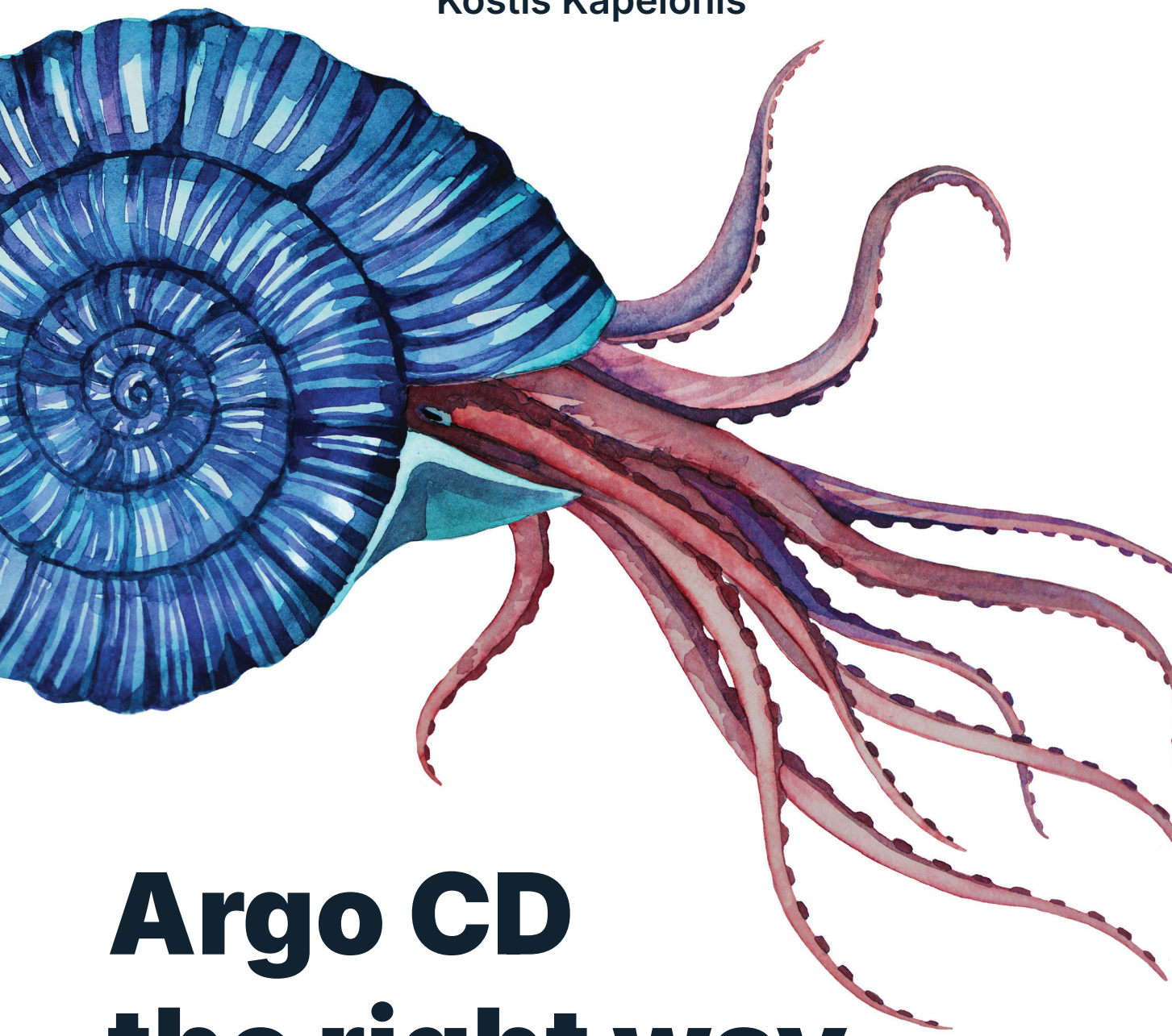
Kostis Kapelonis



Argo CD the right way

How to apply best practices and avoid anti-patterns

First edition

Octopus Deploy

Argo CD the right way

How to apply best practices and avoid anti-patterns

Kostis Kapelonis

Argo CD the right way

How to apply best practices and avoid anti-patterns

Kostis Kapelonis

Paperback ISBN-13 979-250499149

Ed.1.2026.1

Contents

Introduction

The container wars are finished. Kubernetes has won. And it has won on two fronts. Companies that migrate their workloads to the cloud choose Kubernetes for the broad industry support and tool availability. But even organizations that prefer to keep their own data centers are still choosing Kubernetes to take advantage of its autoscaling capabilities.

To get any value from Kubernetes, your teams need to deploy their applications to it, and the most popular way to do so today is Argo CD. Argo CD is a Kubernetes deployment solution that follows the **GitOps Principles**[1] and can continuously synchronize manifest updates to a large number of clusters.

At Octopus Deploy, we have helped many organizations in their GitOps adoption. After a while, we noticed a set of patterns (and anti-patterns) that can help (or hinder) your teams as they onboard to Argo CD.

In this book, we have catalogued 30 good (and bad) practices so that your team can avoid common mistakes and configuration traps, and help you with all architectural decisions for GitOps and Argo CD. The practices we have chosen to discuss are those that affect Enterprise adoption of Argo CD in teams with a large number of applications and an equally large number of Kubernetes clusters.

On several occasions, we have talked to enthusiastic teams that recognize the benefits of GitOps and want to adopt Argo CD as quickly as possible. The initial adoption phase goes smoothly, and more and more teams are onboarded to Argo CD. However, after a certain point, things start slowing down, and developers start complaining about the new process.

The main challenge here is that scaling up Argo CD is entirely different than the initial installation. Onboarding the first team with Argo CD might seem trivial, but onboarding the *n*th team can be very complex.

GitOps principles

The **OpenGitOps project**[1] defines 4 GitOps principles in version 1.0.0.

1. Declarative
A system managed by GitOps must have its desired state expressed declaratively.

2. Versioned and Immutable
Desired state is stored in a way that enforces immutability, versioning, and retains a complete version history.

3. Pulled Automatically
Software agents automatically pull the desired state declarations from the source.

4. Continuously Reconciled
Software agents continuously observe the actual system state and attempt to apply the desired state.

A catalog of anti-patterns

It's difficult for us to determine exactly where your team is on the GitOps adoption journey. You may have picked this book because you are currently evaluating Argo CD and want to avoid all the common pitfalls. Or you may have already deployed your first applications to production and are getting blocked by scaling challenges.

We'll let you make your own determination on your current state and choose which sections most interest you. You don't need to read the whole book from start to finish, but keep in mind that some practices depend on and build on earlier sections.

The order of anti-patterns follows the timeline of an organization that starts with minimal Argo CD knowledge and slowly migrates several applications to the GitOps paradigm. If you've already faced the initial problems mentioned (and solved them), feel free to skip ahead.

If you find the content you are reading too abstract, you might not have uncovered the problem yet. It's okay to skip sections and come back to them later when your team encounters the challenge.

Before adopting Argo CD

This section describes the prerequisites for adopting Argo CD. It can serve as a checklist to help you understand what you need before committing your first manifest to Git. This section will give you a good idea of what you need to have in place before embracing GitOps.

	Description	Area
1	Not understanding the declarative setup of Argo CD	Adopting Gitops
2	Creating Argo CD applications in a dynamic way	Adopting Gitops
3	Using Argo CD parameter overrides	Adopting Gitops
4	Adopting Argo CD without understanding Helm	Prerequisite knowledge
5	Adopting Argo CD without understanding Kustomize	Prerequisite knowledge
6	Assuming that developers need to know about Argo CD	Developer Experience

Foundations and first steps

This section covers practices to follow after initial installation and discusses important choices for Argo CD. How many Git repositories should you have? How are applications organized? What sync policies must you use? All these are essential questions that you need to answer before bringing Argo CD to production deployments.

	Description	Area
7	Grouping applications at the wrong abstraction level	Application Organization
8	Abusing the multi-source feature of Argo CD	Application Organization
9	Not splitting the different Git repositories	Application Organization
10	Turning off auto-sync and self-heal	Adopting Gitops
11	Abusing the target Revision field	Adopting Gitops
12	Misunderstanding immutability for container/git tags and Helm charts	Adopting Gitops
13	Giving too much power (or no power at all) to developers	Developer Experience
14	Referencing dynamic information from Argo CD/ Kubernetes manifests	Application Organization

Configuration management

This section examines the complex scenarios for Argo CD application configuration. Deploying the same application to different clusters might sound easy, but in reality, each cluster (and environment) has a different configuration from the others. We see many misunderstandings in this area, especially with Helm variables.

	Description	Area
15	Writing applications instead of Application Sets	Application Organization
16	Using Helm to package Applications CRDs	Application Organization
17	Hardcoding Helm data inside Argo CD applications	Developer Experience
18	Hardcoding Kustomize data inside Argo CD applications	Developer Experience
19	Attempting to version Applications and Application Sets	Application Organization
20	Not understanding what changes are applied to a cluster	Developer Experience

Cluster management

This section is about Argo CD from an infrastructure aspect. Most companies have several clusters they need to manage uniformly while still reaping all the benefits of GitOps.

	Description	Area
21	Using ad-hoc clusters instead of cluster labels	Cluster management
22	Attempting to use a single application set for everything	Cluster management
23	Using Pre-sync hooks for db migrations	Developer Experience
24	Mixing Infrastructure apps with developer workloads	Developer Experience
25	Misusing Argo CD finalizers	Cluster management
26	Not understanding resource tracking	Cluster management

Extending and integrating Argo CD

The last section discusses using Argo CD with other tools and explains how we see several teams misusing it for tasks outside the GitOps scope. It's great to see teams that love Argo CD, but using a single tool for everything is rarely a good approach.

	Description	Area
27	Creating "active-active" installations of Argo CD	Cluster management
28	Recreating Argo Rollouts with Argo CD and duct tape	Adopting Gitops
29	Recreating Argo Workflows with Argo CD, sync-waves and duct tape	Adopting Gitops
30	Abusing Argo CD as a full SDLC platform	Adopting Gitops

Anti-patterns: Before adopting Argo CD

In this introductory section, we discuss the prerequisites you need to have in place **before** adopting Argo CD. We see several organizations that try to adopt Argo CD without having mastered the basics first (like Helm hierarchies). Other times, we see people who start their Kubernetes journey at the same time as their Argo CD journey. While we understand the mindset behind this approach, attempting to learn everything at once is always a big challenge, especially for developers.

We hear this often: teams say Argo CD actually helped them understand Kubernetes. This is great when it happens, but Argo CD builds on existing Kubernetes concepts, like declarative configuration files and continuous reconciliation loops. Argo CD changes some of the traditional deployment patterns you might have seen with virtual machines or application servers, and one of the biggest mistakes you could make is treating Kubernetes as just another deployment target while assuming Argo CD is similar to Terraform, Ansible, Chef, or Puppet.

1. Not understanding the declarative setup of Argo CD

Following the GitOps principles means that Argo CD can take your Kubernetes manifests (or Helm charts or Kustomize overlays) from Git and sync them on your Kubernetes cluster. Teams well understand this process, and most people are familiar with saving Kubernetes manifests in Git.

It is important to note, however, that even this link between a cluster and a Git repository is itself a Kubernetes resource.

```yaml
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: guestbook
  destination:
    server: https://kubernetes.default.svc
    namespace: guestbook
```

Name of application

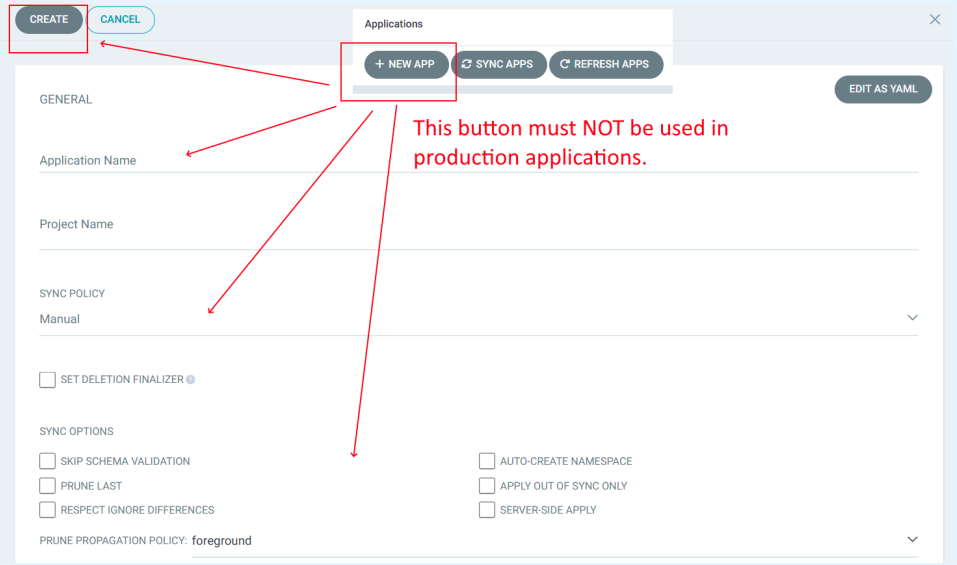
Where to read the Kubernetes manifest from

Which cluster to deploy the application to

Argo CD introduces its own **Custom Resource Definitions (CRDs)**[2] for several of its central concepts like applications and **projects**[3], and also reuses existing Kubernetes CRDs for clusters, secrets, etc.

You should store these files in Git. That is the whole point of following GitOps. It doesn't make sense to use Git only for some files and not store the Applications in the same way.

When teams use the Argo CD UI or CLI to create applications that are not stored in Git, they struggle to understand what they've deployed and where, or how to recreate their Argo CD configuration from scratch. The "create new app" button in the Argo CD UI is only for experiments and quick tests. You should not use it in a production environment at all, as the created application is not saved anywhere.



You should store everything Argo CD needs in Git. When you do, recreating an Argo CD instance should be a simple process with minimal steps:

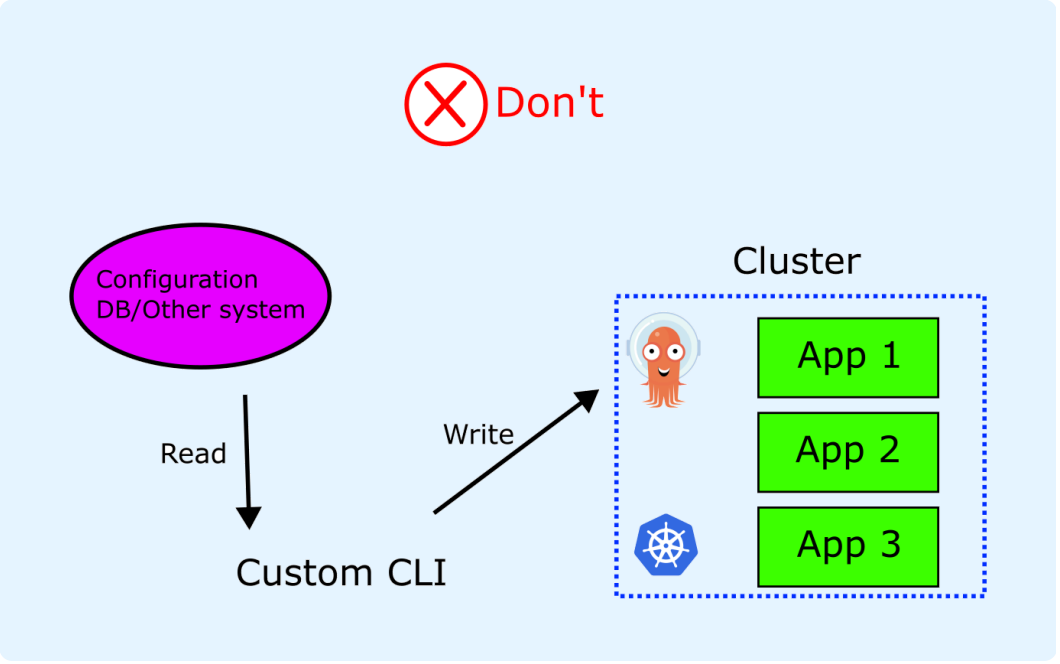
1. Create a new cluster with Terraform, Pulumi, Crossplane, etc.
2. Install Argo CD itself using Terraform, **Autopilot**[4],Codefresh, etc.
3. Point Argo CD to your ApplicationSets or **root app-of-apps file**[5]
4. Done!

