Antony Kervazo-Canut

ArgoCD for Teenagers

SUITABLE FOR ADULTS

Introduction

ArgoCD is a continuous deployment platform specifically designed for Kubernetes, adopting the GitOps approach. GitOps is a method of managing infrastructures and applications that uses Git as the single source of truth. With GitOps, changes in infrastructure or applications are made through modifications in a Git repository, thus ensuring traceability, revision, and reproducibility of deployments.

The benefits of GitOps with ArgoCD include: Automation and consistency:

- Deployments are automatically carried out when the Git repository is updated, ensuring consistency between the source code and the deployed state.
- Traceability and audit: Each modification is recorded in Git, offering complete traceability.
- Ease of recovery after an incident: In case of a problem, it is easy to revert to a previous state using Git's reversion functionalities.
- Enhanced security: Approvals and code reviews in Git contribute to better deployment security.

The architecture of ArgoCD consists of several key components: API Server: The API server is the central point of communication. Repository Server: Manages communication with Git repositories. Controller: Monitors the state of applications and manages synchronization.

ArgoCD Installation

You can install ArgoCD on your Kubernetes cluster by applying the YAML manifest provided by ArgoCD.

ArgoCD also works via CLI, but to access the ArgoCD web interface, expose the argocd-server service outside the cluster.

Finally, you will need to install ArgoCD via CLI to manage it in this way.

```
# Create the namespace
kubectl create namespace argocd

# Install ArgoCD on the cluster
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml

# Create a service to access the web interface
kubectl port-forward svc/argocd-server -n argocd 8080:443

# Linux: Download the ArgoCD CLI (replace VERSION with the
desired version)
curl -sSL -o argocd https://github.com/argoproj/argo-
cd/releases/download/vVERSION/argocd-linux-amd64
chmod +x argocd
sudo mv argocd /usr/local/bin/

# Windows: Install the ArgoCD CLI with Chocolatey
choco install argocd

# Install the ArgoCD CLI with Homebrew
brew install argocd
```

ArgoCD Configuration

Once the CLI is installed, you can connect to your ArgoCD server. The first step involves retrieving the IP address or the name of the argocd-server service.

```
# Get the IP address of ArgoCD (if using a LoadBalancer)
kubectl get svc -n argocd argocd-server -o
jsonpath='{.status.loadBalancer.ingress[0].ip}'

# Or use port-forward to access locally
kubectl port-forward svc/argocd-server -n argocd 8080:443

# Get the initial admin password
kubectl get secret argocd-initial-admin-secret -n argocd -o
jsonpath="{.data.password}" | base64 -d; echo

# Log in to ArgoCD (replace ARGOCD_SERVER with the IP address
or domain name)
argocd login ARGOCD_SERVER --username admin --password
<initial_admin_password>
```

