifests from (Git repo, Helm chart, etc.)
- **Destination**: Which cluster and namespace to deploy to
- **Sync Policy**: How to handle updates (manual or automated)

Example: Basic Application YAML

Here's a sample Application manifest using Kustomize:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/argoproj/argocd-example-apps.git
    targetRevision: HEAD
    path: kustomize-guestbook
  destination:
    server: https://kubernetes.default.svc
    namespace: default
```

```
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

Apply the Application

Save the file as guestbook-app.yaml, then apply it:

```
kubectl apply -f guestbook-app.yaml -n argocd
```

You should now see the application listed in the ArgoCD UI or via CLI:

```
argocd app list
```

Creating Applications via CLI

You can also create applications without writing YAML:

```
argocd app create guestbook \
  --repo https://github.com/argoproj/argocd-example-apps.git \
  --path kustomize-guestbook \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default \
  --sync-policy automated
```

Application Structure Overview

Field	Purpose
metadata.name	Unique name of the app
source.repoURL	Git or Helm repo URL
source.path	Path inside repo to the app

Field	Purpose
destination.server	Target Kubernetes cluster
destination.namespace	Namespace to deploy into
syncPolicy.automated	Enable auto sync/prune/self-heal

Application Types Supported

ArgoCD supports multiple application formats:

- **Kubernetes manifests (raw YAML)**
- **Kustomize**
- **Helm (v2 and v3)**
- **Jsonnet**
- **Custom plugins**

Example for **Helm**:

```
source:
  repoURL: https://charts.bitnami.com/bitnami
  chart: nginx
  targetRevision: 13.1.4
  helm:
    values: |
      service:
        type: LoadBalancer
```

Advanced: App of Apps Pattern

Use one root application to manage multiple child applications. Example structure:

```
apps/
```

```
├── dev-app.yaml
├── prod-app.yaml
└── staging-app.yaml
```

Then deploy an App of Apps:

```
spec:
  source:
    path: apps
    ...
```

5. Application Syncing and Deployment Strategies

ArgoCD's core strength lies in its **GitOps-based syncing**, which continuously ensures that the live Kubernetes state matches the desired state stored in Git. This section covers how syncing works and the deployment strategies ArgoCD offers.

What is Syncing in ArgoCD?

Syncing means ArgoCD compares the desired state (from Git) with the live state (in the cluster). If they differ, ArgoCD can:

- **Show the difference**
- **Allow manual syncing**
- **Automatically apply changes**

Sync Statuses

ArgoCD defines three core sync statuses:

- Synced: Live state matches Git
- OutOfSync: There is a difference between Git and the cluster
- Unknown: ArgoCD can’t determine the state (e.g., missing permissions)

Syncing Methods

1. Manual Sync

Users can manually trigger a sync from the UI or CLI:

argocd app sync <app-name>

Use this method when you want full control over when changes are applied.

2. Automated Sync

Enable automated sync using the syncPolicy.automated field:

```
syncPolicy:
  automated:
    prune: true
    selfHeal: true
```

- prune: Deletes Kubernetes resources no longer defined in Git.
- selfHeal: Reverts any manual changes made in the cluster.

🔁 With this mode, Git becomes the single **source of truth**, and ArgoCD enforces it automatically.

🛤️ Sync Waves (Optional)

If you have interdependent resources, ArgoCD lets you control the sync order using **hooks** or **waves** (Helm/Kustomize) to orchestrate safe deployments.

For example, create a job to run *before* sync with this annotation:

```
metadata:
  annotations:
    argocd.argoproj.io/hook: PreSync
```

Other hook types:

- PreSync
- Sync
- PostSync
- SyncFail

🚀 Deployment Strategies

ArgoCD by itself doesn't provide canary or blue-green rollouts — but it integrates with **Argo Rollouts** for advanced strategies like:

- **Canary Deployments**
- **Blue-Green Deployments**

You define a rollout object and Argo Rollouts takes care of traffic control using Ingress/Nginx/Istio/etc.

Example Rollout snippet:

```
apiVersion: argoproj.io/v1alpha1 kind:
    Rollout
metadata:
     name: my-rollout
   spec:
 strategy:
  canary:
   steps:
        - setWeight: 30
        - pause: {duration: 10s}
```

You can then manage this rollout via ArgoCD just like any other resource.

🌓 Safeguards & Controls

- **Sync Windows**: Define time-based policies for when syncs are allowed.
- **Resource Hooks**: Execute pre/post tasks like DB migrations safely.
- **Retry Logic**: ArgoCD retries failed syncs when enabled.

☑ Best Practices

- Use automated sync with selfHeal for production GitOps workflows.
- Combine with Argo Rollouts for safe, progressive deployments.
- Store all application manifests (and ArgoCD app definitions) in version- controlled Git repositories.
- Monitor sync status and application health in the ArgoCD dashboard.

6. Managing Access and RBAC in ArgoCD

Security is crucial in any GitOps setup. ArgoCD provides a **role-based access control (RBAC)** system to manage **who can do what**, and supports integration with **SSO providers** like GitHub, LDAP, Google, etc.