Recreating your Argo CD instance is a repeatable process that can be completed in less than 5 minutes (as explained in anti-pattern 27).

If you want a comprehensive guide on how to organize your Kubernetes manifests in Git, read our **Application Set guide**[6].

2. Creating Argo CD applications in a dynamic way

A related anti-pattern occurs when organizations have an existing process for creating applications. When they adopt Argo CD, they use the Argo CD CLI or API to pass the information they already have, rather than following GitOps principles.



Essentially, there is an existing database or application configuration somewhere else, and a custom CLI or other imperative tool does the following:

1. Extracts application configuration from the existing database
2. Creates an Argo CD application or Kubernetes manifest on the fly
3. Applies this file to an Argo CD instance without storing anything in Git.

You have fallen into this trap if the "official" way of creating Argo CD applications in your organization is something like this:

```
my-app-cli new-app-name | argocd app create -f -
```

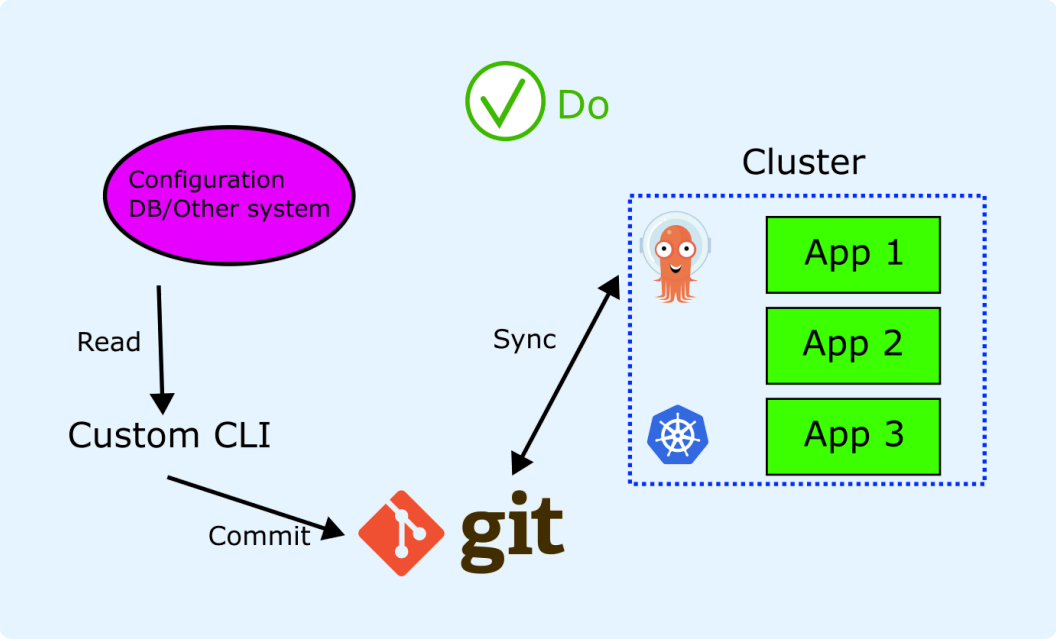
Or if you use **envsubst**[7] like this:

```
envsubst < my-app-template.yaml | kubectl apply -f -n argocd
```

The result is always the same. You have custom Argo CD applications that **are not saved anywhere** in Git. You lose all the main benefits of GitOps:

- You don't have a declarative file for what is deployed right now
- You don't have a history of what was deployed in the past
- Recreating the Argo CD instance is not a single-step process anymore (see anti-pattern 27)

To overcome this anti-pattern, ensure you use Argo CD in a way that aligns with GitOps.



You either need to convert your existing database settings to read and write to Git, or discard them entirely and use Git as the single source of truth for everything. Then, all day 2 operations should also be handled within Git.

The same is true for updating existing applications. If you want to update the configuration of an application, the process should always be the same:

1. You (or an external system) change a file in Git
2. Argo CD notices the change in Git
3. Argo CD syncs the changes to the cluster

If you use the Argo CD API or the Kubernetes API to manually patch resources, you're not following GitOps.

Updating applications in production with any of the following commands goes **against** the GitOps principles.

```
kubectl set image deployment/my-deployment my-container=nginx:1.27.0
kubectl patch deployment <deployment-name> [.....patch here...]
```

Use the Kubernetes API only for experiments and local tests, never for production upgrades (see also anti-pattern 10 - disabling auto-sync).

3. Using Argo CD parameter overrides

Yet another way of updating an Argo CD application in a manner we **don't** recommend is the following:

```
argocd CLI  app set guestbook -p image=example/guestbook:v2.3
```

This script updates the guest book application to version v2.3. Argo CD syncs the version, and everything looks good. But where was this action saved? Nowhere.

This command uses Argo CD's parameter feature to override the Argo CD application with your own custom properties. Even **the official documentation**[8] includes a strong warning against using this feature, as it violates GitOps principles.

Even if you save the parameter information in the Application manifest, you have now destroyed local testing for developers (see anti-patterns 6 and 17).

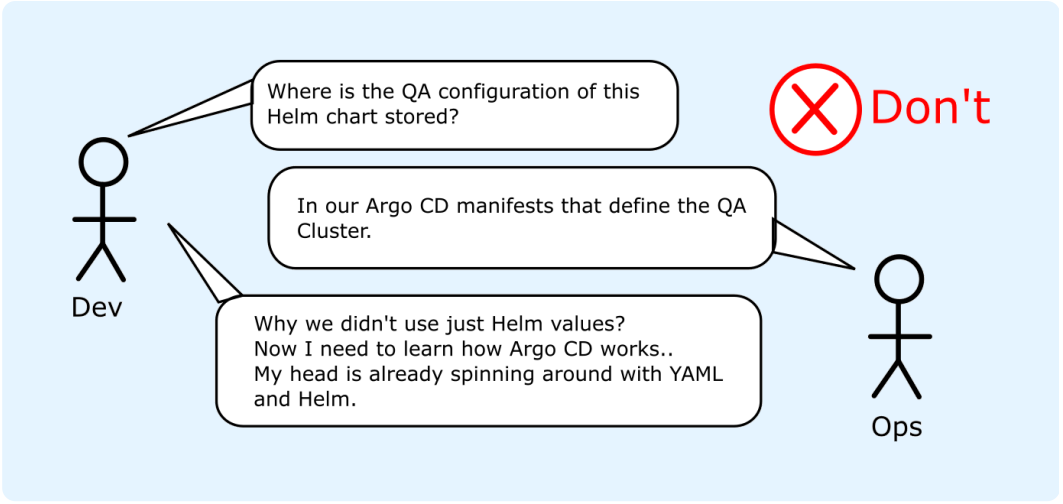
4. Adopting Argo CD without understanding Helm

Helm is the package manager for Kubernetes. In its original form, it offers several essential features in a single platform:

- A package format for Kubernetes manifests
- A repository specification for storing Helm packages in artifact managers
- A templating system
- A comprehensive CLI
- A lifecycle definition (upgrade, install, rollback, test)

Argo CD renders all Helm charts using the **template command**[9] and completely discards most other lifecycle features. Even though in theory this is a good thing, as Argo CD can replace the default Helm lifecycle (e.g., Argo CD comes with its own **rollback command**[10]), Argo CD assumes that you already know how **Helm templates work**[11].

If you are adopting Argo CD and want to use Helm, make sure you understand how Helm works on its own and how to deploy all your applications across your different environments *without* Argo CD. Trying to learn Argo CD and Helm at the same time is a recipe for failure.



At the very least, you should know how to create Helm value hierarchies:

```
common-values.yaml
+-----all-prod-envs.yaml
   +----specific-prod-cluster.yaml
```

And how Helm works when loading the hierarchy with the correct overrides:

```
helm install ./my-chart/ --generate-name -f common.yaml -f more.yaml
```

If you use Helm umbrella charts, understand how to **override child values from the top chart**[12].

In particular, pay attention to the following:

```
restaurant:
  menu: vegetarian
```

This can be a simple value setting for a Helm chart that sets the `restaurant.menu` value to "vegetarian". It can also be an umbrella chart that has a dependency sub-chart called "restaurant", which itself has a property called "menu". You need to understand why those two approaches are different and the advantages and disadvantages of each.

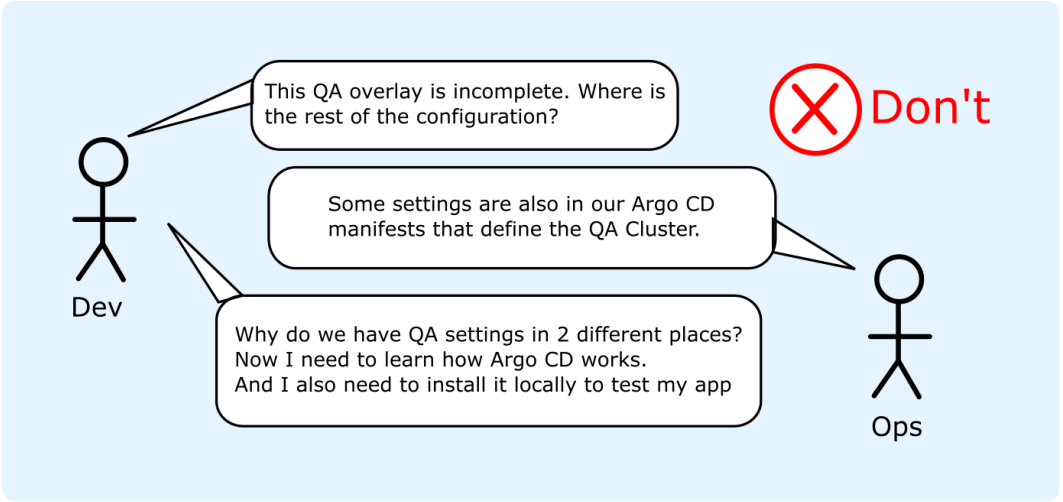
We have written a complete guide on how to use **Argo CD with Helm value hierarchies**[13].

5. Adopting Argo CD without understanding Kustomize

This is similar to the previous anti-pattern, but for Kustomize users. Kustomize is a powerful tool and includes several features such as:

- Overlays for different environments
- Reusable **component**[14] configurations
- Common globals (labels, annotations, namespace prefixes)
- **Configuration generators**[15]
- Transformers/**Replacements**[16]
- Remote resources

Argo CD can reuse all Kustomize features, provided your Kustomize files are correctly structured.

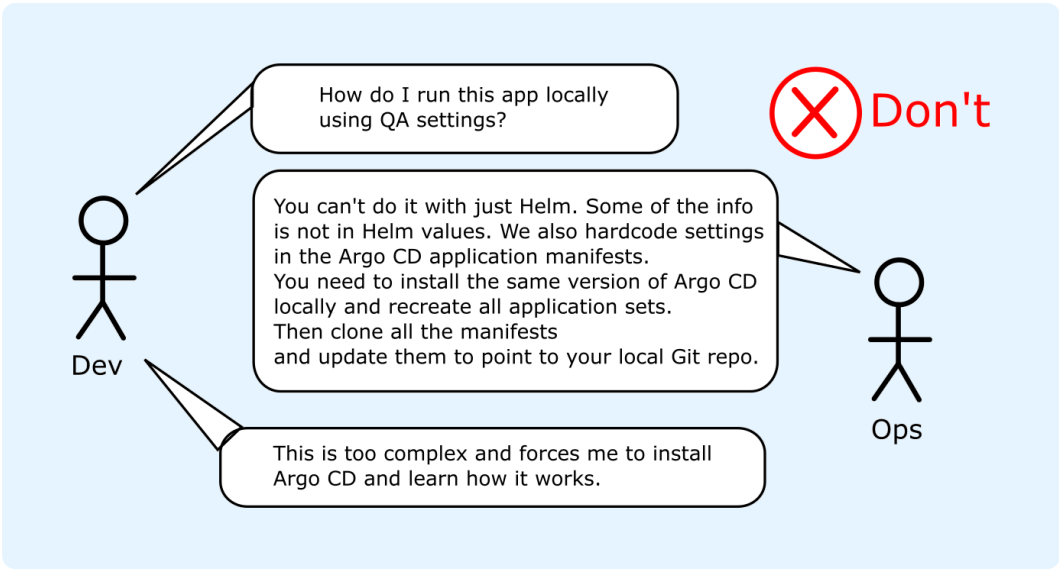


Again, make sure that your Kustomize files work on their own *before* bringing Argo CD into the picture. Well-structured Kustomize applications are self-contained. Any developer should be able to use kustomize (or kubectl) to extract the configuration for an existing environment without Argo CD.

You can find a **full example for Kustomize configurations**[17] in our **Argo CD promotion guide**[18].

6. Assuming that developers need to know about Argo CD

This is the corollary to the previous two anti-patterns. Several teams mix Argo CD configuration data with Kubernetes configuration, making developers' lives extremely difficult.



This is a big problem for developers, as one of their most important tasks is to run an application locally during development and to isolate difficult bugs. Creating Argo CD configurations and coupling them with Kubernetes manifests prevents them from understanding how an application runs independently.

We will see more specific anti-patterns later that also contribute to this problem, but it is best to know about this trap in advance when you start working with Argo CD.

Remember, developers don't care about Kubernetes manifests. They only care about source code features. It is one thing to ask them to learn the basics (i.e., Helm values), and a completely different thing to require knowledge of Argo CD to recreate the configuration of an existing environment.

When you design your Argo CD repository, you should always consider a developer persona who is an expert on Kubernetes, but *knows nothing about Argo CD*. Can they recreate any application configuration on their machine without using Argo CD? If the answer is no, you are using this anti-pattern.

Find where you have hardcoded Argo CD configurations with Kubernetes configurations and remove the tight coupling. For specific advice, see anti-patterns 17 for Helm and 18 for Kustomize.

We have seen how to split Kubernetes configuration from Argo CD manifests in our Application Set guide.

Anti-patterns: Foundations and first steps

With the initial installation complete, your team is ready to start deploying applications with Argo CD. The power of GitOps also comes with great responsibility. Not only do you need to save all your manifests in Git, but you also need to decide on the appropriate structure.

Git is very flexible. You can use different folders, branches, or even repositories to model your Kubernetes environments. Several teams fall into the trap of looking at what worked for application source code (e.g., Git Flow) and try to map the same principles to Argo CD manifests with questionable results. Don't fall into this trap.

First, understand that Argo CD manifests have nothing to do with source code. Not only should they be separate from source code, but they should also have their own lifecycle. If your developers use branches to model different environments (like QA, staging, and production), that doesn't mean Argo CD repositories must use the same branching strategy.