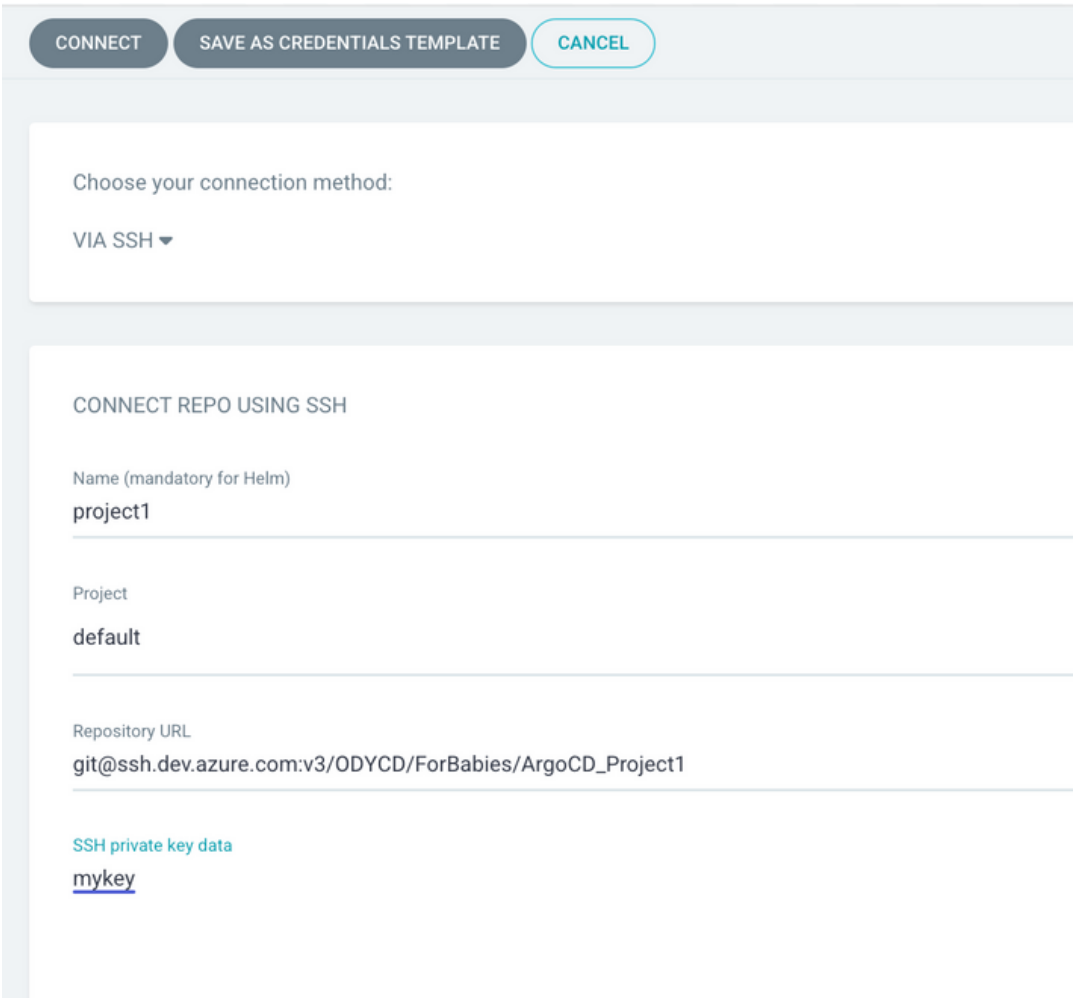
It then becomes possible to connect to the web interface on port 8080. The password for the "admin" account is the one returned by the CLI.

Configuration of a Repository (SSH)

To configure a Git repository via SSH in ArgoCD, you must follow several steps to ensure secure communication between ArgoCD and your Git repository. This involves creating an SSH key pair, adding the public key to your Git repository, and configuring ArgoCD to use the private key when accessing the repository.

The process of creating SSH key pairs will not be detailed here.

On one hand, place your public SSH key in a forge of your choice (Azure DevOps, GitHub, GitLab, etc.) and on the other hand, go to ArgoCD/Settings/Repositories and click the "Connect Repo" button.



Creation of an Application (GUI)

To create an application, you have two methods: the GUI or the CLI. But first, it's necessary to define what an application is for ArgoCD.

It's a repository composed of resources for Kubernetes. So, for our example, we will populate the (empty) repository we connected in the previous page with an nginx deployment.

