Octopus Deploy

Argo CD Best Practices

Learn how to create application dependencies, preview your deployments, and create ephemeral environments with Argo CD and GitOps.

Kostis Kapelonis

Contents

Introduction

Getting started with Argo CD sounds very easy. You install Argo CD in a cluster, point it to your Kubernetes manifests in Git, and sit back while Argo CD deploys all your applications. In reality, things are much more complex, and after the initial installation, there are several open issues and challenges with day-2 operations. In this whitepaper, we explore the following topics:

1. The different ways to perform the initial installation of Argo CD
2. How to organize application dependencies, especially if your company has micro-services
3. How to preview your Argo CD changes when you create pull requests
4. How to use Argo CD for temporary/ephemeral environments

Installation and upgrades

Installing an Argo CD instance for testing purposes is a straightforward process, as the project is provided in **a single installation manifest**[1], so if you want to evaluate Argo CD on a local cluster quickly, no additional effort is required.

Chances are that one of your first Argo CD installations happened with `kubectl` as explained in the **getting started guide**[2]. While this form of installation is great for quick experimentation and for trying Argo CD, we recommend other installation methods for production deployments.

Manual approach

The most obvious installation method is using the official manifests or the respective **Helm chart**[3]. Note that the Helm chart for Argo CD is not official and sometimes lags behind regular Argo CD releases.

While the initial installation of Argo CD using the manifests is quick and straightforward, it suffers from several shortcomings:

- You must apply configuration changes like SSO and notifications manually without any option for rollbacks or auditing.
- Lack of disaster recovery capabilities if your Argo CD instance becomes unavailable.
- Extra effort is needed for modifications on the base install (e.g., for Argo CD plugins).
- It becomes cumbersome to manage multiple Argo CD instances in a manual way.

However, the biggest challenge is safely upgrading Argo CD. New Argo CD versions come with their **own notes and incompatibilities**[4], and trying to manually upgrade your instance without any backup plan is a recipe for disaster.

You should only use manual installation via manifests for quick prototypes and demo installations.

Using Infrastructure as Code (IaC)

Argo CD is a standard Kubernetes application like any other Kubernetes application. If you already have a well-defined workflow for deploying and managing applications on Kubernetes, you can reuse it for the initial Argo CD installation.

A very popular method is to use IaC tools such as **Terraform**[5], **Pulumi**[6], and **Crossplane**[7]. A common bootstrap pattern is:

1. Create a "hub" Kubernetes cluster with Terraform
2. Use the Terraform **K8s**[8] or **Helm providers**[9] to install Argo CD on the "hub" cluster
3. Create more Kubernetes clusters
4. Add the Kubernetes clusters to the main cluster (hub-and-spoke pattern) using **the Argo CD Terraform provider**[10]
5. Point Argo CD to your **Application Sets**[11] in order to start your deployments

You can use a similar process with other IaC tools like Pulumi and Crossplane.

It is important to clearly separate the two worlds. The IAC tool should create the cluster and stop after the installation of Argo CD. Argo CD is then responsible for all Kubernetes workloads.

Using Argo CD to manage Argo CD

Managing your own Argo CD instance with an IaC tool is popular, but there is another approach since Argo CD is a Kubernetes application. You can **manage Argo CD with itself**[12].

This use case is valid, and many organizations use self-managed Argo CD. The advantages are:

- Using GitOps to handle both applications and the Argo CD installation
- Full audit trail via Git
- Easy rollbacks
- Automatic drift detection for any manual changes
- Complete changelog of all configuration changes (e.g., notifications SSO)
- Easy disaster recovery

Using Argo CD to manage itself is a great way to manage a production instance of Argo CD; however, depending on the size of your organization, this approach will still suffer from some key challenges:

1. You still have to perform manual upgrades and make sure that each new version of Argo CD "sits" cleanly on top of the previous one.
2. Managing many Argo CD installations and keeping them all in sync (pun intended) is still a big challenge.

Using Argo CD Autopilot

The use case for using Argo CD to manage Argo CD is very popular as a concept, but there are no best practices for getting started and bootstrapping the whole environment.

We use the same approach internally, and we fully **open-sourced our solution**[13].

Argo CD autopilot provides a CLI for installing and managing Argo CD that:

1. Connects to your Git provider
2. Bootstraps Git repositories for handling both applications and itself
3. Set up Applications and ApplicationSet for auto-upgrading itself and other managed apps
4. Provides a best practice Git repo structure for both internal and external applications
5. Comes with a CLI that lets you to manage and maintain the installation
6. Introduces the concepts of deployment environments/projects

Argo CD Autopilot is under active development. You are welcome to participate in **Github**[14] as well as the `#argo-cd-autopilot` channel in the **CNCF slack**[15].

Using a control plane

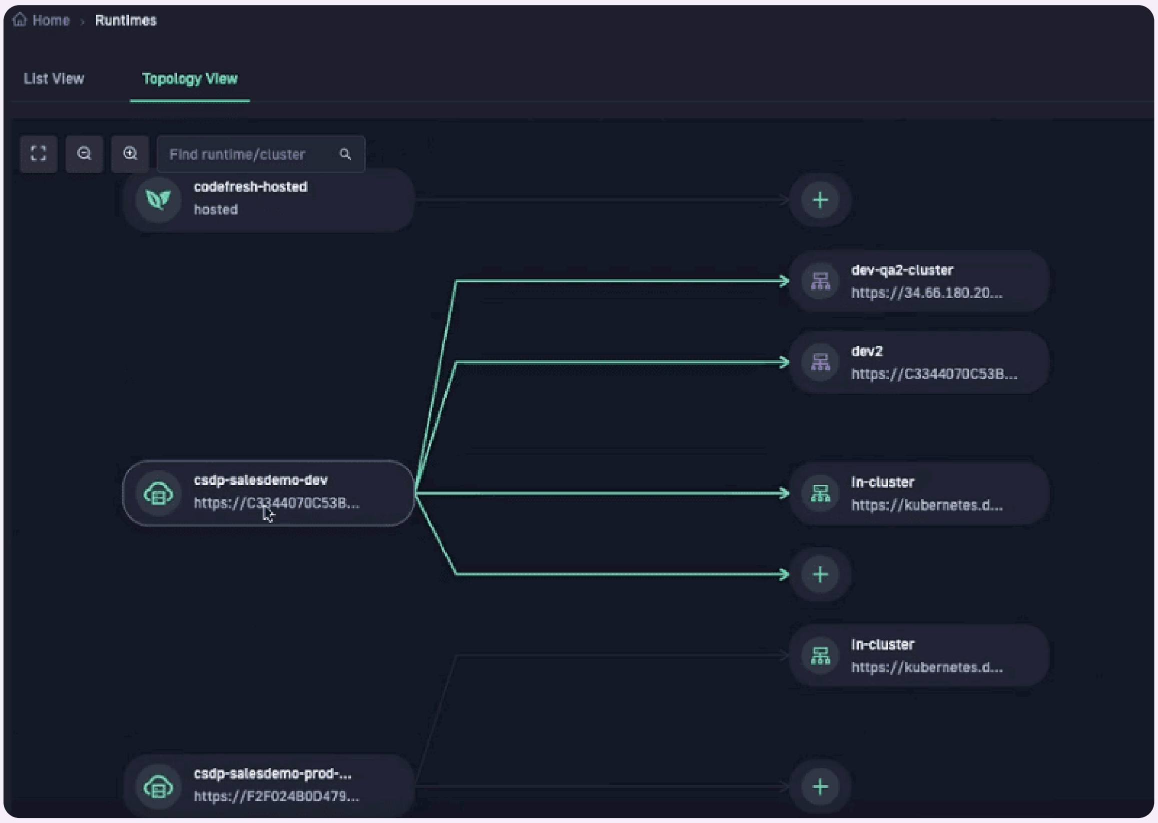
Handling one or two Argo CD instances is pretty straightforward if you choose any of the above installation methods. However, some organizations have many Argo CD instances that need to be kept in sync or rolled out gradually as new versions are released.