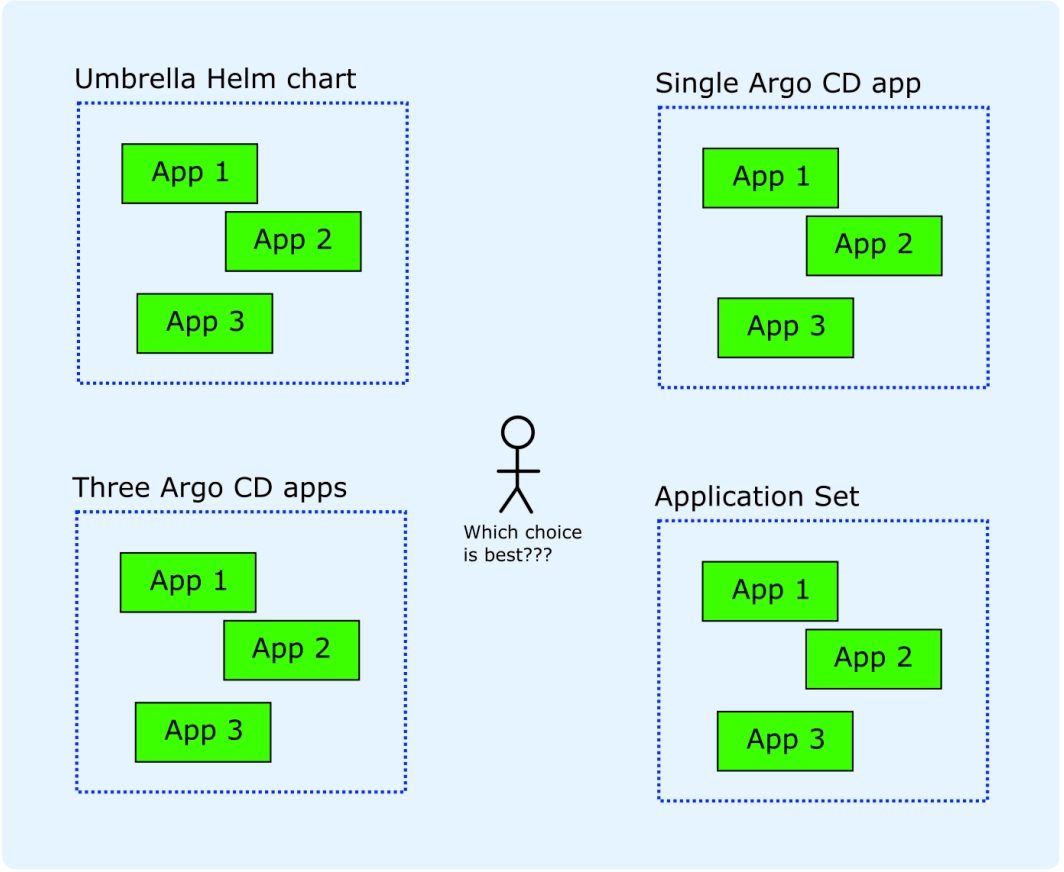
We recommend using folders for your GitOps environments, and this is how popular Kubernetes tools work (overlays in Kustomize and value files in Helm).

This is also the time to decide how much visibility developers will have on Argo CD. Will it be just a read-only dashboard? Will they have access to deploy and synchronize their applications? Can they delete their non-production applications? You need to have a plan in place and understand how Developer Experience can change when adopting Argo CD.

7. Grouping applications at the wrong abstraction level

As we explained in the first anti-pattern (store everything in Git), an Argo CD application is just a link between a Git repository and a Kubernetes cluster. It is not a deployment artifact or a packaging format.

We have seen teams misusing an Argo CD application as a generic grouping mechanism, using it for microservices or even completely unrelated applications.



You need to spend some time understanding what your applications do and how tightly coupled they are. If you have a set of "micro-services" that are always deployed together and upgraded together, you might **want to use an umbrella chart**[19] for them.

Argo CD applications should generally model something that requires individual deployments and updates. If you have several Argo CD applications that you always want to be managed together (but still want to deploy and update individually), a better choice is the **app-of-apps pattern**[20].

If several applications need to be deployed to different or similar configurations, then **Application Sets**[21] are the recommended approach.

So, any time you want to group several applications, ask the following questions:

1. Are those applications always deployed and upgraded as a single unit?
2. Are those applications related in a business or technical manner?
3. Do you want to use different configurations for different clusters for these applications?
4. Is this combination of applications always the same? Do you sometimes wish to deploy a subset or superset of them?
5. Are these applications managed by a single team or multiple teams?

If you are unsure where to start, **looking at Application Sets**[6] is always the best place to start. Check also anti-pattern 19 (attempting to version application CRDs).

8. Abusing the multi-source feature of Argo CD

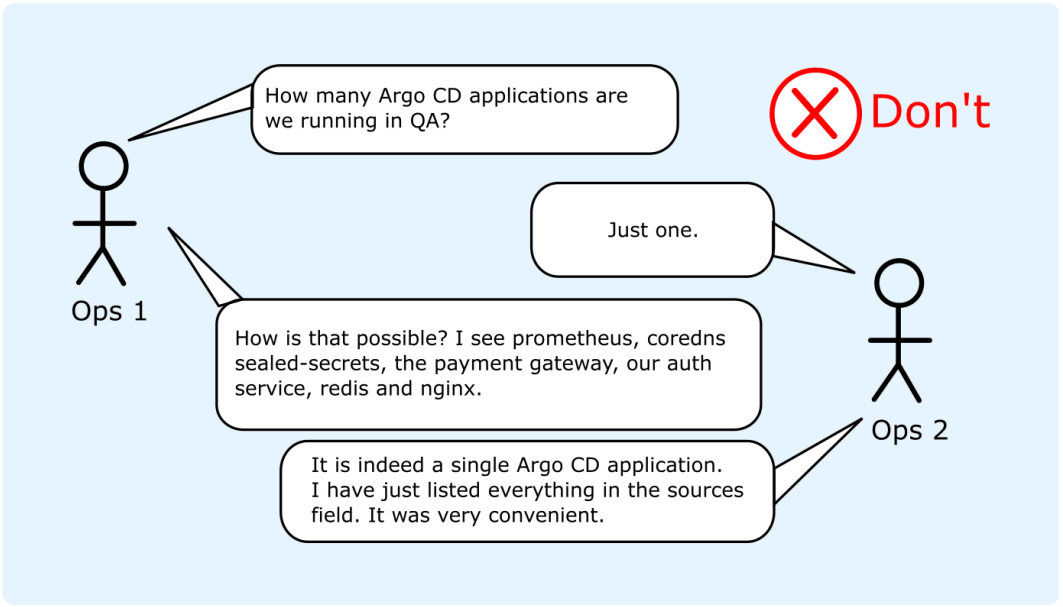
This is a close relative of the previous anti-pattern. The **Multi-source feature**[22] of Argo CD was one of the most requested features in the project's history. The feature was created to solve a single scenario:

1. You wish to use an external Helm chart that your organization does not host
2. You want to use your own Helm values and still store them in your own Git repository
3. You need a way to instruct Argo CD to combine the external Helm chart with your own values

The feature is finally implemented in Argo CD and you can use it as follows:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
spec:
  sources:
  - repoURL: 'https://prometheus-community.github.io/helm-charts'
    chart: prometheus
    targetRevision: 15.7.1
    helm:
      valueFiles:
      - $values/charts/prometheus/values.yaml
  - repoURL: 'https://git.example.com/org/value-files.git'
    targetRevision: dev
    ref: values
```

Unfortunately, Argo CD does not limit the number of items you can place in the "sources" array. Several people have misunderstood this, abusing the feature to group multiple (often unrelated) applications.



Don't fall into this trap. The multi-source feature was never designed to work this way, and several standard Argo CD capabilities will either be broken or not work at all if you use multi-sources as a generic application grouping mechanism.

The correct way to group applications is with Application Sets. If you want to use multi-source applications with Helm hierarchies, we have also written an **extensive guide**[13].

9. Not splitting the different Git repositories

If you look at any Kubernetes application from a high level, it is comprised of 3 distinct types of files:

1. The source code
2. Kubernetes manifests (deployment, service, ingress, etc)
3. Argo CD application manifests (or application sets)

We have already explained that, as a best practice, **source code should be separate**[23] from the manifests. If you are in a big organization, it makes sense to split Kubernetes manifests from Argo CD manifests.

If you keep all manifests in a single Git repository, you will have issues with both CI (Continuous Integration) and CD (Continuous Deployment) phases. Your CI system will auto-build application code when a manifest changes, and Argo CD will sync applications when a developer changes the source code.

There are several workarounds for these scenarios, but why try to solve a problem that shouldn't exist in the first place?

We have also seen several variations of the same pattern that make the deployment process even more complicated. A classic scenario to avoid is the following:

1. The source code of the application is in Git repository A
2. The Helm manifest is in Git repository B
3. Only the Helm values for the different environments are also stored in Git repository A

The assumption here is that Helm values are close to the source code that developers need to change. In reality, it never makes sense to have access to Helm values without also having access to the Helm chart that uses them. Either assume developers know Helm and show them everything, or assume they don't care about Kubernetes at all and offer a different abstraction that hides Argo CD from them.

Yet another problem is mixing Kubernetes and Argo CD manifests in the same repository, but instead of using separate folders, you hardcode Kubernetes information into Argo CD manifests. This is described in detail in anti-patterns 17 and 18.

10. Turning off auto-sync and self-heal

Migrating to Argo CD is a major undertaking, especially for organizations that have invested considerable effort in traditional pipelines. After all, you could argue that Argo CD replicates what is already possible with an existing Continuous Integration system:

1. A change happens in Git in a manifest
2. A separate process picks up the Git event
3. The process uses `kubectl` (or a custom script) to apply the changes in the cluster

This line of thinking couldn't be further from the truth, as Argo CD also works the other way around. It monitors changes in the cluster and compares them against what is in Git. Then you can choose to either review those changes or discard them altogether.

This means that Argo CD **solves the configuration drift problem once and for all**[24], something that is **not** possible with traditional CI solutions.

But this advantage only exists if you let Argo CD do its job. Some organizations turn off the **auto-sync/self-heal**[25] behavior in Argo CD in an effort to "lock down", or fully control, production systems.

This is a bad choice because production systems are precisely the place you want to avoid configuration drift. Manual changes made in production (during hotfixes or other debugging sessions) are among the most significant factors contributing to failed deployments.

We recommend that you switch on auto-sync/self-heal for all your systems, both production and non-production.

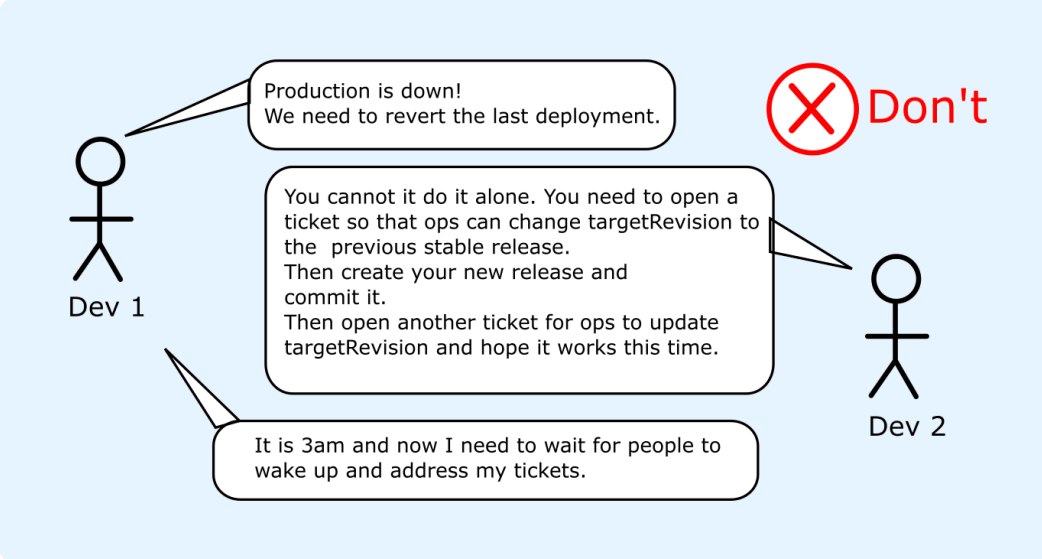
Locking down a system should not be done in Argo CD; it should be enforced at the cluster and Git levels. The most obvious solution is to reject any direct commits to the Git repository that controls your production system and allow developers to create Pull Requests only, which must pass several manual and automated checks before landing in the mainline branch that Argo CD monitors.

If you switch off auto-sync/self-heal you are missing the number one advantage of moving to Argo CD from traditional pipelines (eliminating configuration drift).

11. Abusing the targetRevision field

Promoting applications when adopting Argo CD is one of the biggest challenges for developer workloads. People see the **targetRevision field**[26] in the Argo CD application and assume it is a promotion mechanism.

The first issue is when teams use semantic version ranges to force Argo CD to update an application automatically to a newer version. The second issue is if they continuously update the targetRevision field to **different branch names**[27] or attempt to implement "**preview environments**[28]" by pointing an Argo CD application to a temporary developer/feature branch.



You can find out more in our **complete guide to the issues of abusing targetRevision**[29].

In general, we recommend you always use HEAD in the targetRevision field, which is also the default value.

12. Misunderstanding immutability for container/git tags and Helm charts

This is not an anti-pattern with Argo CD per se, but it is closely related to the targetRevision choices, as explained in the previous anti-pattern.

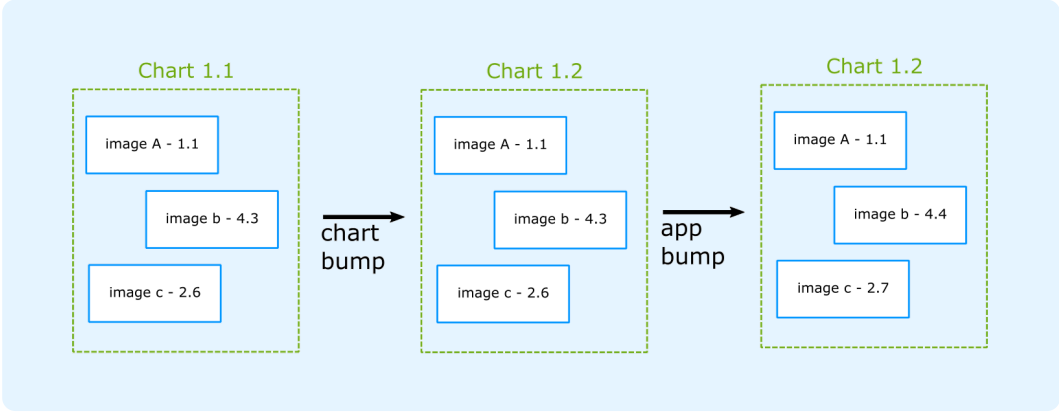
We have seen several cases of people adopting Argo CD without first understanding the foundations (Helm, container registries, git tags). Several times, people use a specific git tag or Helm version in Argo CD without realizing that:

1. Git tags seem to be immutable, but can be deleted and recreated with the same name
2. Helm chart versions are mutable, and this is how Helm was designed
3. Container tags are mutable by default

Let's take these points one by one.

Container tags are mutable. You can create a container called `my-app:v1.2`, change something, and push another container with the same tag. Just because you see the same container tag doesn't mean it's the same application. Some binary repository implementations don't allow you to do this, but this is not always the default setting.

Helm chart versions are also mutable. You can change the contents of a Chart version and use the exact same version. Again, this is how Helm charts work.



In fact, Helm offers a property called `appVersion` which can store the "application" version. So you can have a Helm chart with 3 "version" fields:

- The container image (mutable by default)
- The appVersion field (mutable)
- The Chart version (mutable)

So, unless you control how developers work with code and manifests and also configure your Helm chart repository correctly, you don't really know if a Helm chart version contains the same thing as another chart with the same version.

Git tags can also be overwritten. You can see this very easily with any GitHub repo with default settings.