🔑 Access Control Overview

ArgoCD's access control governs:

- Access to the ArgoCD UI and CLI
- Actions like creating apps, syncing, deleting, or modifying projects
- Resource-level control per namespace or project

🔐 Default Admin User

Upon installation, ArgoCD enables a default admin user with full access. You can:

- Disable the admin account after setting up SSO
- Change the admin password

using: argocd account update-password

🗂 RBAC Configuration File

ArgoCD RBAC policies are defined in the **ConfigMap** named argocd-rbac-cm. To edit:

```
kubectl edit configmap argocd-rbac-cm -n argocd
```

❎ Example RBAC Rules

```
apiVersion: v1
  kind: ConfigMap
  metadata:
 name: argocd-rbac-cm
```

```
namespace: argocd

data:
  policy.default: role:readonly
  policy.csv: |
    p, role:readonly, applications, get, *, allow
    p, role:admin, *, *, *, allow
    g, alice, role:admin
    g, bob,
    role:readonly
```

Explanation:

- p: policy rule
- g: group mapping (user-to-role binding)
- role:admin: full control
- role:readonly: view-only

👥 Authentication Modes

ArgoCD supports the following auth modes:

1. **Local Users (default)** – defined in argocd-cm ConfigMap
2. **SSO Integration (recommended for production)** – OIDC-based with:
    - GitHub
    - GitLab
    - Google
    - Azure AD
    - LDAP

🔧 Example: SSO with GitHub

Edit argocd-cm:

```
data:
  oidc.config: |
    name: GitHub
    issuer: https://github.com/login/oauth
    clientID: <your-client-id>
    clientSecret: $oidc.github.clientSecret
    requestedScopes: ["openid", "profile", "email", "groups"]
```

Then add your secret as a Kubernetes secret:

```
bash
CopyEdit
kubectl -n argocd create secret generic argocd-secret \
  --from-literal=oidc.github.clientSecret=<your-secret> \
  --dry-run=client -o yaml | kubectl apply -f -
```

🔐 Best Practices for Access Management

- **Use SSO with group-based access** for easier scaling and auditability
- **Limit use of the admin account** in production
- Define roles **per project**, not globally, to minimize blast radius
- Enable **audit logging** using ArgoCD's logging features or external tools (e.g., Loki/Grafana)

7. Monitoring and Observability

Monitoring and observability in ArgoCD help you **track the health, sync status, and performance** of your applications and infrastructure. This visibility is essential for ensuring smooth operations in a GitOps-driven Kubernetes environment.

📊 ArgoCD UI Dashboard

ArgoCD includes a powerful **web UI** at https://<your-server-ip>:<port> or via port-forward:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

- View all applications, their sync status, health, and history.
- Drill down into live diffs between Git and cluster state.
- Manually sync, delete, or rollback apps.

🩺 Health Statuses

ArgoCD assigns a **health status** to each resource:

- Healthy: Resource is running and stable.
- Progressing: Deployment or pod is not fully available yet.
- Degraded: Resource failed (e.g., CrashLoopBackOff).
- Missing: Resource is no longer found in the cluster.

Each resource kind has its own health checks, and you can **extend/customize** them.

🔁 Sync Statuses

- Synced: Live state matches Git.
- OutOfSync: Live state differs from Git.
- Unknown: ArgoCD cannot determine the status (often due to permission issues).

Prometheus Metrics Exporter

ArgoCD exposes **Prometheus-compatible metrics** via /metrics endpoint:

kubectl port-forward svc/argocd-metrics -n argocd 8082:8082

Common metrics include:

- argocd_app_info
- argocd_app_health_status
- argocd_app_sync_status

These are very useful for alerting and dashboards.

Grafana Dashboards

ArgoCD has ready-made dashboards you can import into Grafana.

- Install Grafana and Prometheus into your cluster.
- Scrape ArgoCD metrics.
- Import ArgoCD dashboard from

Grafana You'll see:

- Application health over time
- Sync trends
- Error and failure patterns

Alerts and Notifications

Use the **ArgoCD Notifications Controller** (optional component) to send alerts via:

- Slack
- Email
- Microsoft Teams
- Webhooks
- PagerDuty

Example notification configuration:

```
trigger.on-deployed:
    - when: app.status.health.status ==
      'Healthy' send: [slack-notification]
```

📝 Audit Logs

ArgoCD logs:

- API access
- Sync actions
- User activity (via audit logs)

You can collect these logs using tools like:

- **Fluentd / Fluent Bit**
- **ELK Stack**
- **Loki + Grafana**

☑ Best Practices

- Always monitor both health and sync statuses.
- Integrate with Prometheus/Grafana for long-term insights.
- Set up notifications for drift, sync failures, and degraded apps.
- Use health annotations for custom resource types.

8. Best Practices and Troubleshooting Tips

This section summarizes tried-and-tested **best practices** for operating ArgoCD reliably and efficiently in production, along with **common issues and how to fix them**.