Argo CD can natively support a management instance that manages multiple deployment clusters. Theoretically, you could have a single Argo CD instance for all your environments. We have already talked about this pattern in our article about **scaling Argo**[16]. In the end, having a single instance is a single point of failure and comes with its own issues for security and redundancy.

On the other hand, having an Argo CD instance for each deployment cluster is also excessive. It can lead to a cumbersome setup where maintaining Argo CD instances becomes tedious and unmanageable, especially across virtual private clouds and firewalls.

Ideally, you would like a single management interface that can handle all possible combinations (Argo CD management cluster and deployment clusters), allowing you to craft your perfect topology.

This management interface exists in the form of the Codefresh GitOps control plane!



Argo CD multi cluster management

The **Codefresh platform**[17] gives you a unified interface for handling all Argo CD instances, no matter where you locate them. All possible configurations are supported:

- Argo CD management clusters
- Argo CD deployment clusters
- Hosted Argo CD installations
- Deployment clusters managed by the hosted instance
- Argo CD instances that deploy on the same cluster where you install them
- Argo CD instances that are deployed behind a firewall or in an on-premises environment

Via the control plane interface, you can:

1. Connect Argo CD instances
2. Connect target deployment clusters
3. See the status of each Argo CD instance and each connected cluster
4. Upgrade Argo CD instances to a new version in a controlled manner
5. Keep track of versions/security alerts, and easily upgrade

The unified control plane is the perfect tool for organizations that need a management interface for all their Argo CD instances without the hassle of manual upgrades.

Scaling usage

You should now know the possible installation options for Argo CD. It is important to understand that you can mix-and-match installation methods or change them completely as your company grows.

We have seen several customers start with the Terraform installation and migrate to the Codefresh Cloud platform as the number of clusters increases. Remember that it is possible to migrate Argo CD applications to a different instance without downtime, so don't assume you are locked in using any one installation method.

Application dependencies

With the installation of Argo CD out of the way, we can focus on application management. We will see a very common use case - Application ordering.

If you are using Argo CD, you may already know how the application Custom Resource Definition (CRD) object helps you logically group your Kubernetes Manifests. The application object is the atomic unit of work in Argo CD, and you should think of all your Kubernetes objects in an application as a single entity.

Applications are also autonomous, meaning that by design, one application doesn't know about the status or health of another application. This autonomy could pose a challenge in organizations implementing a microservices architecture, with each component in its own application Custom Resource (CR). Some examples include:

1. Database -> Back end
2. Queue -> Queue workers -> Back end
3. Kyverno/Opa -> Apps that need to be limited
4. Database -> Back end -> Frontend

Since there is no way **currently**[18] to set up Application dependencies natively, is there a way to do it with what's available?

The answer is: Yes, by combining App-of-Apps and Syncwaves.

App-of-Apps pattern

The **App-of-Apps**[19] pattern was a design that came from the community of Argo CD users. The App-of-Apps design is an Argo CD Application comprising other Argo CD Applications. Initially, the use case was for bootstrapping, as administrators needed a way to deploy Argo CD Applications using Argo CD itself. The natural fit was to create an application of other Argo CD Applications, since an application is just another Kubernetes object.

Given the above example, we'd have the following Argo CD Applications:

- Cert-Manager
- Back-end Application
- Caching System
- Kyverno
- Frontend Application
- Ingress

Instead of deploying 6 individual Argo CD Applications, you can deploy one Argo CD Application that deploys the other 6 applications for you.

The App-of-Apps pattern also provides a way to bootstrap a cluster with your applications through a convenient "entry point" and creates a "watcher" Application. Another benefit is that you can use other Argo CD paradigms within this App-of-Apps pattern, such as SyncWaves.

SyncWaves

Syncwaves and Synchooks[20] are a way to order how Argo CD applies individual manifests within an Argo CD Application. The order is determined by annotating the object with the order you'd like to apply the manifest, starting with the lowest number first (negatives are allowed). For example, if you annotate your deployment as "0", and your service as "1", Argo CD will apply the deployment *first*, wait for it to report a healthy status, and then apply the service.

You may already have used Syncwaves and App-of-Apps separately. But what happens if they are used together? Can we order individual Argo CD Applications instead of just Kubernetes resources with Syncwaves?

Prerequisites

You need to set up a few things before integrating SyncWaves with your App-of-Apps deployment and creating Argo CD Application dependencies.

Argo CD application health

Argo CD has built-in health checks for several standard Kubernetes objects. It then bubbles these checks up to the overall Application health status as one unit. For example, Argo CD marks an Application with a Service and a Deployment as "healthy" only if both objects are considered healthy.

Some of the built-in health checks include (but are not limited to): Deployment, ReplicaSet, StatefulSet, DaemonSet, Service, Ingress, and PersistentVolumeClaim. You can also add custom health checks. You can read more about it in the **official documentation**[21].