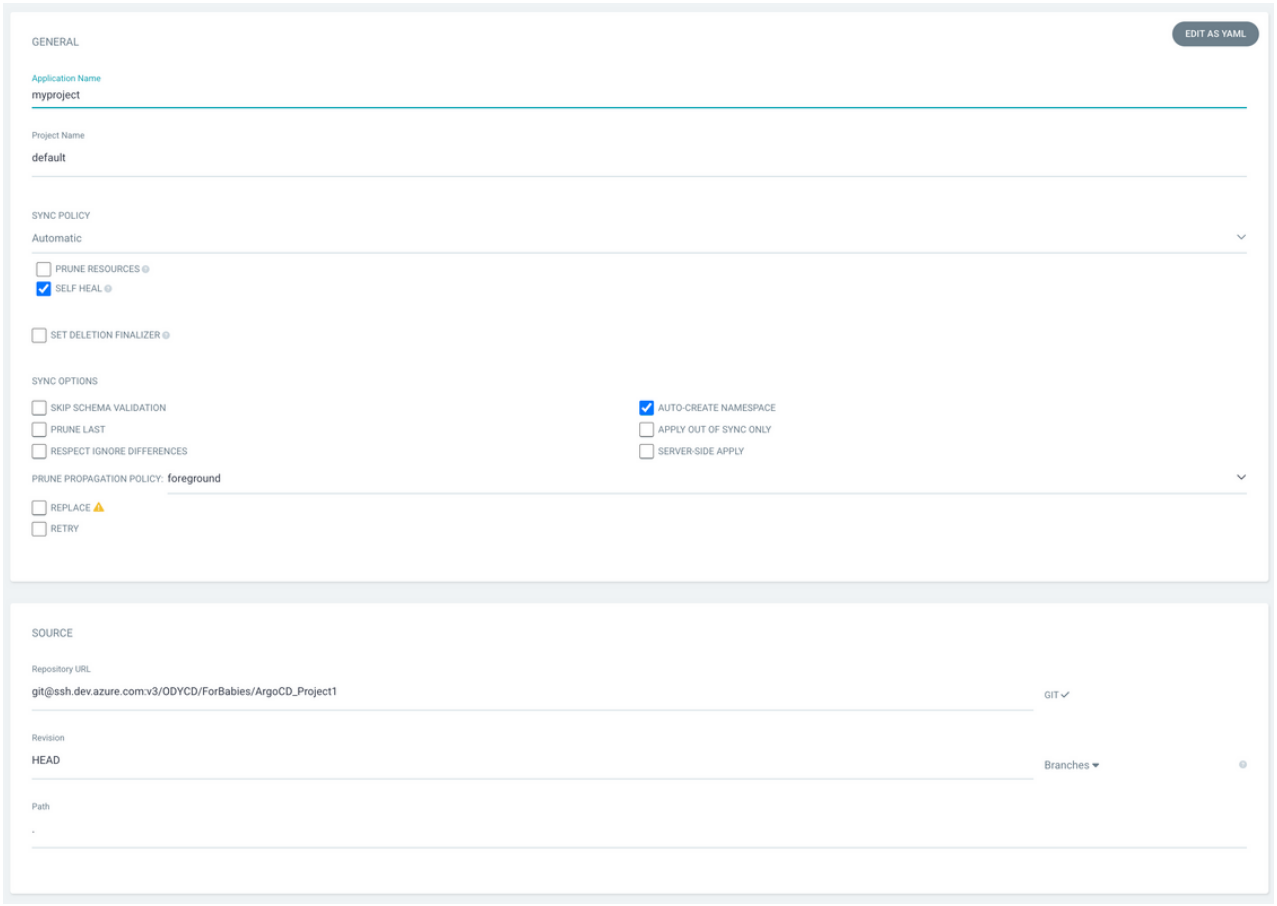
```yaml
# file: nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: nginx-namespace
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Creation of an Application (GUI)

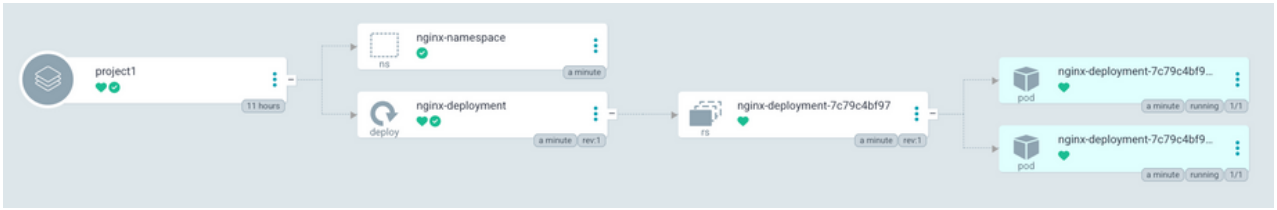
ArgoCD offers yaml files specific to its operation if using text fields is daunting. They can also be easily used for insertion via a CLI, and therefore, potentially integrated into a Delivery pipeline.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myproject
spec:
  destination:
    name: ''
    namespace: nginx-namespace
    server: 'https://kubernetes.default.svc'
  source:
    path: .
    repoURL:
'git@ssh.dev.azure.com:v3/ODYCD/ForBabies/ArgoCD_Project1'
    targetRevision: HEAD
  sources: []
  project: default
  syncPolicy:
    automated:
      prune: false
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

Creation of an Application (GUI)



Once the application is created with our repository as a reference, ArgoCD will deploy our resource and observe it on the Kubernetes cluster.



Creation of an Application (CLI)

In CLI, you have several ways to deploy the application. Using a YAML file, which is more practical to maintain, or directly passing the creation parameters in the command.