```
git tag -a v1.2 -m "first tag"
git push --tags
echo "A change" >> README.md
git tag -d v1.2
git push origin :v1.2
git tag -a v1.2 -m "first tag"
git push --tags
```

Here we just pushed the same tag (v1.2) twice, but with different contents. So if you were using this tag in the targetRevision field of Argo CD, you now have the "same" application without actually having the same contents.

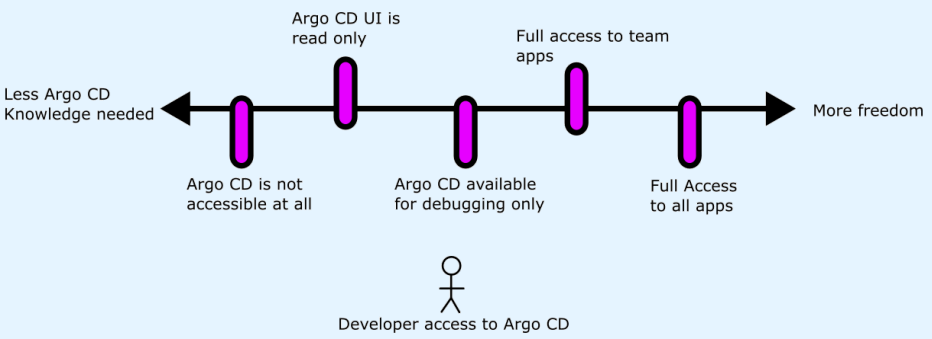
The end result is that using tags and Helm chart versions in Argo CD doesn't really restrict your developers unless you actively set up the rest of the ecosystem (Git repositories, Helm repos, and binary artifact managers) to work with immutable data as well.

Never assume you have a "locked-down" Argo CD system when the rest of the ecosystem allows developers and operators to create stuff with the same container/Helm/Git version.

13. Giving too much power (or no power at all) to developers

When adopting Argo CD, you need to decide how much power and exposure you want to give developers. On one end of the spectrum, we see installations where developers have full access to the Argo CD UI and can sync/deploy their applications at will. On the other end, we see locked-down installations where developers have very little power, or where Argo CD is hidden from them.

It is essential to understand that despite these two extremes, there are several choices in the middle. First, you can configure the Argo CD access (and UI) to show only applications specific to each team. You can even create advanced scenarios that show applications from other teams in read-only mode.



In our **Argo CD RBAC guide**[30], we explain how Argo CD RBAC works and how you can show content specific to a developer team.

There is no right or wrong answer here, but you need to balance the flexibility you want to offer developers with the security you wish to provide.

On a related note, we have already explained that developers **don't really care about Argo CD manifests**[31], and they shouldn't have to install Argo CD for local testing (see anti-pattern 6).

So a recommended workflow would be the following:

1. Developers can test and deploy their applications locally without using Argo CD at all
2. When they have created a feature branch, this should **be converted to a preview/temporary environment using the Pull Request generator**[28] without any human intervention
3. Once the feature is ready, it will be deployed to production simply by merging the Pull request
4. A system designed for promotion should propagate the changes to the next environment (check anti-pattern 30).

Ideally, developers should only interact with Argo CD in the last phase, and only in the event of a failure. In the happy path scenario where everything works as planned, developers shouldn't have to debug anything with Argo CD.

14. Referencing dynamic information from Argo CD/ Kubernetes manifests

This is a more specialized anti-pattern related to number 2 (creating applications dynamically). The second **GitOps principle**[1] explains that the desired state of your system should be immutable, versionable, and auditable.

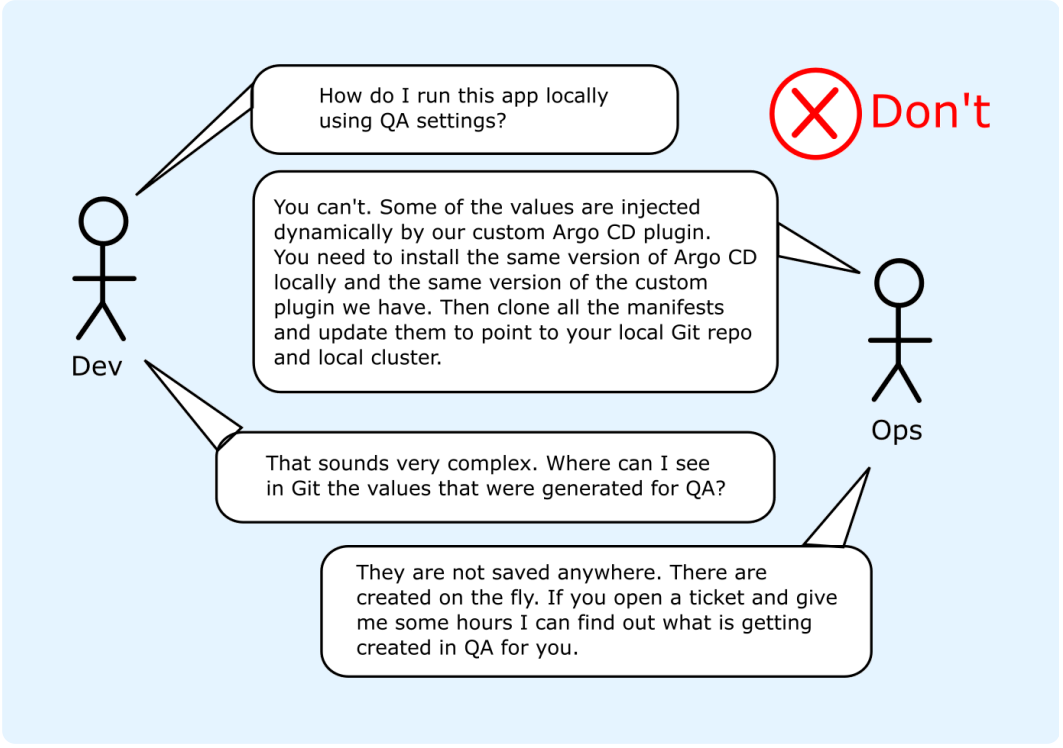
This is true only if you store **everything** the application needs in Git (or your chosen storage method). For Kubernetes/Argo CD manifests, this means that all values used should be static and known in advance.

It is okay if you want to post-process your manifests in some way, as long as it is done in a repeatable manner, or you also save the resulting manifest in Git.

The problem starts when your configuration is not known in advance but requires real-time access to something else.

The best example to illustrate this is the **Helm lookup method**[32], which mutates the Helm chart to a different value without knowing in advance what this value is. This is problematic with Argo CD because having access only to the application manifests is no longer enough to run the application.

Like anti-patterns 17 and 20, this also makes life difficult for developers as they cannot run the application locally anymore (anti-pattern 6).



Note that the only exception to this rule is secrets. Even though you **can store encrypted secrets in Git**[33], it is also ok **to reference them from an external source**[34].

But make sure to understand **the difference between referencing secrets from manifests versus injecting secrets**[35] into manifests.

Anti-patterns: Configuration management

This section is about one of the hottest topics in application management with GitOps. We will explain how to map your Helm values or Kustomize overlays across different environments. This is one of the most controversial decisions in application management. We see several teams that don't understand that Argo CD is not a generic configuration platform. Many people assume that injecting Helm values into Argo CD manifests is a well-accepted practice. It's not, and we advise against it.

Here we also talk about the "Helm sandwich". This is a questionable practice where you have two layers of Helm manifests, one for the Argo CD applications and one for the Kubernetes manifests of the applications themselves. We strongly suggest you do **not** follow this pattern. It's especially harmful to developers as it destroys local testing and makes it very difficult for them to understand how to run a Kubernetes application.

15. Writing applications instead of Application Sets

The Application CRD is the main entity in Argo CD that links a cluster and a GitHub repository. If you have a small number of applications or work in a home lab environment, it is okay to write these files by hand. But for any organization's production installation of Argo CD, you shouldn't write Application files by hand. In fact, you shouldn't deal with Application files at all.

The recommendation is **to use Application Sets directly**[21].

In a big organization, you will rarely have to deal with a single application on its own. Instead, you'll likely work with a group of applications. Some examples are:

- A set of applications that go to the same cluster
- A set of applications managed by the same team
- A set of applications that share a configuration
- A set of applications that should be deployed/updated as a unit.

Application Sets implement this grouping and automate the otherwise tedious YAML needed. For example, if you have 4 applications and 10 clusters, you don't really want to create and manage 40 YAML files by hand.

1. You create a single cluster generator that iterates over your clusters
2. You create 4 folders for the 4 applications
3. The Application set automatically creates and runs all 40 combinations for you

Some people resist using Application Sets because they think it's yet another abstraction that hides their real Application manifests.

This used to be true, but in the latest Argo CD releases **the CLI allows you to render an application set**[36] and see exactly which applications will be created. You can run this command either manually or as part of a CI pipeline (when a pull request is created) to get instant visibility into what will change.

In general, though, as we will see later, application set files should be created only once. They are not a deployment format (anti-pattern 19), and you shouldn't have to edit Application Sets for simple operations.

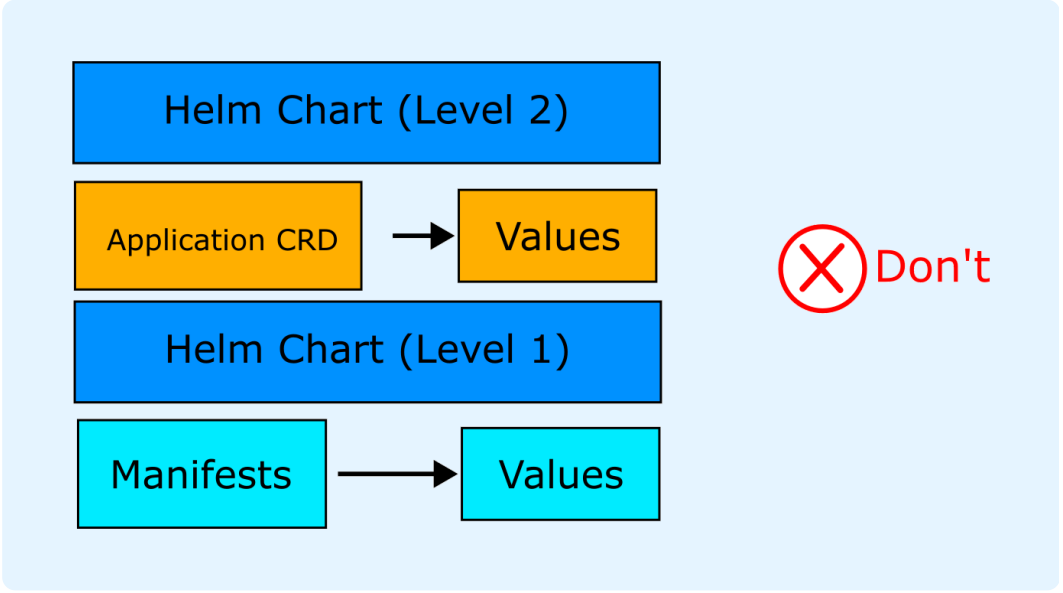
You can read more in the **dedicated guide to Application Sets with Argo CD**[6].

16. Using Helm to package Applications instead of Application Sets

This anti-pattern is the "Helm sandwich". This happens when:

1. A set of Kubernetes manifests is packaged as a Helm chart
2. The Helm chart is referenced from an Argo CD application Manifest
3. The Argo CD application manifest is packaged in another Helm chart

Essentially, there are two instances of Helm templating on two different levels.

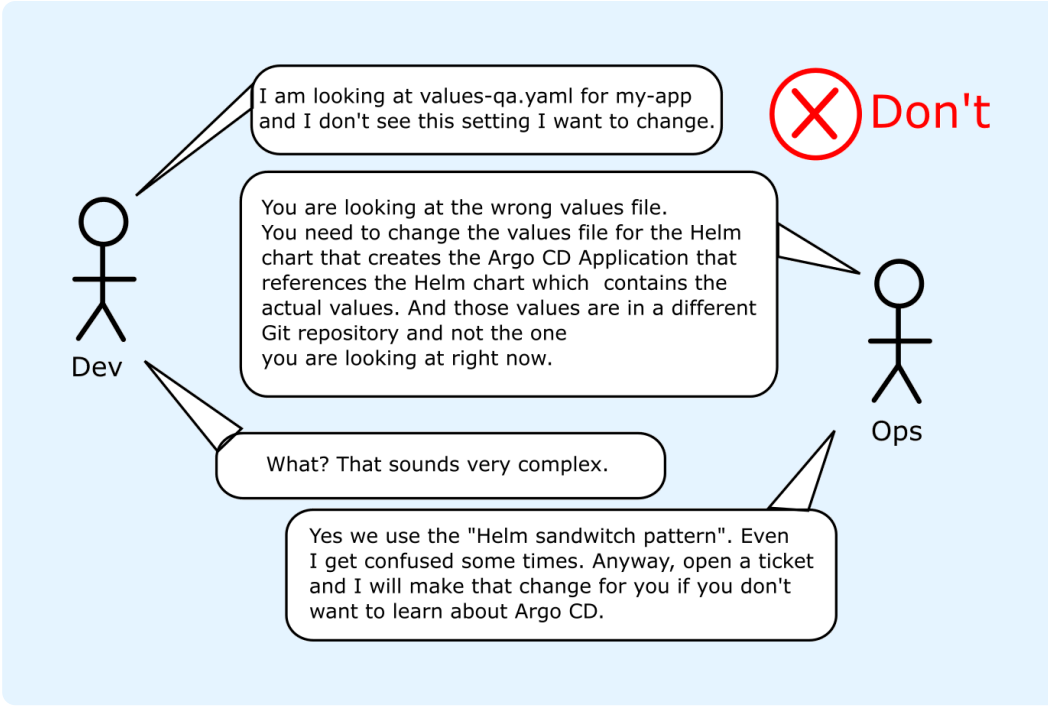


Teams that adopt this anti-pattern are familiar with Helm and assume that, if it works well for plain Kubernetes manifests, it would also work well for Argo CD application manifests.

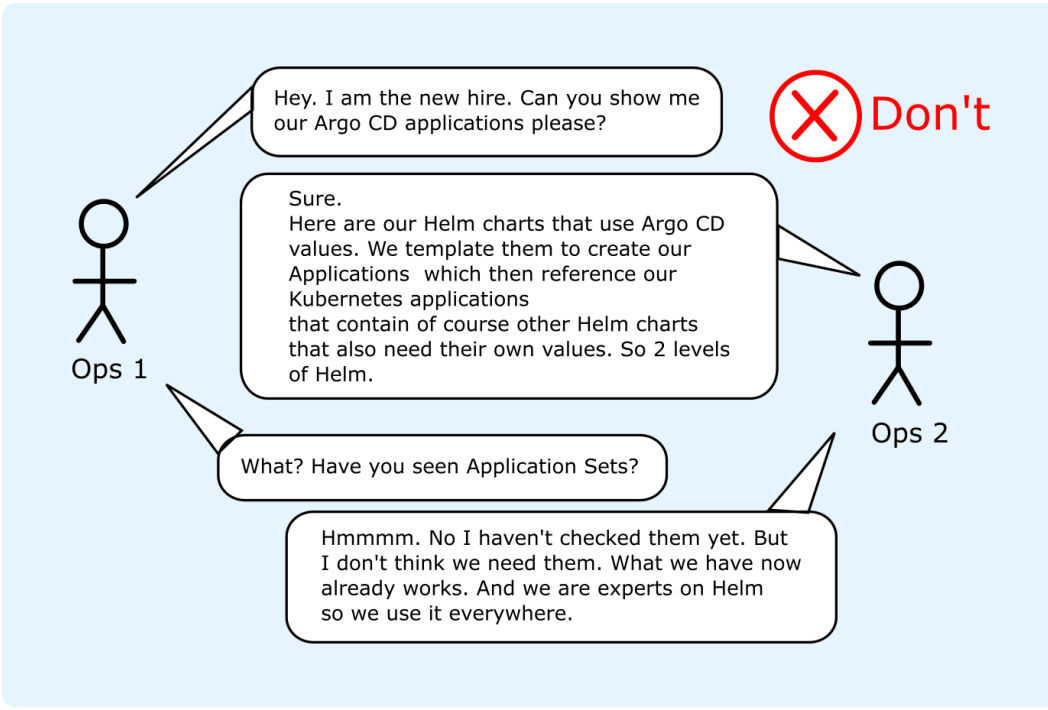
So why is this approach an anti-pattern? Using Helm for applications is not an issue on its own, but it is the start of several other anti-patterns:

1. Because Helm charts have a version, people try to version Applications (Anti-pattern 19)
2. Chart version numbers often result in abusing target Revision for promotions (Anti-pattern 11)
3. If you use Helm everywhere, it is super easy to hard-code Helm values in Application manifests (Anti-pattern 17)
4. People miss all the Argo CD specific features of application sets (Anti-pattern 15)
5. Distributing applications to different clusters happens with snowflake servers instead of the cluster generator (Anti-pattern 21)

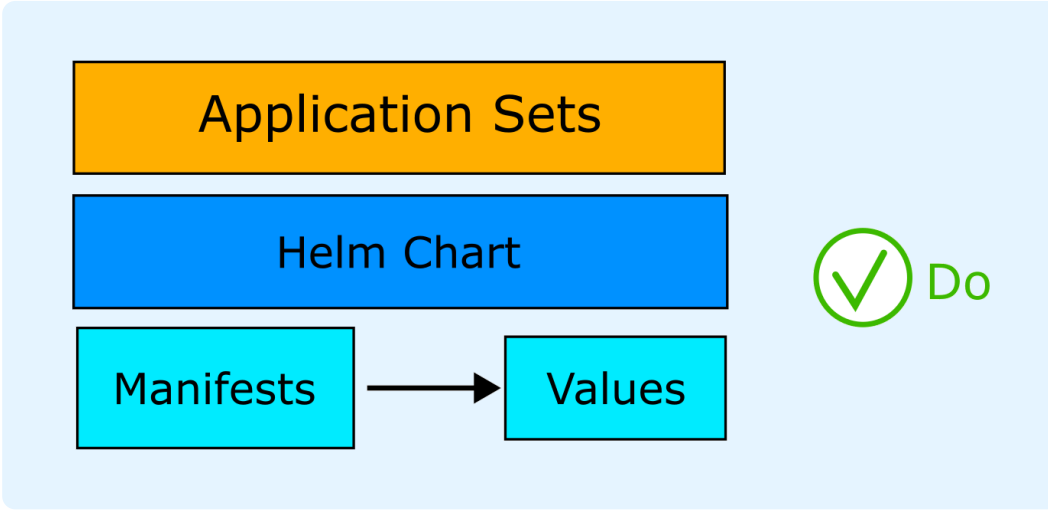
The biggest problem is that a "Helm sandwich" completely ruins the developer experience. You now have 2 levels of Helm values, and this is a recipe for disaster:



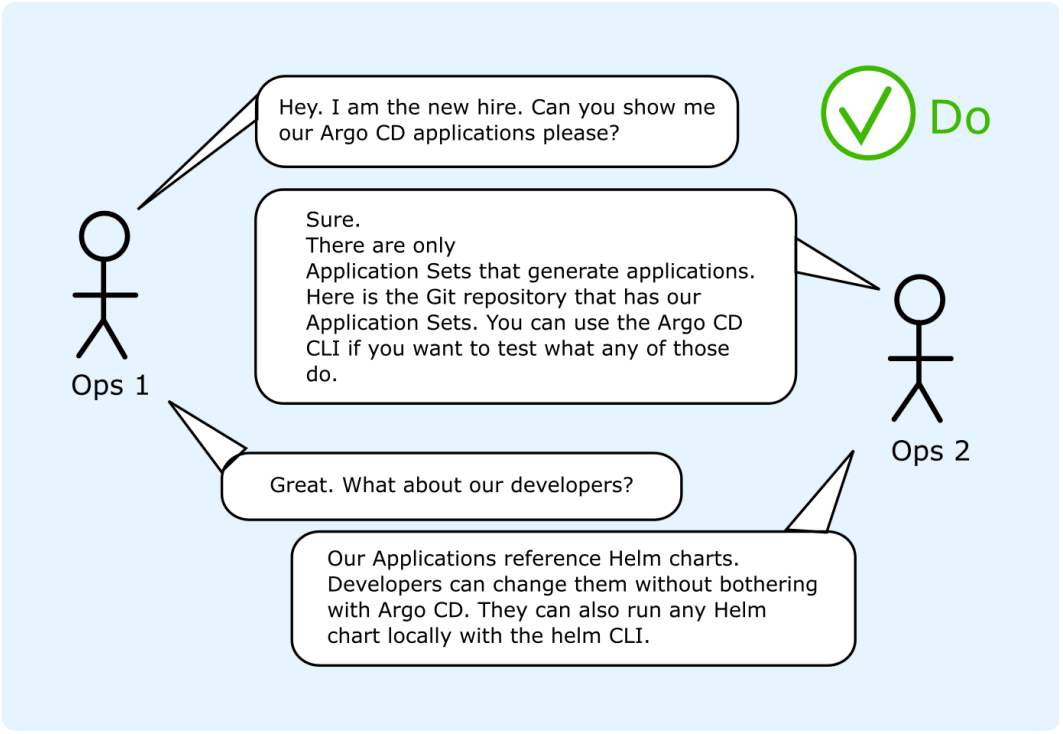
And it's not just developers who suffer. Operators and admins also suffer because packaging applications in a Helm chart creates an extra level of indirection. This isn't needed and makes your life more difficult, as Argo CD wasn't designed for this "Helm sandwich".



Our recommendation, of course, is to use Application Sets to configure Applications, as already explained in the previous anti-pattern.



Application Sets are the preferred way to generate Applications and support the same templating functions as Helm charts. If you can template it with Helm, you should be able to template it with Application Sets.



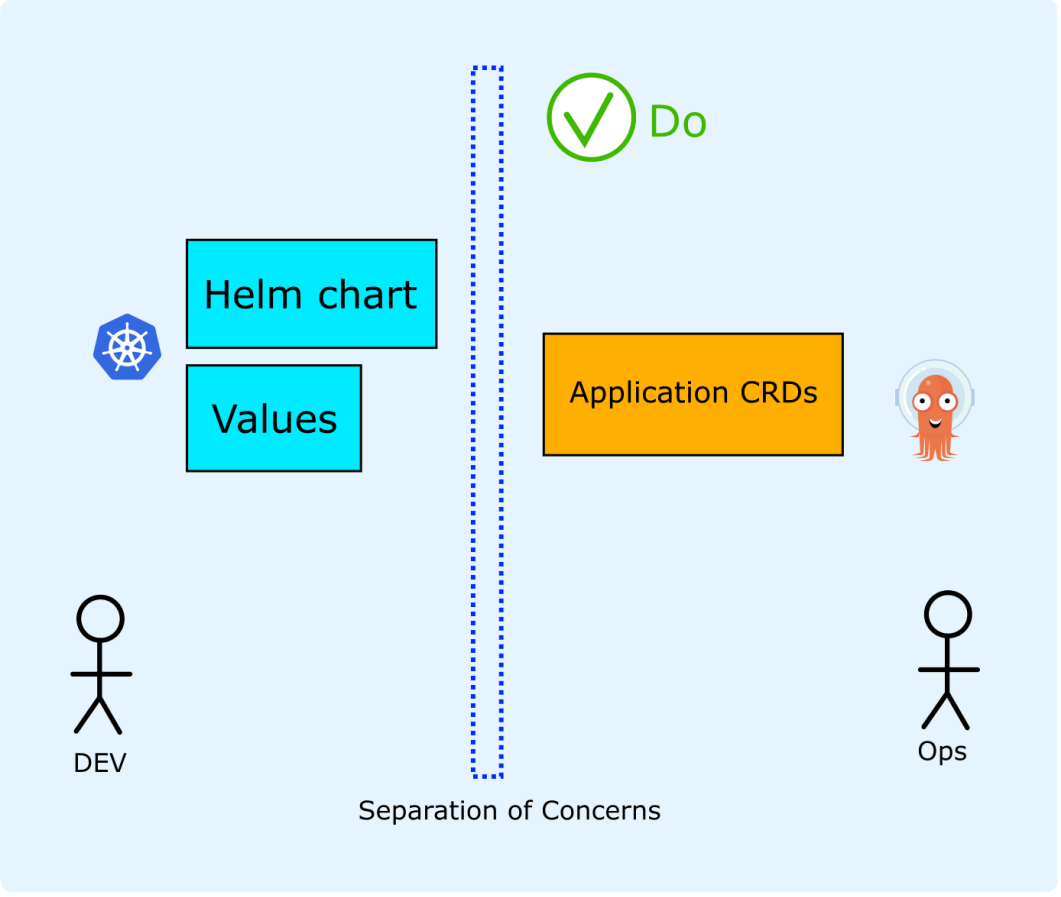
Using the "Helm sandwich" pattern makes the process more complex for everybody involved in the software lifecycle.

For more information, see our **Application Set guide**[6].

17. Hardcoding Helm data inside Argo CD applications

If you follow the advice we outlined in anti-pattern 4 you should have a clear separation between your Helm charts and your Argo CD manifests.

The Helm charts can be used independently (even by developers) and contain all application settings. The Argo CD application manifest simply defines where this application runs, and only operators need to change this file.



Those two kinds of files also have different lifecycles:

- Helm values are expected to change all the time (even by developers)
- Argo CD application manifests are created once and never changed again.

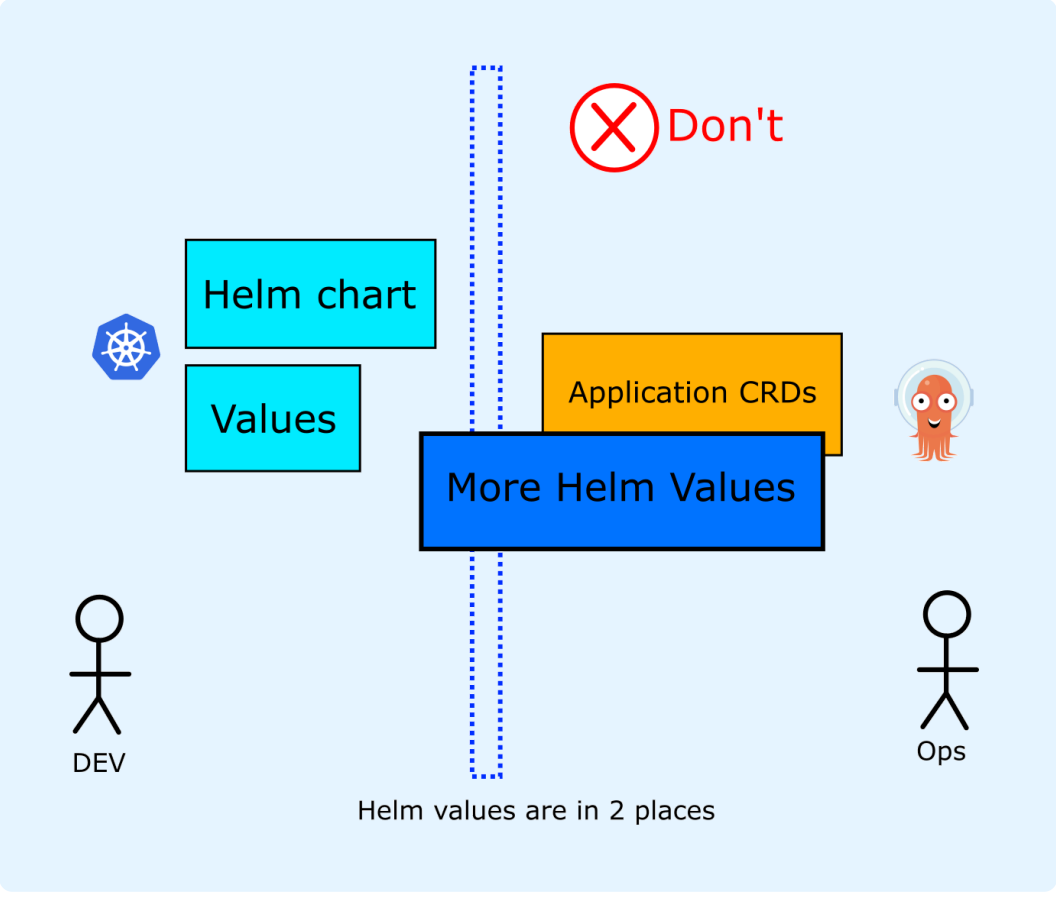
With Argo CD, it is possible to hardcode Helm information inside an Application CRD. But just because you can do this doesn't mean it is a good idea to use this feature.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-helm-override
  namespace: argocd
spec:
  project: default

  source:
    repoURL: https://github.com/example-org/example-repo.git
    targetRevision: HEAD
    path: my-chart

    helm:
      # DON'T DO THIS
      parameters:
      - name: "my-example-setting-1"
        value: my-value1
      - name: "my-example-setting-2"
        value: "my-value2"
        forceString: true # ensures that value is treated as a strin

      # DON'T DO THIS
      values: |
        ingress:
          enabled: true
          path: /
          hosts:
            - mydomain.example.com

      # DON'T DO THIS
      valuesObject:
        image:
          repository: docker.io/example/my-app
          tag: 0.1
          pullPolicy: IfNotPresent
```

```
  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
```