Also described in the **official documentation**[22], you need to tell Argo CD how to check for the Application Custom Resource overall health. You do this by modifying the *argocd-cm* ConfigMap and adding a resource customization. For example:

```
apiVersion: v1
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  resource.customizations: |
    argoproj.io/Application:
      health.lua: |
        hs = {}
        hs.status = "Progressing"
        hs.message = ""
        if obj.status ~= nil then
          if obj.status.health ~= nil then
            hs.status = obj.status.health.status
            if obj.status.health.message ~= nil then
              hs.message = obj.status.health.message
            end
          end
        end
        return hs
```

This ensures the application controller reports the health of the Application CR correctly. Starting in Argo CD version 1.8, you must use this setting (see issue **3781**[23] for more details).

Readiness and liveness probes

Another part of getting Application dependencies up and running with App-of-Apps is to make sure your deployments/statefulsets/daemonsets have the proper **readiness/liveness probes**[24] set up. This is important because Argo CD will look at the object's health and use that to determine if the application is healthy. This can cause issues if you don't have proper readiness/liveness probes.

For example, a Deployment without readiness/liveness probes will be marked healthy as soon as it is applied and the container is running. This will cause the Application to be marked as healthy, when, in fact, your application may take some time to come up. To get your Application dependency working optimally, you should configure liveness and readiness probes. Here is a snippet of a Deployment manifest that has these set for a web application.

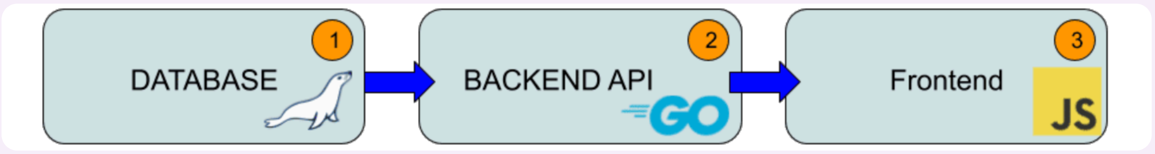
```
livenessProbe:
  httpGet:
    path: /
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 3
readinessProbe:
  httpGet:
    path: /
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 3
```

The above snippet shows that the container will be marked as "alive" when it returns the expected HTTP response when probing that port and path. Similarly, the container will be marked "ready" when the probe returns the expected HTTP response.

App dependency walkthrough

I will be using an example using a repo called **application-dependency,**[25] which is a 3 tiered application made up of a **frontend app**[26], **back-end api**[27], and a **database**[28]. These three apps are deployed, individually, using an Argo CD Application for each of them.

I want them to deploy in a specific order, and the logical way is to have the database come up first, then the back-end app, then the frontend. Here is a simple diagram:



Deployment order

To achieve this, I need to:

1. Create an individual Argo CD Application for each tier (**already done here**[29])
2. Create a parent Argo CD Application that deploys them (**already done here**[30])
3. Update Argo CD with the health check for the applications
4. Make sure I have probes setup for my objects
5. Annotate my applications with the right syncwave

To add the health check for Argo CD Applications, **I created a patch file**[31]; it should look like this:

```
$ cat patch-argocd-cm.yaml
data:
  resource.customizations.health.argoproj.io_Application: |
    hs = {}
    hs.status = "Progressing"
    hs.message = ""
    if obj.status ~= nil then
      if obj.status.health ~= nil then
        hs.status = obj.status.health.status
        if obj.status.health.message ~= nil then
          hs.message = obj.status.health.message
        end
      end
    end
    return hs
```

Then update the argocd-cm ConfigMap:

```
kubectl patch cm/argocd-cm --type=merge -n argocd \
--patch-file patch-argocd-cm.yaml
```

NOTE : This is just an example for this paper. Ideally you would use GitOps to manage this Argo CD ConfigMap. Also this can be done automatically with **ArgoCD Autopilot**[13].

You can verify with the following command:

```
kubectl get cm argocd-cm -n argocd -o yaml
```

Now that the argocd-cm ConfigMap is updated, you can annotate the Argo CD Application definition with the syncwave number. First, I annotate the Database application with "1", as I want it to get deployed first. You can **see the full manifest for the database in my repo**[28], here is a snippet of the YAML:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "1"
  name: my-db
  namespace: argocd
```

Next, I annotate my back-end API Argo CD Application with "2", since I want it to come up only after the database application finishes. It's a **similar manifest**[27], so here is a snippet:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "2"
  name: my-api
  namespace: argocd
```

Finally, I have my **frontend application manifest**[26], which has a syncwave annotation of "3":

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "3"
  name: my-frontend
  namespace: argocd
```

Before I go on to the App-of-App setting, I want to point out that all 3 of these applications have readiness/liveness probes configured. For example, the **Frontend Deployment**[32] and the **API Deployment**[33] have them set.

Now that we have the Argo CD Application health check set up, annotated the applications in the order I want them in, and made sure we have readiness/liveness probes in our apps, we can now take a look at the **"Parent" Argo CD Application**[30] in this App-of-Apps configuration.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: all-apps-in-order
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  source:
    path: apps
    repoURL: 'https://github.com/kostis-codefresh/application-depend
    targetRevision: main
  destination:
    namespace: argocd
    server: 'https://kubernetes.default.svc'
  project: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    retry:
      limit: 5
      backoff:
        duration: 5s
        maxDuration: 3m0s
        factor: 2
    syncOptions:
      - CreateNamespace=true
```

There is nothing inherently special about this application (it just looks like a regular application), and that's because there isn't! The thing to note is that **apps is the path in my repo**[29], where the other Argo CD Applications are.