```
# Create an application directly using the YAML file.
argocd app create --file myproject-application.yaml

# Or create an application using parameters
argocd app create myproject \
  --dest-namespace nginx-namespace \
  --dest-server https://kubernetes.default.svc \
  --repo
git@ssh.dev.azure.com:v3/ODYCD/ForBabies/ArgoCD_Project1 \
  --path . \
  --project default \
  --sync-policy automated \
  --sync-option CreateNamespace=true \
  --auto-prune false \
  --self-heal true
```

Kustomize

ArgoCD uses Kustomize configuration files (kustomization.yaml) to generate the final Kubernetes resources from base templates and customization overlays. Overlays allow specifying modifications for a specific environment without needing to duplicate the entire set of configuration files.

Git applications then deserve a restructuring and organization of the files.

```
my-application/
├── base/
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays/
    ├── production/
    │   ├── kustomization.yaml
    │   └── patch-production.yaml
    └── development/
        ├── kustomization.yaml
        └── patch-development.yaml
```

Kustomize

When creating the Application object in ArgoCD for your application, specify the path to the overlay folder corresponding to the environment you want to deploy.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: mon-application-production
spec:
  project: default
  source:
    repoURL:
'git@ssh.dev.azure.com:v3/ODYCD/ForBabies/ArgoCD_Project1'
    targetRevision: HEAD
    path: mon-application/overlays/production
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: production
```

Creation of an Application (Helm)

It's possible to deploy applications using Helm Charts with ArgoCD, combining Helm's package management for Kubernetes with GitOps automation for simplified application updates and configuration.

This can be more convenient for deploying an application across multiple teams or environments by just changing the configuration.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: sonarqube
spec:
  destination:
    name: ''
    namespace: default
    server: 'https://kubernetes.default.svc'
  source:
    path: ''
    repoURL: 'https://SonarSource.github.io/helm-chart-
sonarqube'
    targetRevision: 10.4.0+2288
    chart: sonarqube
  sources: []
  project: default
  syncPolicy:
    automated:
      prune: false
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

Helm and Multiple Sources

With ArgoCD, it is entirely possible to only store the configuration on Git and to deploy only Helm releases.
This allows for the evolution of the configuration on a single version.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: sonarqube
spec:
  project: default
  source:
    repoURL: 'https://SonarSource.github.io/helm-chart-
sonarqube'
    targetRevision: 10.4.0+2288
    chart: sonarqube
    helm:
      valueFiles:
        - $values/values-sonar-dev.yaml
  sources:
    - repoURL:
'git@ssh.dev.azure.com:v3/ODYCD/ForBabies/ArgoCD_ProjectSonar
'
      targetRevision: main
      ref: values
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: default
  syncPolicy:
    automated:
      prune: false
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

Synchronization

Automatic vs Manual: You can configure an application to synchronize automatically whenever the Git repository changes (automatic synchronization), or you can choose to trigger synchronization manually (manual synchronization).