The big problem with this file is that you are mixing 2 different kinds of information with 2 different lifecycles. The application CRD is mostly interesting to operators and administrators, while Helm information is interesting to developers and is likely to change a lot.

By mixing this information you make manifests harder to understand for everybody.

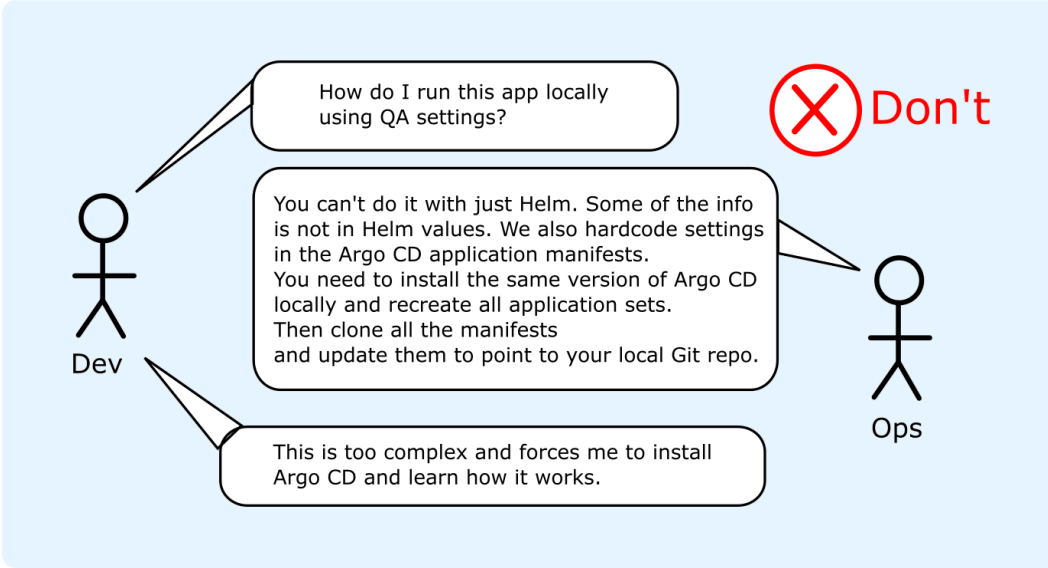


Now you have Helm information in two places (`helm values` and the `helm` property inside the Argo CD application manifest). It is tough to understand what settings exist where and how to audit deployment history. Argo CD application manifests must now change frequently, especially if they define container images.

This manifest mixing is also the root of several other anti-patterns, such as:

- Using Applications as a unit of work (anti-pattern 19)
- Abusing the targetRevision field for promotions (anti-pattern 11)
- Not understanding how Helm hierarchy works (anti-pattern 4)
- Assuming that developers need Argo CD (anti-pattern 6)

The last point is crucial for developers. If you follow this practice, you have completely destroyed local testing for developers, as they can no longer run the application independently.