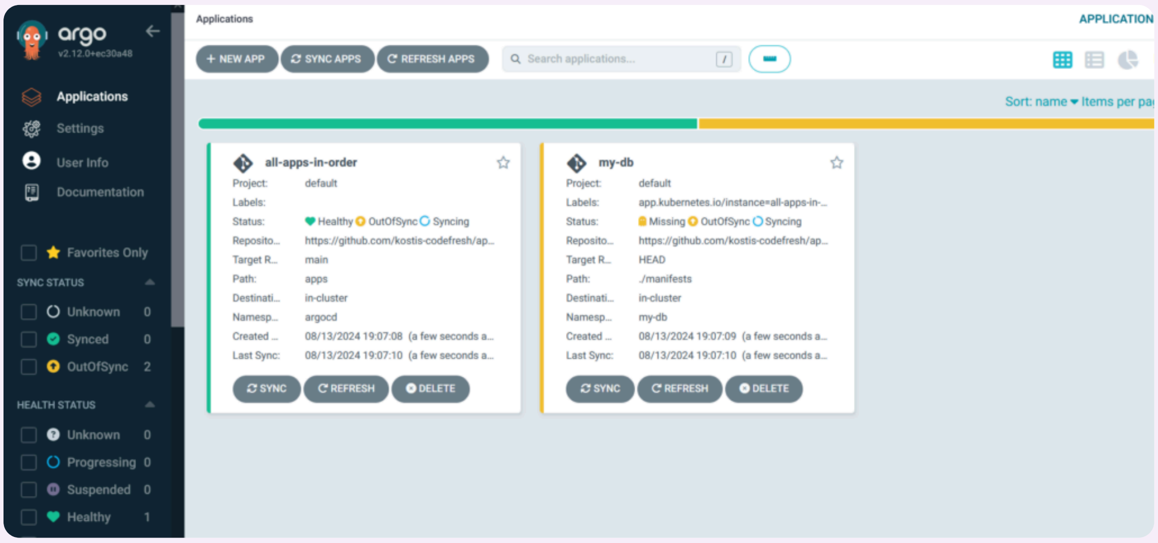
So far we have:

1. Created the Argo CD Applications for my apps
2. Created the Parent Argo CD Application that will deploy those applications
3. Added the proper Argo CD Application health check to Argo CD
4. Made sure my Deployments had the right readiness/liveness probes
5. Annotated my Argo CD Applications with the Syncwave I want

Now, let's see this in action! With this configuration in place, I can apply the "parent" Argo CD Application.

```
kubectl apply -f all-apps.yml
```

Here you'll see two Applications in the Argo CD UI, the "parent" Application and the first Application in the syncwave, in my case: the database Application.



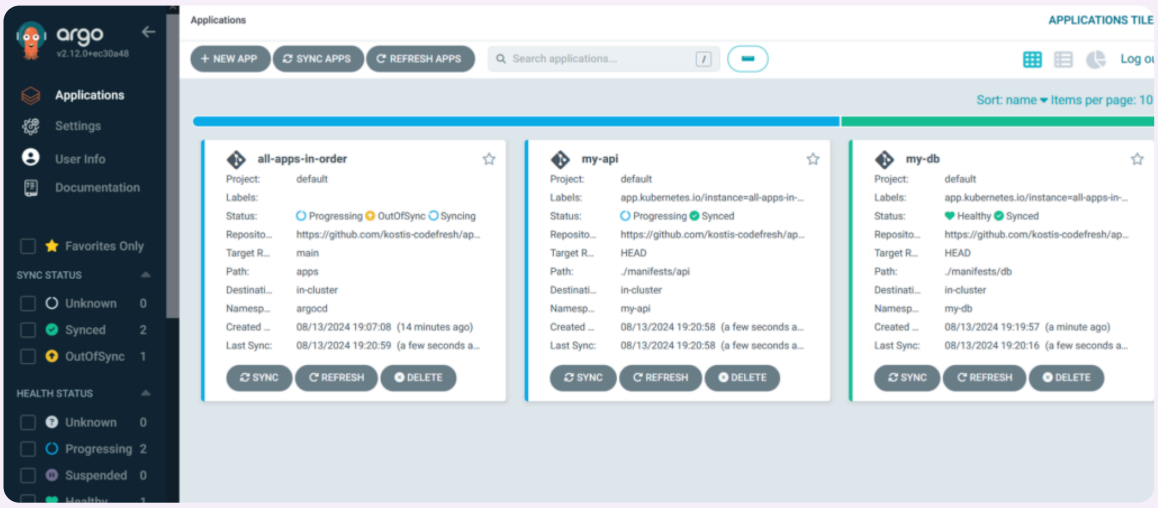
all-apps-in-order

Project:	default
Labels:	
Status:	Healthy OutOfSync Syncing
Reposito...	https://github.com/kostis-codefresh/ap...
Target R...	main
Path:	apps
Destinati...	in-cluster
Namesp...	argocd
Created ...	08/13/2024 19:07:08 (a few seconds a...
Last Sync:	08/13/2024 19:07:10 (a few seconds a...

my-db

Project:	default
Labels:	app.kubernetes.io/instance=all-apps-in-...
Status:	Missing OutOfSync Syncing
Reposito...	https://github.com/kostis-codefresh/ap...
Target R...	HEAD
Path:	./manifests
Destinati...	in-cluster
Namesp...	my-db
Created ...	08/13/2024 19:07:09 (a few seconds a...
Last Sync:	08/13/2024 19:07:10 (a few seconds a...

After that is complete, the next application in the syncwave starts which is the API application.

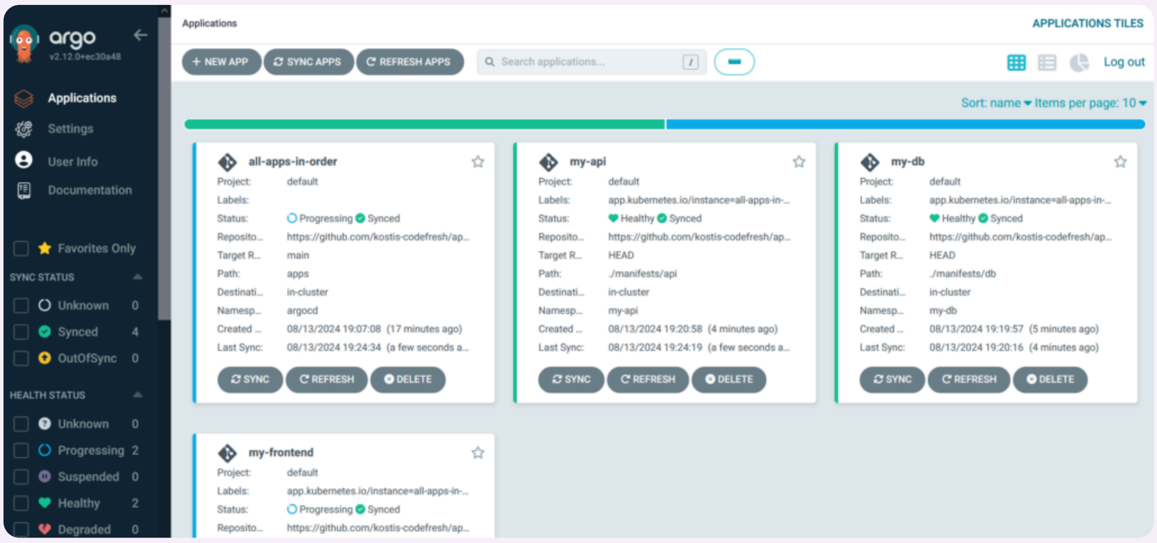


all-apps-in-order
Project: default
Labels:
Status: Progressing OutOfSync Syncing
Reposito... https://github.com/kostis-codefresh/ap...
Target R... main
Path: apps
Destinati... in-cluster
Namesp... argocd
Created ... 08/13/2024 19:07:08 (14 minutes ago)
Last Sync: 08/13/2024 19:20:59 (a few seconds a...