- Automatic: With the automated option, synchronization happens automatically. You can also specify whether resources not present in the Git repository should be deleted from the cluster (prune: true) and whether ArgoCD should attempt to automatically correct deviations without manual intervention (selfHeal: true).

- Manual: Manual synchronization requires the user to explicitly trigger a synchronization via ArgoCD's user interface or CLI.

- Prune: When enabled (prune: true in an automatic synchronization policy), this option tells ArgoCD to delete Kubernetes resources that are no longer present in the Git repository but still exist in the cluster.

- Self-Heal: If enabled (selfHeal: true), ArgoCD will monitor changes to Kubernetes resources in the cluster that diverge from the Git configuration and automatically reapply them. This ensures that the state of the cluster remains in sync with the Git repository.

SyncOptions

The syncOptions allow for further customization of synchronization:

- CreateNamespace: If set (CreateNamespace=true), ArgoCD will automatically create the specified namespace for the application if it does not already exist in the cluster. This is useful for deployments that require the creation of new namespaces on the fly.

- Validate: Controls the validation of resources before their application. You can disable validation with Validate=false if necessary.

- PrunePropagationPolicy: Defines the propagation policy for delete operations, for example. PrunePropagationPolicy=Foreground will ensure that all dependent resources are deleted before the resource itself.

- PruneLast: Allows specifying resources that should be deleted last during the prune operation, useful for managing dependencies between resources.

- Retry: Configures retry attempts in case of synchronization failure, for example:
  Retry=limit:5,backoff:exponential,maxDuration:15m.

SyncOptions

The application YAML with SyncOptions enabled would look something like this:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myproject
spec:
  project: default
  source:
    repoURL:
'git@ssh.dev.azure.com:v3/ODYCD/ForBabies/ArgoCD_Project1'
    path: .
    targetRevision: HEAD
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: nginx-namespace
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
      - Validate=true
      - PrunePropagationPolicy=Foreground
      - PruneLast=true
      - ApplyOutOfSyncOnly=true
      - SkipDryRunOnMissingResource=true
```

Resource Hooks

Resource hooks in ArgoCD are powerful tools that allow you to orchestrate the deployment of your applications by executing specific tasks at different points in the synchronization process. These hooks use standard Kubernetes resources, such as Jobs or Pods, to run scripts or commands before, during, or after the application's synchronization. They are particularly useful for managing database migrations, cleanups, notifications, and other management or initialization tasks necessary for a successful deployment.

ArgoCD supports several types of hooks, each executed at a specific moment in the deployment lifecycle:

- PreSync: Executed before any changes are applied to the cluster.
- Sync: Executed after the PreSync hooks have completed but before the desired state synchronization is performed.
- PostSync: Executed after all resources have been synchronized with the desired state in the Git repository.
- SyncFail: Executed if the synchronization process fails at any point.

Resource Hooks

To use a resource hook, you need to define a Kubernetes resource in your Git repository that includes annotations specifying the type of hook and when it should execute. Here's an example of a Kubernetes Job configured as a PreSync hook:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: example-pre-resources-hook
  annotations:
    argocd.argoproj.io/hook: PreSync
spec:
  template:
    spec:
      containers:
      - name: pre-sync
        image: alpine
        command: ["/bin/sh", "-c"]
        args: ["echo Pre-sync hook; sleep 60"]
      restartPolicy: Never
```

The concept of weight (hook-weight) in Resource Hooks can be used to control the execution order of hooks when there are multiple to execute in the same phase of the deployment lifecycle (for example, multiple PreSync hooks). The weight is an integer, and hooks are executed from the lowest to the highest weight. Hooks with the same weight are executed in an indeterminate order relative to each other.