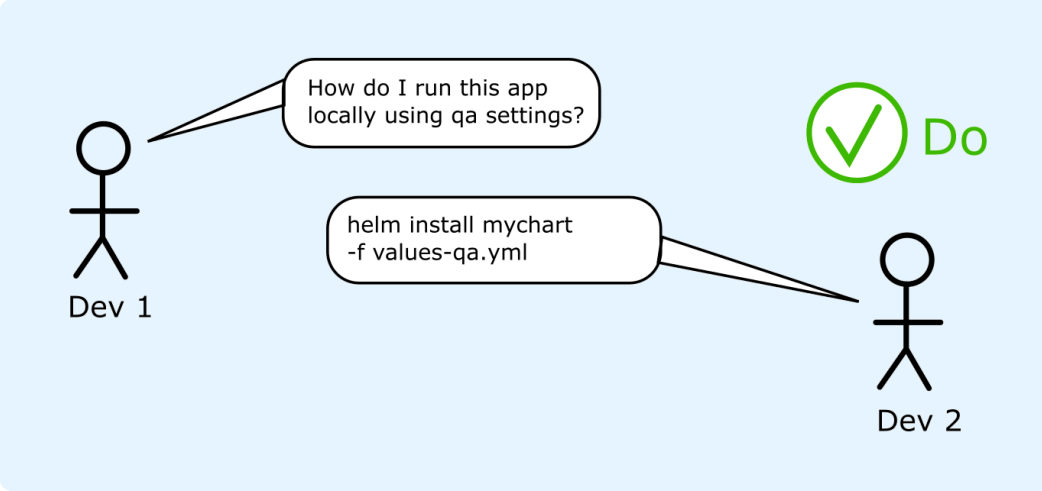
Even though we speak only about developers and operators here, several other scenarios will make this approach difficult to use

1. Security teams will have a hard time understanding the settings for each application
2. Your CI system cannot upgrade images just on Kubernetes manifests anymore. It also needs to look at Argo CD manifests and check if image definitions exist there as well
3. It couples your Applications to specific Argo CD features

The correct solution is to store all Helm information in Helm values:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-helm-override
  namespace: argocd
spec:
  project: default

  source:
    repoURL: https://github.com/example-org/example-repo.git
    targetRevision: HEAD
    path: my-chart


    helm:
      ## DO THIS (values in Git on their own)
      valueFiles:
      - values-production.yaml
  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
```

Now there is a clear separation of concern between developers and operators.



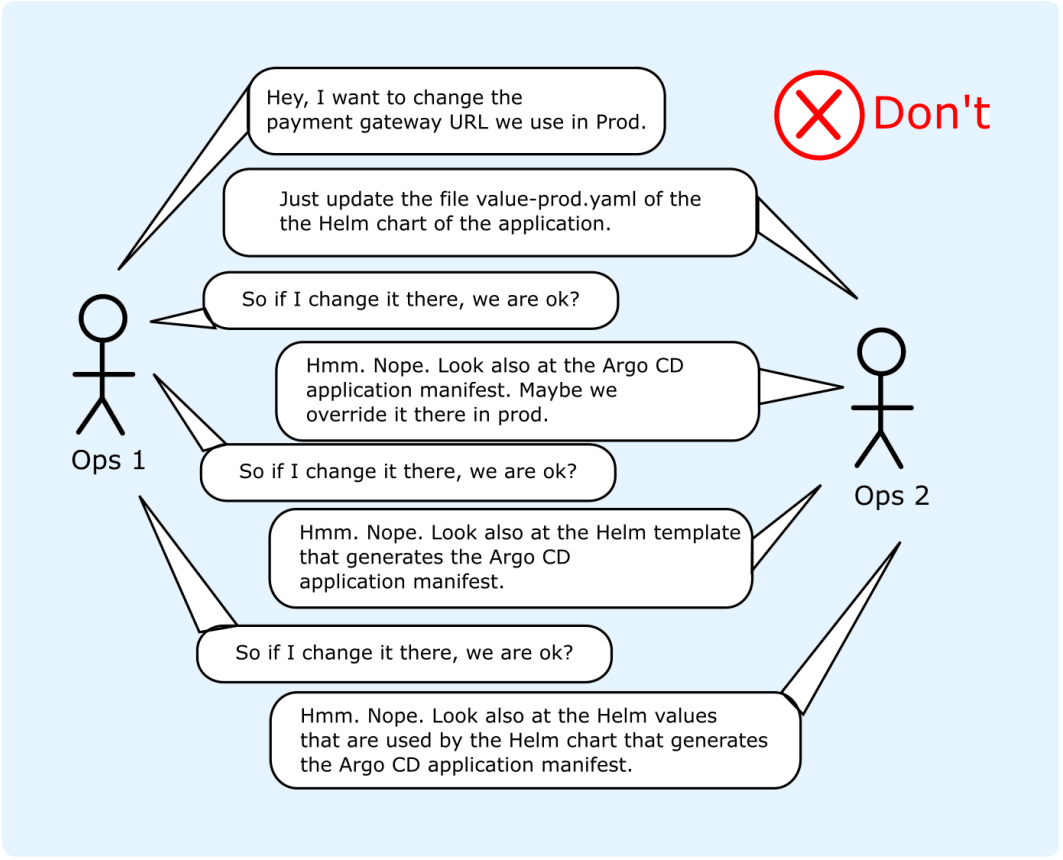
People who want to know an application's settings can look at Helm values, while people who want to see where an application is deployed can look at Argo CD manifests. It is possible to run a Helm application without Argo CD.

For more information about the different types of manifests and how to split them see **our Application Set guide**[6].

This anti-pattern is even worse if coupled with the previous anti-pattern (the Helm sandwich).

Now you have 3 places where configuration settings can be stored:

1. The Helm values of the chart that gets deployed
2. The helm property inside the Application Manifest that references the Helm chart
3. The Helm template that renders the Application manifest before getting passed to Argo CD



The result is a nightmare for anybody who wants to understand how an application gets deployed.

18. Hardcoding Kustomize data inside Argo CD applications

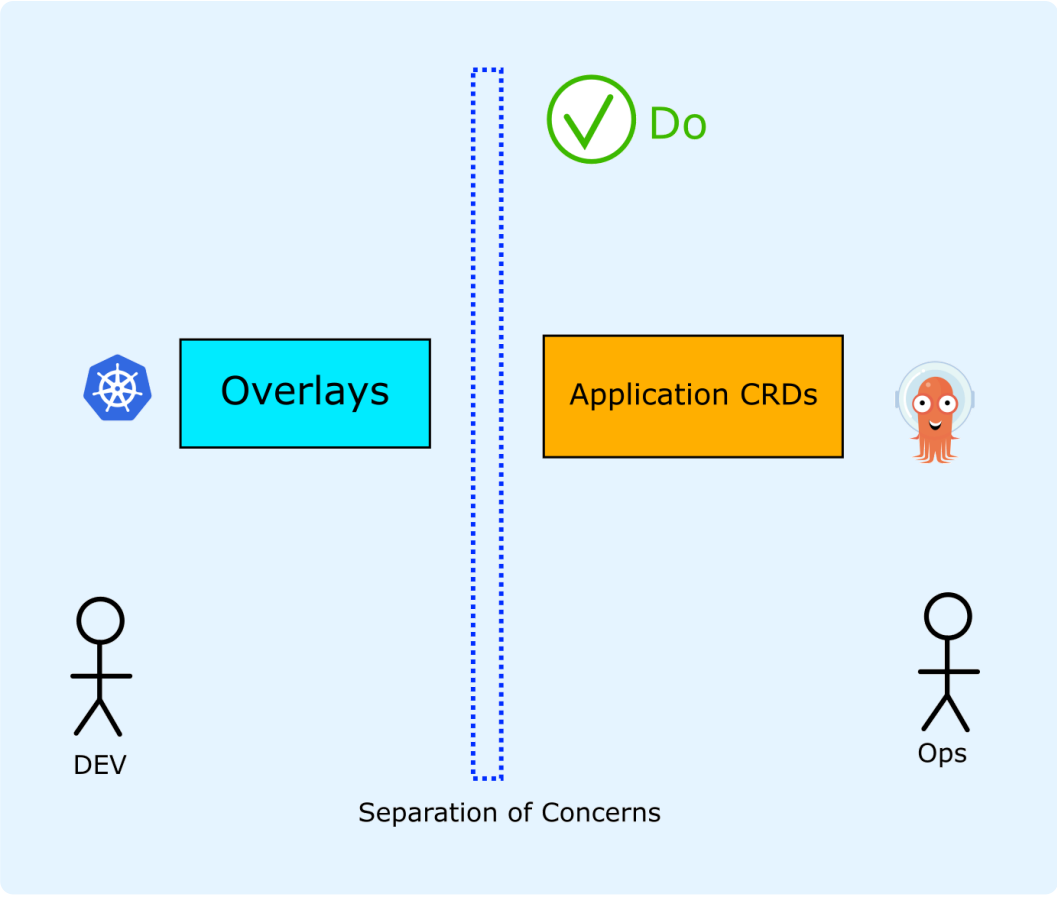
This is the same anti-pattern as the previous one but for Kustomize. Again, for convenience, Argo CD allows you to hardcode Kustomize information inside an Application YAML:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-kustomize-override
  namespace: argocd
spec:
  source:
    repoURL: https://github.com/example-org/example-repo.git
    targetRevision: HEAD
    path: my-app

    # DON'T DO THIS
    kustomize:
      namePrefix: prod-
      images:
      - docker.io/example/my-app:0.2
      namespace: custom-namespace


  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
```

It is the same problem as before, where you are mixing different concerns in a single file. Kustomize information should only exist in Kustomize overlays:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-proper-kustomize-app
  namespace: argocd
spec:
  source:
    repoURL: https://github.com/example-org/example-repo.git
    targetRevision: HEAD
    ## DO THIS. Save all values in the Kustomize Overlay itself
    path: my-app/overlays/prod


  destination:
    server: https://kubernetes.default.svc
    namespace: my-app
```

As explained before, this helps developers during local testing. A developer can run `kustomize build my-app/overlays/prod` and get the full configuration of how my-app runs in production. No knowledge of Argo CD is required, and no local installation of Argo CD is needed.



Developers can define *how* an application runs (its settings), while operators can decide *where* (which cluster) the application is deployed.

At the same time, several supporting functions become straightforward:

- Git history of the overlays is the same as the deployment history
- There is only a single source of truth for configuration (the overlays)
- Developers don't need to know how to use Argo CD at all
- It is very easy to diff settings between environments.

A detailed example with Kustomize overlays **is available in our GitOps promotion guide**[18].

19. Attempting to version and promote Applications/Application Sets

An Argo CD application is just a link between a cluster and a Git repository. There is NO version field in the Application CRD. An application manifest is neither a packaging format nor a deployment artifact. The same is also true for Application Sets. You never *deploy* application sets. You use them to auto-generate application manifests. Application Sets have no version on their own.

The lack of a version field is not a big problem because the expectation for both Applications and Application Sets is that you create them once and then never update them again. So, a version field is unnecessary as all Argo CD Applications are considered static.

However, we see several teams that try to use Applications as the unit of work (see anti-pattern 7) or continuously update those files in the targetRevision field (see anti-pattern 11).

At this point, teams often try to create their own versioning on top of the Argo CD manifests. This fails because Argo CD was never designed this way.

The same is true for promotions. You cannot really "promote" an Argo CD application from one cluster to the next. It doesn't work this way. You can only promote values that are referenced from one application to the next (essentially copying them).

The way promotions work in Argo CD is the following:

1. There is an Argo CD application manifest in QA that points to a Helm chart or Kustomize overlay
2. There is an Argo CD application manifest in Staging that points to different Helm values or Kustomize overlays
3. When it is time to promote, you *copy* the Helm values or Kustomize overlay from the QA files to the Staging files.
4. The Argo CD application manifests are not affected in any way. They are exactly the same as they were before.

See also the related anti-patterns of hardcoding Helm (anti-pattern 17) or Kustomize data (anti-pattern 18) inside applications.

If you find yourself constantly updating Argo CD application manifests (or Application Sets) you have fallen into this trap. Your Argo CD manifests should be created only once and never touched again. No process can be easier than not needing a process at all (to update Application manifests).

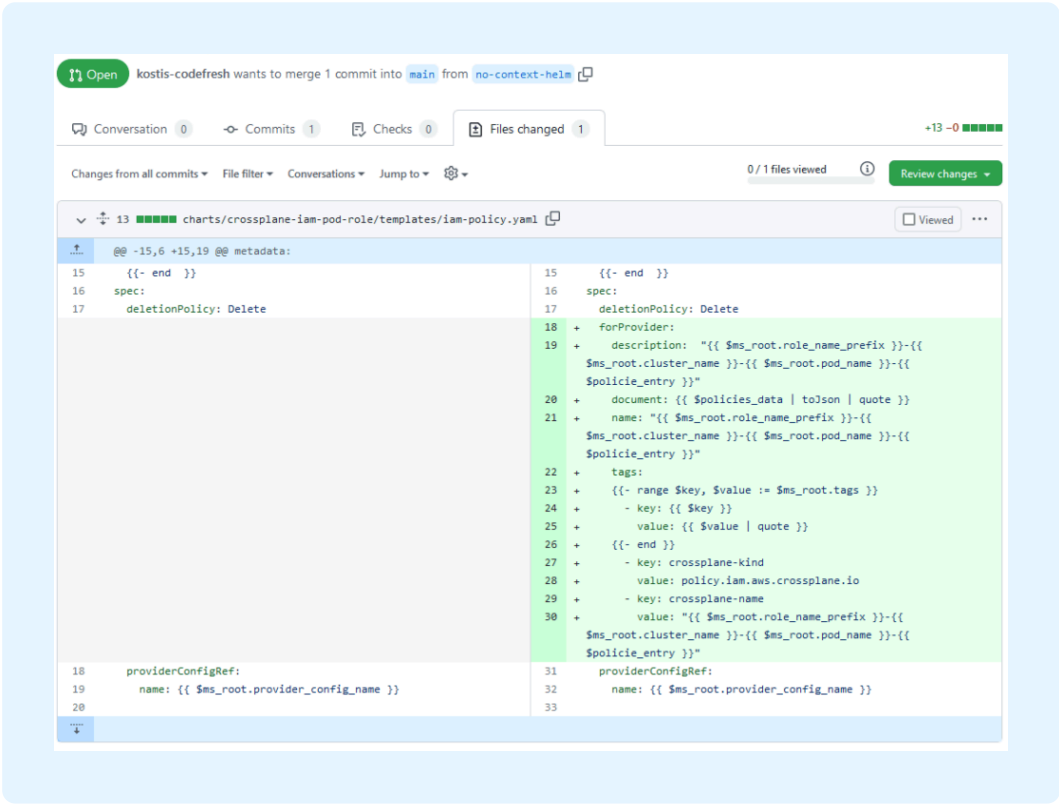
On a related note, you can use Argo CD with **Octopus Deploy**[37] to solve the problem of promoting applications from one cluster to another.

20. Not understanding what changes are applied to a cluster

One of the main benefits of using Argo CD is that all your Git tools work out of the box and you can reuse your code review process for your Kubernetes manifests.

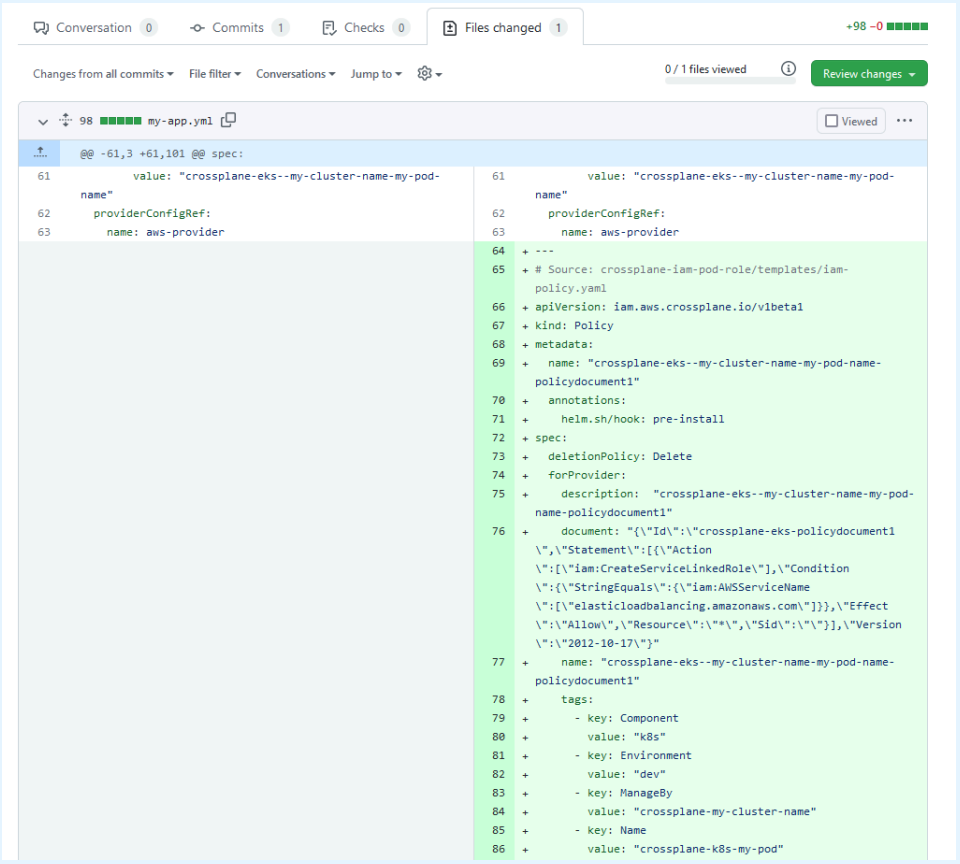
The most basic capability in GitHub is the ability to store anything by creating a Pull Request before merging any changes. This allows humans to review what will change and also run any automated tools to verify and validate the changes.

Unfortunately, this review process will not really work on files such as Helm charts, Application Sets, or Kustomize Overlays. Let's assume you need to review the following change:



You need to manually run Helm in your head to understand what is happening here. Humans never want to do that. One approach is to pre-render all your manifests so that reviews only occur for the final content. There are several alternatives you can consider, such as having your CI system render manifests on the fly to show you what will actually happen.

Here is the exact change as before, but this time on the final chart:



We have written a **full guide on how to preview and diff your Kubernetes manifests**[38] that offers several other approaches.

Specifically for Argo CD, you should also check **Argo Diff Preview**[39] and also understand that you can use the **Argo CD CLI to render your Application Sets to Application CRDs**[36].

Anti-patterns: Cluster management

In this section, we will see several day-2 operations for large Argo CD installations. The most important challenge is managing multiple Argo CD clusters with similar (but not identical) configurations effectively. Here we see several teams that miss completely the **cluster generator**[40]. They then try to model their applications in the same way that they handle Ansible and Terraform. The same is true for trying to create preview/temporary environments without understanding what is offered by the **Pull Request generator**[41].

Another highlight of this section is trying to perform database migration procedures during the sync process. This is another questionable practice based on tradition and inertia. As explained multiple times, Argo CD and GitOps require a new mindset about application deployments. The "lift-and-shift" shortcoming apply to Argo CD applications as well.

21. Using ad-hoc clusters instead of cluster labels

If you have a large number of clusters that you wish to manage with Argo CD, your first question is always **whether to use a single Argo CD instance or multiple ones**[42].

Once you answer this question the next step is to understand how you can distribute different applications across different clusters. The answer to this question is Application Sets.

Unfortunately many teams don't understand **how the cluster generator works**[40] and instead try to create ad hoc cluster configurations (the pet versus cattle philosophy).

Examples to avoid often appear like this:

```
## DO NOT DO THIS
- merge:
      mergeKeys:
        - app
      generators:
        - list:
            elements:
              - app: external-dns
                appPath: infra/helm-charts/external-dns
                namespace: dns
              - app: argocd
                appPath: infra/helm-charts/argocd
                namespace: argocd
              - app: external-secrets
                appPath: infra/helm-charts/external-secrets
                namespace: external-secrets
              - app: kyverno
                appPath: infra/helm-charts/kyverno
                namespace: kyverno
        - list:
            elements:
              - app: external-dns
                enabled: "true"
              - app: argocd
                enabled: "true"
              - app: external-secrets
                enabled: "false"
              - app: kyverno
                enabled: "true"
    selector:
      matchLabels:
        enabled: "true"
```

Or this:

```
## DO NOT DO THIS
Version: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: my-staging-cluster
  namespace: argocd
spec:
  goTemplate: true
  generators:
    - matrix:
        generators:
        - git:
            repoURL: '<url>'
            revision: HEAD
            files:
              - path: customConfig/base.yaml
              - path: customConfig/{{ .Values.domainId }}/*.yaml
        - list:
            elements:
              - appName: 'auth'
              - appName: 'search'
              - appName: ‘billing'
              - appName: ‘payments'
```

These kind of configurations create snowflake servers that need constant maintenance. If you have fallen into this trap, try to understand how much time you will need to spend in the following scenarios:

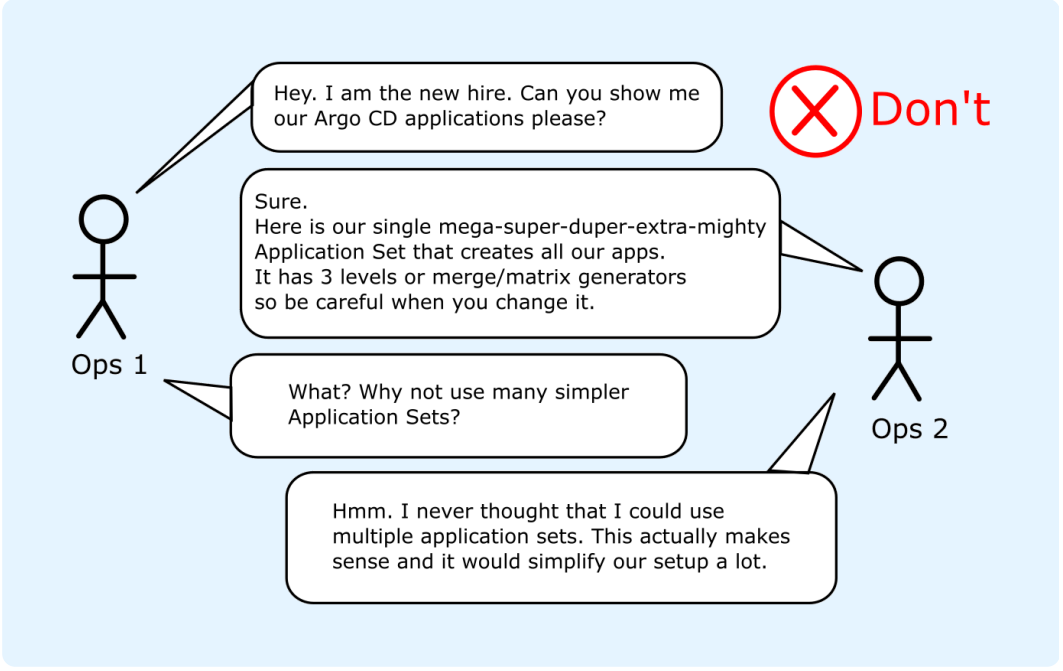
- Creating a brand new server
- Migrating an application from one server to another
- Applying a global setting to all your servers
- Making a different configuration change for a subset of your servers.

This kind of cluster management is even more problematic for developers, as simply understanding what applications are deployed where is not easy (already explained in anti-pattern 6).

We recommend creating a production-ready setup using cluster labels. We have described all the details in a **dedicated guide for the cluster generator and Argo CD application sets**[43].

22. Attempting to use a single application set for everything

A related anti-pattern is when teams discover application sets and, for some unknown reason, try to cram all their applications into a single application set. This often results in a complex mix of different generators that is hard to understand and hard to debug.



The recommendation is to have many different application sets in your Argo CD setup. Ideally, you should have different application sets per "type" of application. This type can be anything that makes sense to you. For example:

- An application set for all staging apps
- An application set for all AWS clusters
- An application set for all infra apps
- An application set for the billing team
- An application set for the payments team

You can slice and dice your applications in different dimensions. But in the end, you will have many application sets, and any time you make a change, you should instantly know *where* it should happen.

- A new requirement is that all AWS clusters need to get sealed secrets -> Change the AWS application set.
- A new requirement is that the billing team add a new microservice to their setup -> Change the "billing" application set.
- All new clusters must upgrade Prometheus -> Change the "common" application set.

There is no technical limitation on the number of application sets you can have on a single Argo CD installation. So, spend some time understanding your application sets accordingly.

As always, **our Argo CD application guide**[6] is the best starting point.

23. Using Pre-sync hooks for db migrations

Argo CD phases/waves[44] let you define the order of Resources synced within a single Argo CD application. The pre-sync phase, in particular, can be used to check deployment requirements or perform other checks that need to happen before the main sync phase.

We often see organizations attempting to use pre-sync hooks for database migrations. The assumption here is that the database schema must be updated immediately before the new application version. Unfortunately, most organizations use legacy DB migration tools, and almost always, they package a DB migration CLI tool in the pre-sync phase, **which is not the proper approach for Kubernetes applications**[45].

Using pre-sync hooks for database migrations has several issues. The most important one is that the DB CLI tool is just a black box for Argo CD. The CLI runs and never reports back to Argo CD what really happened with the database. This can leave the DB in an inconsistent state, causing the main sync phase to always fail.

The second problem is that Argo CD is based on continuous reconciliation, where an application might be synced for several reasons and in different time frames. Traditional database CLI tools weren't designed with this scenario in mind. They assume they run only once within a typical CI pipeline.

At this point, Argo CD users are looking for several hacks to force the pre-sync hook to run ONLY in the initial application deployment and not in any subsequent sync events, as this either slows down the deployment or breaks the database completely.

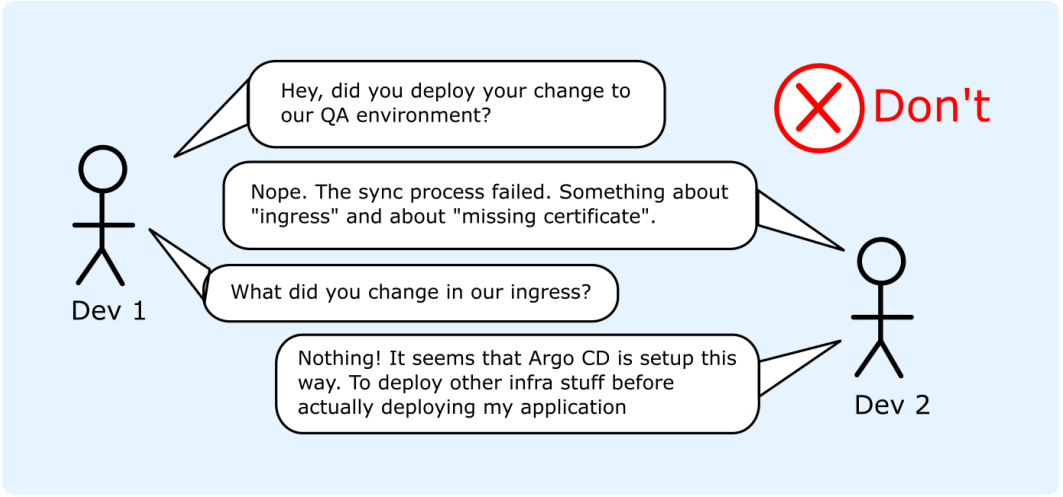
The correct approach is to use a Database migration operator built for Kubernetes. We **have written a full guide**[46] using the **AtlasGo DB operator**[47].

24. Mixing Infrastructure apps with developer workloads

We have seen in anti-pattern 7 several ways to group applications with GitOps (application sets, apps-of-apps, Helm umbrella charts).

Grouping methods should always be used to group the same kind of applications. They should group either infrastructure applications (core-dns, nginx, prometheus) or the applications developers create.

Never mix applications of different types, as this forces developers to deal with infrastructure errors.



When you create a new cluster and hand it over to developers, it should already have everything they need. The respective application sets should also have installed any infrastructure applications.

Mixing both infrastructure and developer applications might be easier for you (to control the deployment order), but it always results in a bad experience for developers.

If developers have access to the Argo CD UI and see a deployment error, they should know immediately that it is something they can fix themselves.

25. Misusing Argo CD finalizers

Argo CD finalizers[48] let you define what happens when an Argo CD application (or application set) is removed. You must understand how finalizers work and the impact of adding/removing a finalizer from a resource.

When a team doesn't understand finalizers, they often delete one or more Argo CD applications accidentally. In other cases, resources become "stuck" and are never recreated because of finalizer misconfigurations.

Finalizers can be useful when you want to migrate applications from one Argo CD instance to another.

We have written **a comprehensive guide about Argo CD finalizers**[49] and how to use them.

26. Not understanding resource tracking

This anti-pattern is related to the previous one. First of all it's crucial to understand how Argo CD tracks and "adopts" Kubernetes resources. You can have Kubernetes resources that aren't managed by Argo CD, or Argo CD resources that are "owned" by other Kubernetes controllers.

It is vital to know that the relationship between a Kubernetes resource and the Argo CD application that owns it is not always 1-1. You can have

1. Argo CD applications that no longer contain any Kubernetes resources
2. Kubernetes resources that are no longer owned by an Argo CD application

You can achieve the second scenario with finalizers. This pattern is handy when you want to migrate applications from one Argo CD instance to another without downtime.

The full process for this is:

1. Argo CD instance A owns all Kubernetes resources
2. You remove all finalizers for all applications (and application Sets)
3. You delete all Argo CD applications
4. The Kubernetes resources **are still running** just fine. There is no downtime
5. You apply the same Argo CD applications to Argo CD instance B
6. Argo CD instance B will adopt the same Kubernetes resources as before (with no downtime)

You can try this exact scenario of moving Argo CD applications between instances without downtime in our **Gitops Certification (level 3) course**[50].

You can use the same process to safely upgrade an existing Argo CD instance to a new version.

Anti-patterns: Extending and integrating Argo CD

In this last section, we will see how Argo CD fits into the existing Continuous Delivery ecosystem. We see organizations that enjoy using Argo CD and mistakenly assume it is a one-stop shop for the whole application delivery process. Argo CD is mainly a synchronization engine. It is not a observability dashboard, it is not a configuration management platform and it is *definitely not* a secrets management solution.

Argo CD is a victim of its own success. Organizations adopt it and like it so much that they try to abuse for tasks that are not fit for a synchronization engine. Argo CD comes with several extension points in the form of backend and frontend plugins. These plugin mechanisms are created for complementary functions that are closely related to the sync process. They are not generic extension points for turning Argo CD into something else.

Despite its strengths and advantages, GitOps remains a building block for application delivery. You need more than just a synchronization engine to deploy applications at scale.

27. Creating "active-active" installations of Argo CD

Teams sometimes try to create "active-active" installations of Argo CD with the following requirement:

1. The main Argo CD instance is controlling all applications and deployments
2. There is a secondary Argo CD instance also pointing to the same cluster
3. If the main Argo CD instance fails, the secondary instance jumps in
4. When the main Argo CD instance is restored, it re-adopts all applications

These teams are disappointed to learn that Argo CD doesn't support this and even **the centralized mode is for controlling other Kubernetes clusters**[42] and not other Argo CD instances.

This requirement doesn't really make sense for Argo CD, and teams that seek this "active-active" configuration haven't really understood resource tracking.

Argo CD only deploys applications. It doesn't control them. If Argo CD fails, new deployments will stop, but **existing applications will continue to work just fine**. And even if those fail for some reason, the Kubernetes cluster will reschedule or restart them (even if Argo CD is no longer operational).

The disaster recovery scenario for Argo CD is straightforward if your team has everything in Git (see anti-pattern 1). You can launch a second Argo CD cluster and point it to the same application manifests. Argo CD will then adopt the existing Kubernetes resources.

This is even possible if the central Argo CD instance has issues but still runs, as you can use finalizers (as explained in the previous section) to migrate applications to the second Argo CD instance without downtime.

Keeping a second Argo CD instance in "active-active" mode only wastes resources.

28. Recreating Argo Rollouts with Argo CD and duct tape

Bad deployments can always happen, regardless of whether you are using Argo CD. That's a fact for any software team. So, how do failed deployments work if you have adopted GitOps?

The simplest way to fix a failed deployment is to roll "forward". Make a new release or fix the Kubernetes manifests, and once you commit, Argo CD will deploy the latest changes and hopefully restore the application to a good state.

Argo CD also includes a "**rollback**[51]" command, which simply points the application back to a previous Git hash. This sounds great in theory but it has 2 major problems:

1. It works only if auto-sync is disabled (anti-pattern 10)
2. It breaks GitOps as your cluster doesn't represent what is in Git anymore

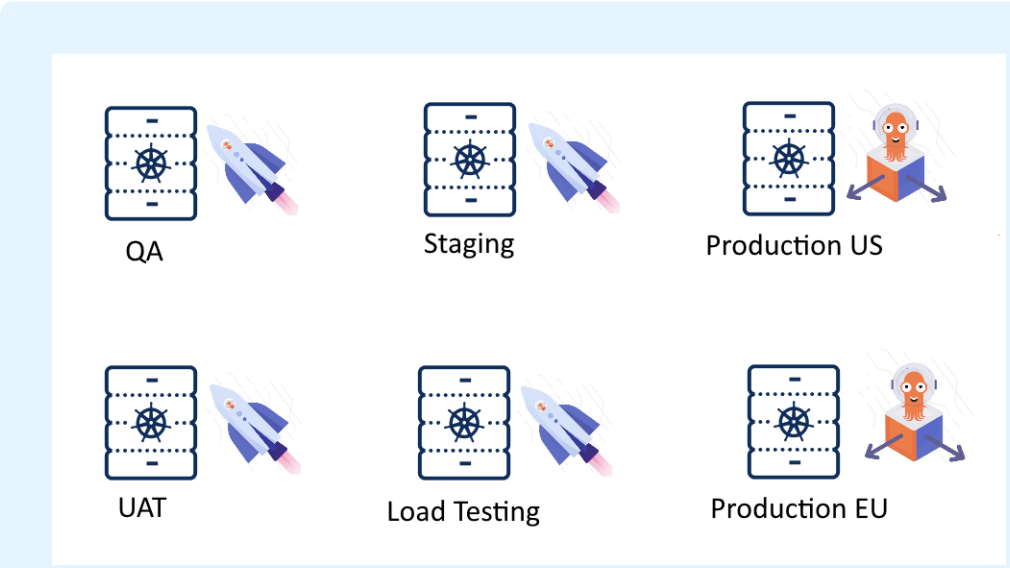
At this point, teams start creating custom solutions to overcome these limitations. The most common approaches we see are:

1. Trying to detect a failed deployment, and then switching off auto-sync on the fly while rolling back
2. Using notifications with external metric providers who will automatically try to revert a commit on their own so that Argo CD will sync as usual to the previous version