my-api
Project: default
Labels: app.kubernetes.io/instance=all-apps-in-...
Status: Progressing Synced
Reposito... https://github.com/kostis-codefresh/ap...
Target R... HEAD
Path: ./manifests/api
Destinati... in-cluster
Namesp... my-api
Created ... 08/13/2024 19:20:58 (a few seconds a...
Last Sync: 08/13/2024 19:20:58 (a few seconds a...

my-db
Project: default
Labels: app.kubernetes.io/instance=all-apps-in-...
Status: Healthy Synced
Reposito... https://github.com/kostis-codefresh/ap...
Target R... HEAD
Path: ./manifests/db
Destinati... in-cluster
Namesp... my-db
Created ... 08/13/2024 19:19:57 (a minute ago)
Last Sync: 08/13/2024 19:20:16 (a few seconds a...

Finally, the last application in the wave will start to deploy, which is the frontend application.



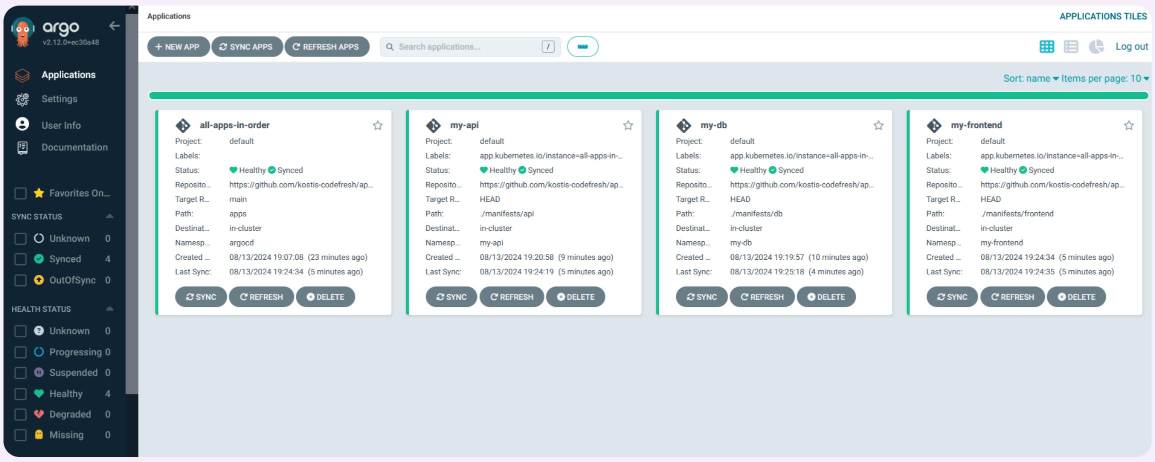
all-apps-in-order
Project: default
Labels:
Status: Progressing Synced
Reposito... https://github.com/kostis-codefresh/ap...
Target R... main
Path: apps
Destinati... in-cluster
Namesp... argocd
Created ... 08/13/2024 19:07:08 (17 minutes ago)
Last Sync: 08/13/2024 19:24:34 (a few seconds a...

my-api
Project: default
Labels: app.kubernetes.io/instance=all-apps-in-...
Status: Healthy Synced
Reposito... https://github.com/kostis-codefresh/ap...
Target R... HEAD
Path: ./manifests/api
Destinati... in-cluster
Namesp... my-api
Created ... 08/13/2024 19:20:58 (4 minutes ago)
Last Sync: 08/13/2024 19:24:19 (a few seconds a...

my-db
Project: default
Labels: app.kubernetes.io/instance=all-apps-in-...
Status: Healthy Synced
Reposito... https://github.com/kostis-codefresh/ap...
Target R... HEAD
Path: ./manifests/db
Destinati... in-cluster
Namesp... my-db
Created ... 08/13/2024 19:19:57 (5 minutes ago)
Last Sync: 08/13/2024 19:20:16 (4 minutes ago)

my-frontend
Project: default
Labels: app.kubernetes.io/instance=all-apps-in-...
Status: Progressing Synced
Reposito... https://github.com/kostis-codefresh/ap...

The App-of-Apps is now complete! Once all 3 applications are synced and healthy, the "Parent" Application will report as synced and healthy.



	all-apps-in-order	my-api	my-db	my-frontend
Project:	default	default	default	default
Labels:		app.kubernetes.io/instance=all-apps-in-...	app.kubernetes.io/instance=all-apps-in-...	app.kubernetes.io/instance=all-apps-in-...
Status:	Healthy Synced	Healthy Synced	Healthy Synced	Healthy Synced
Reposito...	https://github.com/kostis-codefresh/ap...	https://github.com/kostis-codefresh/ap...	https://github.com/kostis-codefresh/ap...	https://github.com/kostis-codefresh/ap...
Target R...	main	HEAD	HEAD	HEAD
Path:	apps	./manifests/api	./manifests/db	./manifests/frontend
Destinat...	in-cluster	in-cluster	in-cluster	in-cluster
Namesp...	argocd	my-api	my-db	my-frontend
Created ...	08/13/2024 19:07:08 (23 minutes ago)	08/13/2024 19:20:58 (9 minutes ago)	08/13/2024 19:19:57 (10 minutes ago)	08/13/2024 19:24:34 (5 minutes ago)
Last Sync:	08/13/2024 19:24:34 (5 minutes ago)	08/13/2024 19:24:19 (5 minutes ago)	08/13/2024 19:25:18 (4 minutes ago)	08/13/2024 19:24:35 (5 minutes ago)

ApplicationSets

ApplicationSets[34] are an evolution of what the App-of-Apps pattern provides. ApplicationSets are designed to not only help with bootstrapping but also with templating out an Argo CD Application. From a high level, an ApplicationSet can use a single manifest to target multiple Kubernetes clusters. Furthermore, you can use a single manifest to deploy multiple applications from one or more Git repos.