Resource Hooks

Here's an example of two resource hooks where the execution order is managed by their weight:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pre-sync-database-migration
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-weight: "10"
spec:
  template:
    spec:
      containers:
      - name: migration
        image: myapp/migration:latest
        command: ["/bin/sh", "-c", "perform-migration"]
      restartPolicy: Never
---
apiVersion: batch/v1
kind: Job
metadata:
  name: pre-sync-seed-database
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-weight: "20"
spec:
  template:
    spec:
      containers:
      - name: seed
        image: myapp/seed:latest
        command: ["/bin/sh", "-c", "seed-database"]
      restartPolicy: Never
```

Wave Synchronization

Synchronization "waves" allow you to group resources into sequential deployment phases. You can specify the deployment order by assigning "wave" values to resources using the annotation argocd.argoproj.io/sync-wave. Resources with a lower "wave" value are deployed before those with a higher value.

Imagine you are deploying an application composed of a database, a backend, and a frontend. You want the database to be deployed first, followed by the backend, and then the frontend.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: database-deployment
  annotations:
    argocd.argoproj.io/sync-wave: "0"
spec:
  # Deployment specification
```

Wave Synchronization

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  annotations:
    argocd.argoproj.io/sync-wave: "1"
spec:
  # Deployment specification
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  annotations:
    argocd.argoproj.io/sync-wave: "2"
spec:
  # Deployment specification
```

You then have precise control over the deployment order, which is essential for interdependent applications.
This reduces the risk of issues caused by unresolved dependencies during deployment.

Multi-cluster Kubernetes

Multi-cluster support in ArgoCD allows managing deployments across multiple Kubernetes clusters from a single ArgoCD instance, offering a powerful solution for organizations operating distributed Kubernetes environments. This feature facilitates centralized management of applications across different clusters, enabling consistent operations, increased visibility, and improved governance.

It is only done via CLI.

```
# Retrieve the list of different known clusters
kubectl config get-contexts

# Add a new cluster managed by ArgoCD
argocd cluster add CONTEXT_NAME --name name_for_argocd
```

Clearly name your clusters and applications to reflect their environment and function (for example, prod, dev, us-east, etc.), facilitating large-scale management.

Community

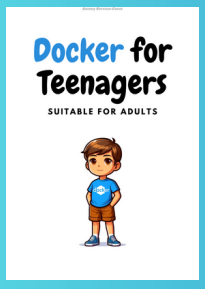
The ArgoCD community has developed several tools and extensions to enrich and extend the core functionalities of ArgoCD, addressing various needs around the deployment, management, and automation of Kubernetes applications. Among them are a few tools:

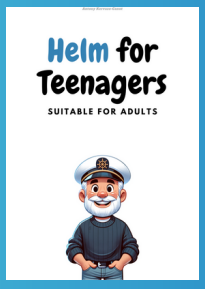
- ArgoCD Notifications
  - Sends configurable notifications for ArgoCD events through various channels like Slack, Email, and others.
- ArgoCD Image Updater
  - Automates the updating of container images in deployments managed by ArgoCD, based on defined rules.
- ArgoCD Autopilot
  - Simplifies getting started with GitOps and ArgoCD by automating the initial setup and management of GitOps projects.
- ArgoCD Vault Plugin
  - Integrates ArgoCD with HashiCorp Vault for secure management of secrets in application configurations.
- ArgoCD Exporter
  - Provides Prometheus metrics for ArgoCD, allowing monitoring of the health and performance of the ArgoCD instance.
- ArgoCD Rollouts
  - Offers advanced deployment capabilities, such as Blue/Green and Canary, for Kubernetes, which can be integrated with ArgoCD for more refined deployment strategies.
- Argo Workflows
  - A workflow engine for Kubernetes that enables executing parallel and sequential tasks within a cluster, often used in tandem with ArgoCD for CI/CD pipelines.

There are others, and the complete and detailed list can be found here:
https://github.com/argoproj

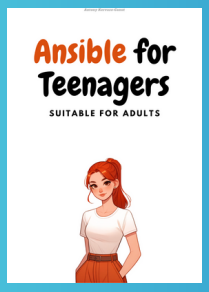
In the same collection

Container Orchestration and Management





Infrastructure as Code



Security & Secrets Management

ANTONYCANUT

ANTONY KERVAZO-CANUT