These custom solutions are always clunky and create more problems than they solve. You know that your team is falling into this trap if you hear people always asking the question, "How can I switch off auto-sync temporarily?"

The recommended solution is to use **Argo Rollouts**[52].

Argo Rollouts is a progressive Delivery controller designed for this exact scenario—automated rollbacks when things go wrong. It also comes with its own resource (**Analysis**[53]) that lets you **look at your metrics**[54] during a deployment and roll back without human intervention.



Argo Rollouts will handle production deployments, while non-production environments can still use plain Argo CD.

29. Recreating Argo Workflows with Argo CD, sync-waves and duct tape

The **sync wave feature**[55] of Argo CD allows you to execute tasks before and after the main sync phase. These tasks should ideally be idempotent and quick to finish. Some examples are:

- Sending a notification to another system
- Performing a quick smoke test
- Verifying that a dependency exists

We see teams that misuse the sync waves in Argo CD with long-running tasks that are part of a bigger process with strict requirements such as

- Automatic retries
- If/else control flows
- Dependency graphs and fan-in/fan-out configuration
- Artifact storage and retrieval

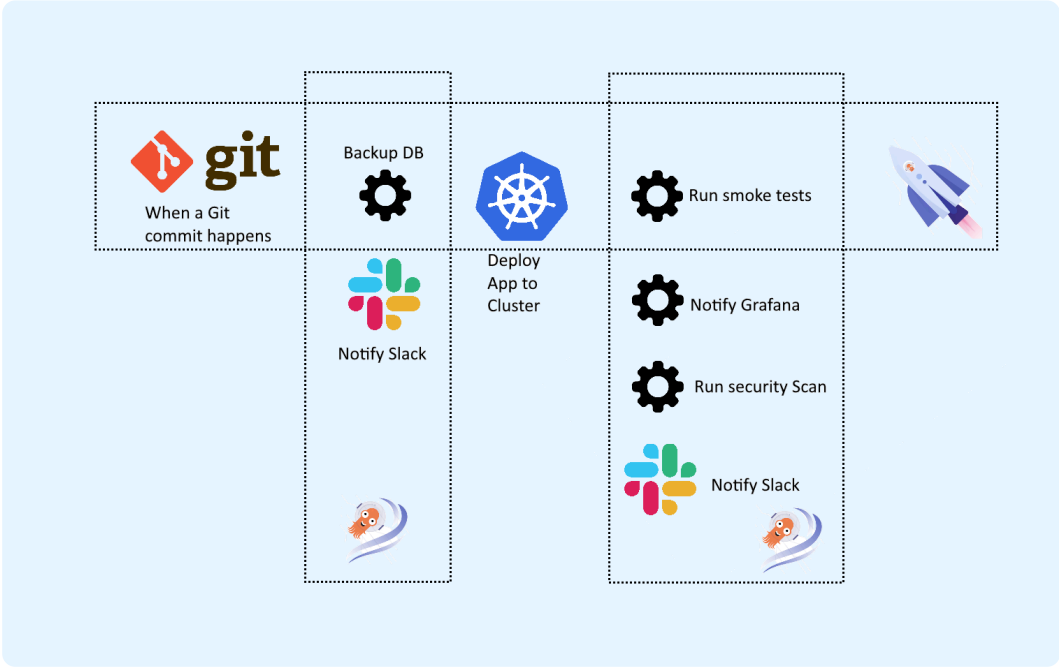
Sync waves were **never** designed for this kind of requirement. If you try to do this, you will soon resort to custom scripts that nobody wants to maintain. Adopting Argo CD for deployments and then trying to incorporate custom scripts in the sync process is a huge step backward.

If you have this kind of process, you should use **Argo Workflows**[56] which handles these requirements.

Argo Workflows are Kubernetes-native workflows that offer all these features out of the box.

Therefore, the whole sync process should be:

1. Run an Argo Workflow before the sync process
2. Perform the main Sync phase
3. Run another Argo Workflow after the sync process.



Argo Workflows will then handle all the heavy lifting using declarative Kubernetes resources instead of custom scripts.

30. Abusing Argo CD as a full SDLC platform

Despite its powerful features and developer-friendly UI, Argo CD is very simple at its core. It is a powerful sync engine that continuously monitors what you have in Git and applies changes to your cluster. All its features are centered around this use case.

However, deploying an application is only part of the software development life cycle (SDLC). Several other requirements must be met before (the CI process) and after (observability) the main deployment.

We have seen several teams try to expand Argo CD's scope and turn it into something it never was. Most importantly, Argo CD has no visibility in your Continuous Integration (CI) process. Argo CD doesn't know:

1. What are the new features in the container it deploys
2. Who made the container build
3. If the application has passed your unit and integration tests
4. If your new container has passed your security scans
5. Who approved the pull request

In fact, Argo CD doesn't even know that it deploys a new container that includes commits from a source code repo. All it knows is diffing and applying Kubernetes manifests without any insights into the business features behind them.

Attempting to integrate this information into Argo CD, whether through custom plugins or custom YAML segments, is always a clunky process. We understand the need for a unified interface. Developer teams love the Argo CD UI and think they can use it as a central dashboard for everything.

That is **not** the role of Argo CD. You can create a developer portal that uses Argo CD behind the scenes, but Argo CD is not a developer portal itself.

If you want to use a central platform for all your Argo CD instances that also handles application promotion, **check out Octopus Deploy**[37].

Conclusion

We hope that this book has been a comprehensive guide for your GitOps journey. You should now be able to identify bad practices and opt to use the better alternatives. Argo CD is a great tool, but it has a series of knobs and switches that can lead to undesirable results.

Some features can be abused in several ways because there is no good documentation on their history, intended use, or what to avoid.

Using this guide, you can start your Argo CD journey in the best possible way, as you now know what to avoid before investing a significant amount of effort into your Application manifests.

Here is a summary of all the anti-patterns and our recommendations:

Before adopting Argo CD

1. Not understanding the declarative setup of Argo CD → Store Application CRDs in Git.

2. Creating dynamic Argo CD applications → Use Git as the single source of truth for application configuration.

3. Using Argo CD parameters → Avoid using the parameters feature as it goes against GitOps principles.

4. Adopting Argo CD without understanding Helm → Understand how Helm works independently before adopting Argo CD.

5. Adopting Argo CD without understanding Kustomize → Ensure your Kustomize files work on their own before integrating with Argo CD.

6. Assuming that developers need to know about Argo CD → Design your Argo CD applications so developers can recreate configurations without Argo CD knowledge.

Foundations and first steps

7. Grouping applications at the wrong abstraction level → Use Application Sets or app-of-apps pattern for proper application grouping.

8. Abusing the multi-source feature of Argo CD → Use multi-source as a last resort and only for edge case scenarios.

9. Not splitting the different Git repositories → Separate source code, Kubernetes manifests, and Argo CD application manifests into different Git repositories.

10. Turning off auto-sync and self-heal → Keep auto-sync/self-heal enabled for all systems, including production.

11. Abusing the targetRevision field → Always use HEAD in the targetRevision field.

12. Misunderstanding immutability for container/git tags and Helm charts → Actively set up the ecosystem (Git, Helm repos, artifact managers) to work with immutable data.

13. Giving too much power (or no power at all) to developers → Balance flexibility and security with Argo CD RBAC, and enable local testing without Argo CD.

14. Referencing dynamic information from Argo CD/ Kubernetes manifests → Store all values used in manifests statically in Git.

Configuration management

15. Writing applications instead of Application Sets → Use Application Sets to automate the creation of Application files.

16. Using Helm to package Applications instead of Application Sets → Learn how Application Sets work and their features.

17. Hardcoding Helm data inside Argo CD applications → Store all Helm information in Helm values, separate from Argo CD manifests.

18. Hardcoding Kustomize data inside Argo CD applications → Store Kustomize information only in Kustomize overlays separate from Argo CD manifests

19. Attempting to version and promote Applications/Application Sets → Promote values or overlays, not Application manifests themselves.

20. Not understanding what changes are applied to a cluster → Use tools or CI systems to preview and diff rendered Kubernetes manifests.

Cluster management

21. Using ad-hoc clusters instead of cluster labels → Use cluster labels and Application Sets to distribute applications to different clusters.

22. Attempting to use a single application set for everything → Have many different Application Sets, each with a different purpose/scope.

23. Using Pre-sync hooks for db migrations → Use a Database migration operator explicitly built for Kubernetes.

24. Mixing Infrastructure apps with developer workloads → Separate infrastructure applications from developer workloads.

25. Misusing Argo CD finalizers → Understand how finalizers work and use them correctly for application deletion and migration.

26. Not understanding resource tracking → Understand how Argo CD tracks and adopts Kubernetes resources.

Extending and integrating Argo CD

27. Creating "active-active" installations of Argo CD → Avoid active-active setups, rely on Git and resource tracking for disaster recovery.

28. Recreating Argo Rollouts with Argo CD and duct tape → Use Argo Rollouts for progressive delivery and automated rollbacks.

29. Recreating Argo Workflows with Argo CD, sync-waves and duct tape → Use Argo Workflows for long-running tasks and complex process orchestration.

30. Abusing Argo CD as a full SDLC platform → Use a different system as a developer portal or promotion orchestrator.

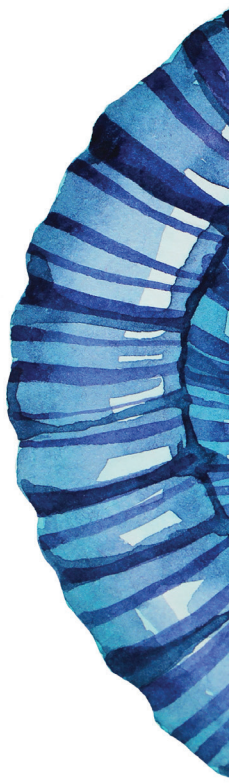
If you want to dive deep into any of these topics, **Octopus Deploy also offers Argo CD support**[57] and advice. Our team members are not just Argo CD experts; they are core members of the Argo CD team. That means you can ask questions from the people who **develop and release Argo CD**[58]!

References

1. opengitops.dev
2. argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/
3. argo-cd.readthedocs.io/en/stable/user-guide/projects/
4. argocd-autopilot.readthedocs.io/en/stable/
5. github.com/kostis-codefresh/many-appsets-demo/blob/main/root-argocd-app.yml
6. codefresh.io/blog/how-to-structure-your-argo-cd-repositories-using-application-sets/
7. man7.org/linux/man-pages/man1/envsubst.1.html
8. argo-cd.readthedocs.io/en/stable/user-guide/parameters/
9. helm.sh/docs/helm/helm_template/
10. argo-cd.readthedocs.io/en/latest/user-guide/commands/argocd_app_rollback/
11. helm.sh/docs/chart_best_practices/templates/
12. helm.sh/docs/chart_template_guide/subcharts_and_globals/
13. codefresh.io/blog/helm-values-argocd/
14. kubectl.docs.kubernetes.io/guides/config_management/components/
15. kubectl.docs.kubernetes.io/references/kustomize/kustomization/configmapgenerator/
16. kubectl.docs.kubernetes.io/references/kustomize/kustomization/replacements/
17. github.com/kostis-codefresh/gitops-environment-promotion
18. codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/
19. helm.sh/docs/howto/charts_tips_and_tricks/#complex-charts-with-many-dependencies

20. argo-cd.readthedocs.io/en/latest/operator-manual/cluster-bootstrapping/#app-of-apps-pattern
21. argo-cd.readthedocs.io/en/latest/user-guide/application-set/
22. argo-cd.readthedocs.io/en/latest/user-guide/multiple_sources/
23. codefresh.io/blog/argo-cd-best-practices/
24. www.cncf.io/blog/2020/12/17/solving-configuration-drift-using-gitops-with-argo-cd/
25. argo-cd.readthedocs.io/en/stable/user-guide/auto_sync/#automatic-self-healing
26. argo-cd.readthedocs.io/en/latest/user-guide/tracking_strategies/
27. codefresh.io/blog/stop-using-branches-deploying-different-gitops-environments/
28. codefresh.io/blog/creating-temporary-preview-environments-based-pull-requests-argo-cd-codefresh/
29. codefresh.io/blog/argocd-application-target-revision-field/
30. codefresh.io/blog/multi-tenant-argocd-with-application-projects/
31. codefresh.io/blog/why-environments-beat-clusters-for-dev-experience/
32. github.com/argoproj/argo-cd/issues/5202
33. codefresh.io/blog/handle-secrets-like-pro-using-gitops/
34. codefresh.io/blog/gitops-secrets-with-argo-cd-hashicorp-vault-and-the-external-secret-operator/
35. argo-cd.readthedocs.io/en/latest/operator-manual/secret-management/
36. argo-cd.readthedocs.io/en/latest/user-guide/commands/argocd_appset_generate/
37. octopus.com/use-case/argo-cd-in-octopus
38. codefresh.io/blog/argo-cd-preview-diff/
39. github.com/dag-andersen/argocd-diff-preview
40. argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/Generators-Cluster/
41. argo-cd.readthedocs.io/en/latest/operator-manual/applicationset/Generators-Pull-Request/

42. codefresh.io/learn/argo-cd/a-comprehensive-overview-of-argo-cd-architectures-2025/
43. codefresh.io/blog/argocd-clusters-labels-with-apps/
44. argo-cd.readthedocs.io/en/latest/user-guide/sync-waves/
45. codefresh.io/blog/database-migrations-in-the-era-of-kubernetes-microservices/
46. codefresh.io/blog/using-gitops-for-databases/
47. atlasgo.io/
48. argo-cd.readthedocs.io/en/stable/user-guide/app_deletion/
49. codefresh.io/blog/argocd-application-deletion-finalizers/
50. learning.octopus.com/
51. argo-cd.readthedocs.io/en/stable/user-guide/commands/argocd_app_rollback/
52. argoproj.github.io/rollouts/
53. argo-rollouts.readthedocs.io/en/stable/features/analysis/
54. argo-rollouts.readthedocs.io/en/stable/best-practices/#prepare-your-metrics
55. argo-cd.readthedocs.io/en/stable/user-guide/sync-waves/
56. argoproj.github.io/workflows/
57. octopus.com/support/enterprise-argo-support
58. blog.argoproj.io/announcing-argo-cd-v3-small-but-mighty-df05c0b39ad6



Octopus Deploy
Level 4, 199 Grey St
South Brisbane, QLD 4101, Australia

Email: sales@octopus.com

Phone: +1 512-823-0256

octopus.com