With all that ApplicationSets give you, there is one caveat. There is no way to set up application dependencies with ApplicationSets. It's something that has been proposed (**including using DAG**[35]), and you can track that **upstream**[36]; so you will still need to use App-of-Apps if you need to set up application dependencies.

Argo CD and microservices

You should now be able to organize your microservice applications with Argo CD and deploy them in the required order. It goes without saying that ideally you shouldn't have to instruct Argo CD to do this at all.

Your developers should create microservices based on *eventual consistency*. Each microservice should start in any order and automatically retry all its connections until they are healthy. This means that deployment order does not really matter if your microservices are created correctly. After 2-3 minutes, all dependencies should be satisfied, and the whole application must be healthy as a single entity.

Of course, we realize that changing legacy applications is not always possible, and this is where Argo CD comes in.

Reviewing and diffing deployments

The next topic we will address is understanding the impact of changes in Kubernetes manifests when you use Argo CD.

Adopting Kubernetes has introduced several new complications regarding verifying and validating all the manifests that describe your application. Several tools are available for checking the syntax of manifests, scanning them for security issues, enforcing policies, etc.

But in the most basic case, one of the major challenges is understanding what each change means for your application (and optionally approve/reject the pull request containing that change).

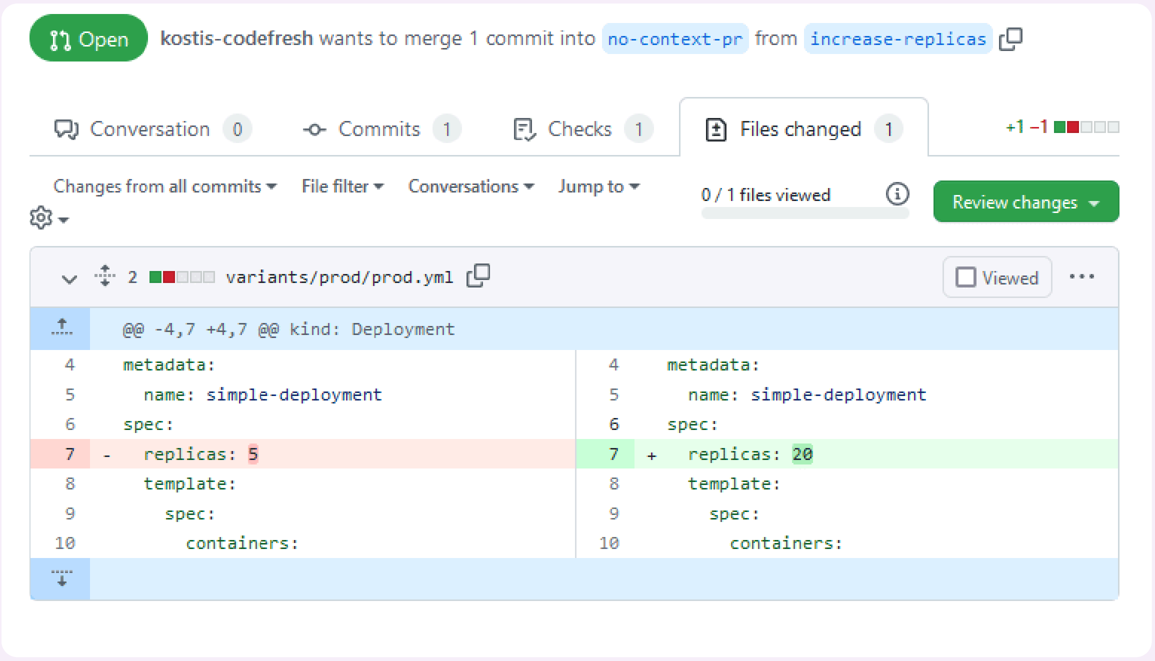
This challenge was already present even outside GitOps, but it has become even more important for teams that use GitOps tooling such as Argo CD for their Kubernetes deployments.

The problem

Any major Git platform has built-in support for showing diffs between the proposed change and the current code when you create a pull request. In theory, the presented diff should be enough for a human to understand what the changes contain and how they will affect the target environment.

In practice, however, several teams have adopted a templating tool (such as Kustomize or Helm) that renders the Kubernetes manifests for a target cluster.

As a quick example, let's say you need to review a pull request with the following changes:


kostis-codefresh wants to merge 1 commit into no-context-pr from increase-replicas

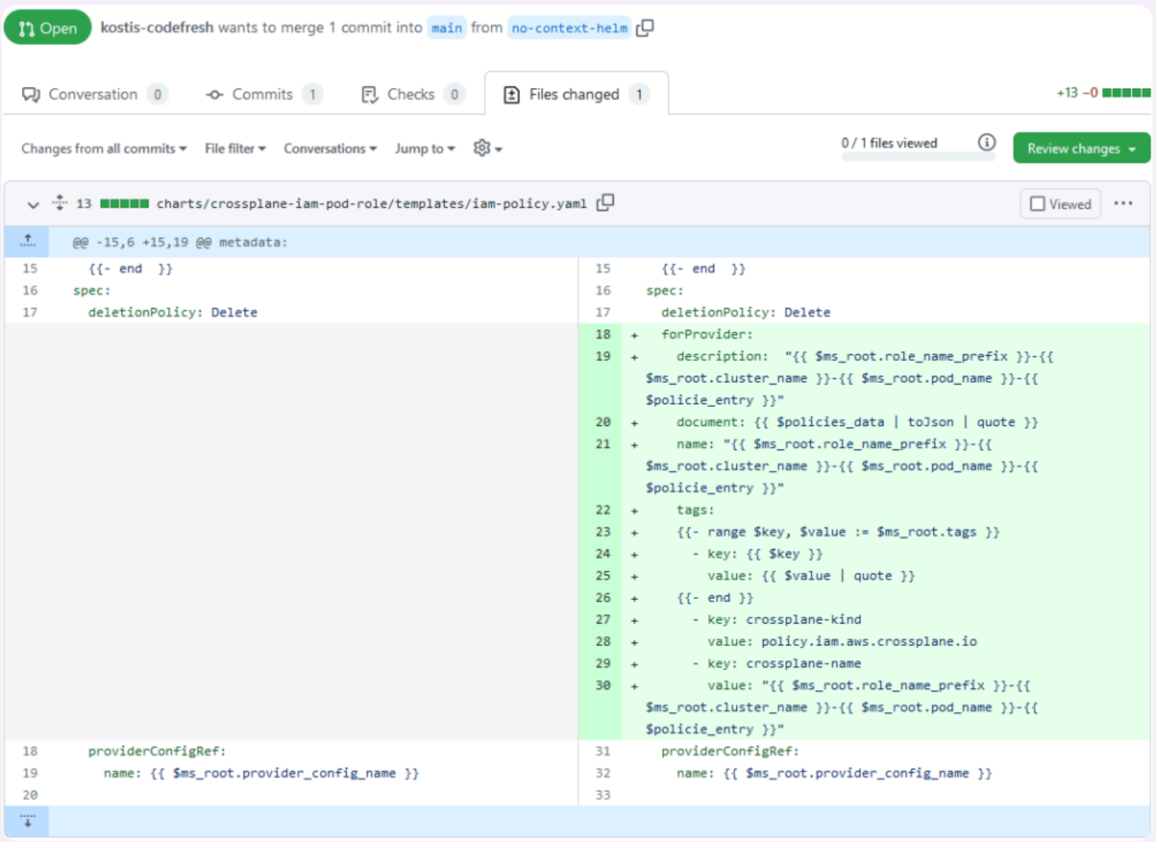
```
variants/prod/prod.yml
@@ -4,7 +4,7 @@ kind: Deployment
  metadata:                              metadata:
    name: simple-deployment                name: simple-deployment
  spec:                                  spec:
-   replicas: 5                        +   replicas: 20
    template:                              template:
      spec:                                  spec:
        containers:                            containers:
```