☑ Best Practices for Using ArgoCD

1. Git as the Single Source of Truth

- Store *all Kubernetes manifests*, *application definitions*, and *environment configurations* in Git.
- Any manual change in the cluster will be marked as **OutOfSync** — promoting GitOps discipline.

2. Use Automated Sync + Self-Heal in Production

Enable auto-sync and self-heal in your app spec:

```
syncPolicy:
  automated:
    prune: true
      selfHeal: true
```

- prune: Deletes resources removed from Git
- selfHeal: Restores deleted or modified resources

This ensures the cluster *always aligns with Git*, minimizing drift.

3. Environment Segregation via Projects

Use **ArgoCD Projects** (kind: AppProject) to isolate environments like dev, staging, and prod.

Benefits:

- Set **resource quotas**
- Restrict **destination clusters**

Define **allowed Git repos**

- Attach **role-based access (RBAC)** per

project Example:

```
spec:
 destinations:
  - namespace: dev
    server: https://kubernetes.default.svc
```

4. Use the "App of Apps" Pattern for Large Deployments

Structure your Git repos to allow nesting multiple ArgoCD apps under a parent app:

```
apps/
├── dev-app.yaml
├── prod-app.yaml
└── staging-app.yaml
```

Deploy all apps by syncing just the root app.

5. Secret Management with External Tools

ArgoCD does **not manage secrets natively**. Use:

- **Sealed Secrets (Bitnami)**
- **SOPS (Mozilla)**
- **External Secrets Operator**
- **HashiCorp Vault**

Keep sensitive data **out of Git**, and reference it via Kubernetes Secrets created securely.

6. Git Branch Per Environment

Organize apps by using Git branches like:

- main → production
- develop → dev
- release/x.x → staging

Or use directory-based separation if using mono-repos:

```
apps/
├── dev/
├── prod/
```

7. Backup ArgoCD Configurations

Important ArgoCD configurations to back up:

- argocd-cm (Core config)
- argocd-rbac-cm (RBAC)
- argocd-secret (OAuth tokens, webhooks)
- Application CRDs (Application,

AppProject) Regularly export:

```
kubectl get application -n argocd -o yaml > apps-backup.yaml
```

8. Integrate Observability and Alerts

- Use **Prometheus & Grafana** for metrics and dashboards.
- Install **ArgoCD Notifications Controller** for alerts.
- Monitor metrics like:
  - argocd_app_health_status
  - argocd_app_sync_status

Troubleshooting Common Issues

❌ App Shows "OutOfSync" Constantly

Cause: Drift between Git and cluster or unsupported field differences (e.g., timestamps).

Fix:

- Run argocd app diff <app-name> to see the exact changes
- Check if kubectl apply adds auto-generated fields

❌ "Unknown" Health Status

Cause: ArgoCD can't determine resource health.

Fix:

- Check ArgoCD controller logs
- Create custom health checks via Lua scripts or annotations

🔒 SSO Login Fails

Fixes:

- Validate your oidc.config syntax
- Check clientID/clientSecret and correct redirect URIs
- Check for argocd-server service logs

🔃 App Sync Stuck or Fails

Possible Causes:

- Missing RBAC for ArgoCD service account
- Incomplete Git manifests
- Hook resource fails

Fix:

- Check pod logs via kubectl
- Run argocd app get <app-name> for sync status and history

Use argocd app history <app-name> to roll back

🛑 Resource Deletion Doesn't Work (Prune Fails)

Fix:

- Ensure automated.prune: true is set
- Check if finalizers block deletion
- Inspect ArgoCD's permissions: it must have delete rights

🧠 Final Thoughts

- Think of ArgoCD as the **GitOps enforcer**—its job is to maintain consistency between Git and the cluster.
- Combine ArgoCD with CI tools (like GitHub Actions or Jenkins) for full CI/CD.
- Regularly audit access, application state, and Git history.
- Treat your GitOps repo like code: pull requests, code reviews, and version control should apply.

☑ Conclusion

ArgoCD has revolutionized Kubernetes deployments by bringing the **GitOps** model to life—turning Git into the *single source of truth* for declarative infrastructure and application management. By automating continuous delivery

directly from Git repositories, ArgoCD ensures consistent, auditable, and reliable Kubernetes operations.

Throughout this guide, we explored:

- Installing and setting up ArgoCD
- Creating and managing applications
- Enforcing sync policies and health checks
- Securing access through RBAC and SSO
- Observing and troubleshooting applications effectively

With its intuitive UI, CLI, RBAC, and integration with tools like Prometheus, Grafana, and Vault, ArgoCD offers a comprehensive GitOps solution that's **scalable**, **secure**, and **developer-friendly**.

By following best practices such as using projects, managing secrets securely, adopting the App of Apps pattern, and enabling automated syncs, teams can streamline their CI/CD pipelines and minimize manual overhead.

Ultimately, adopting ArgoCD is not just about deploying apps—it's about enabling a **culture of collaboration, traceability, and automation** that aligns development and operations around a shared, Git-centric workflow.