No context diff

This seems simple enough. You assume that this change will increase the number of replicas to 20. You merge the pull request and ... nothing happens.

What you didn't know is that there is a **downstream Kustomize overlay**[37] that also defines replicas on its own. So the proposed change has no effect at all. The problem was that the pull request contained only a segment of a Kustomize source manifest and didn't show a diff for the end result (the full rendered manifest).

The problem is even more apparent when your organization is using Helm. Let's say that you need to approve a pull request with the following changes:



```
@@ -15,6 +15,19 @@ metadata:
   {{- end  }}
 spec:
   deletionPolicy: Delete
+  forProvider:
+    description:  "{{ $ms_root.role_name_prefix }}-{{ $ms_root.cluster_name }}-{{ $ms_root.pod_name }}-{{ $policie_entry }}"
+    document: {{ $policies_data | toJson | quote }}
+    name: "{{ $ms_root.role_name_prefix }}-{{ $ms_root.cluster_name }}-{{ $ms_root.pod_name }}-{{ $policie_entry }}"
+    tags:
+    {{- range $key, $value := $ms_root.tags }}
+      - key: {{ $key }}
+        value: {{ $value | quote }}
+    {{- end }}
+      - key: crossplane-kind
+        value: policy.iam.aws.crossplane.io
+      - key: crossplane-name
+        value: "{{ $ms_root.role_name_prefix }}-{{ $ms_root.cluster_name }}-{{ $ms_root.pod_name }}-{{ $policie_entry }}"
   providerConfigRef:
     name: {{ $ms_root.provider_config_name }}

```

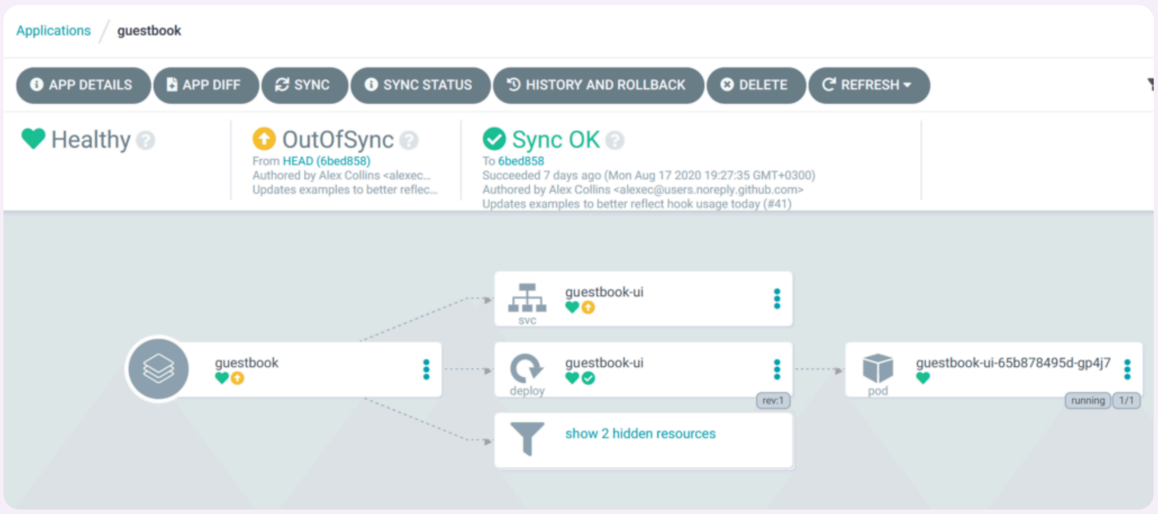
No context Helm diff

It is very difficult for humans to understand what is happening here. You need to mentally run the templates through your head and decide if this change is correct or not. Wouldn't it be nice if the diff had the actual manifest created from this chart?

Essentially, the diff functionality found in your Git system is not enough for complex Kubernetes applications.

Visualizing diffs in the UI

One of the main benefits of using the Argo CD UI during a deployment is the built-in diff feature. When a resource is "out-of-sync" and differs from what is in Git, Argo CD will mark it with a special color/icon. In the following example, somebody has changed the service resource of an application:

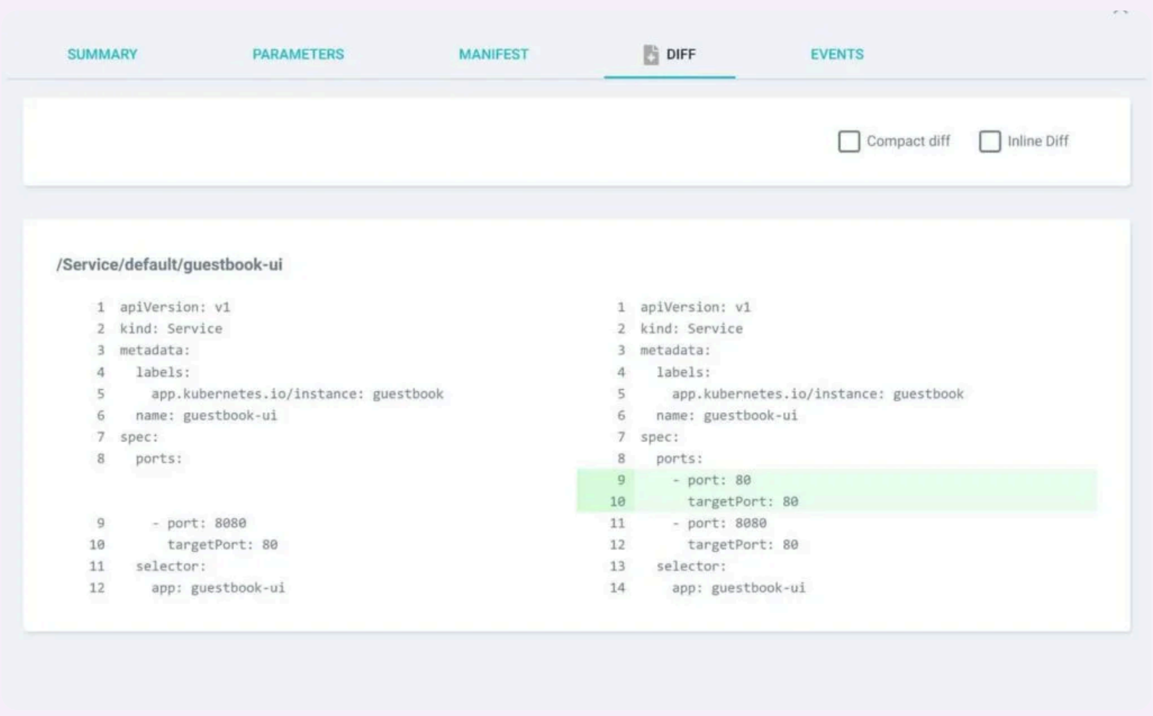


From HEAD (6bed858)
Authored by Alex Collins <alexec...
Updates examples to better reflec...

To 6bed858
Succeeded 7 days ago (Mon Aug 17 2020 19:27:35 GMT+0300)
Authored by Alex Collins <alexec@users.noreply.github.com>
Updates examples to better reflect hook usage today (#41)

Out of Sync Resource

You can then click on the service and see the diff:



/Service/default/guestbook-ui

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/instance: guestbook
  name: guestbook-ui
spec:
  ports:


    - port: 8080
      targetPort: 80
  selector:
    app: guestbook-ui
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/instance: guestbook
  name: guestbook-ui
spec:
  ports:
    - port: 80
      targetPort: 80
    - port: 8080
      targetPort: 80
  selector:
    app: guestbook-ui
```

Argo CD UI diff

The big advantage here is that Argo CD has already integrated support for Kustomize and Helm. The diff you will see is on the final rendered manifests, which is exactly what you want, as you can preview changes in their full context.

Unfortunately, this method also has several disadvantages.

The first is that Argo CD shows only diffs for applications when the **auto-sync**[38] (and self-heal) behavior is disabled. This means that you are losing the main benefit of GitOps. The proper way to follow GitOps is to have auto-sync enabled (and self-heal as well), as this way guarantees the fundamental premise that the cluster and the Git repository contain the same thing - that the desired state and actual state have not diverged.

The second problem regarding Continuous Delivery is that the diff on the manifests is shown when the changes are already committed and pushed. This is too late to perform a serious review. Ideally, you want to review changes as early as possible. A pull request lets you add comments, talk with your team about the changes, and also reject the pull request altogether without affecting a production system.

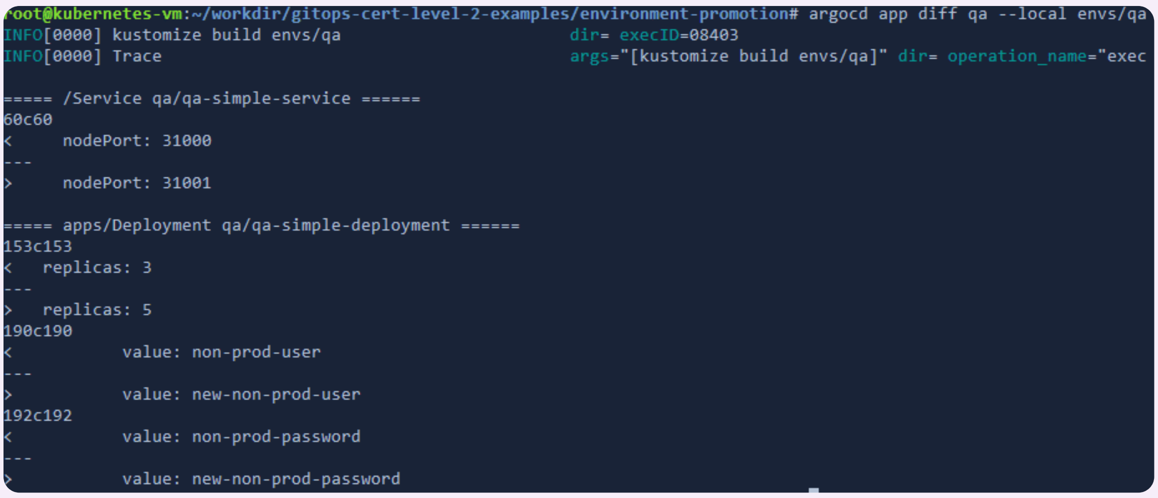
Using the built-in diff functionality in the Argo CD GUI is great for validating a change and doing a final review just before production. However, it should not be the main review milestone for a manifest change. And ideally, **you should setup all your applications to sync automatically**[39], so having this diff process is not available in the first place.

Running local diffs with the CLI

The built-in diff UI in Argo CD is shown very late in the delivery process. Can we use the same diff approach earlier in the life of a change?

It turns out that the Argo CD CLI also comes with **a diff command**[40]. This diff command takes a `–local` parameter that lets you compare what is happening in the cluster against any local files, which you don't have to push (or even commit). It will also automatically run your favorite template tool as you define it in the Argo CD application.

Here is how it looks:



```
root@kubernetes-vm:~/workdir/gitops-cert-level-2-examples/environment-promotion# argocd app diff qa --local envs/qa
INFO[0000] kustomize build envs/qa                           dir= execID=08403
INFO[0000] Trace                                             args="[kustomize build envs/qa]" dir= operation_name="exec

===== /Service qa/qa-simple-service ======
60c60
<     nodePort: 31000
---
>     nodePort: 31001

===== apps/Deployment qa/qa-simple-deployment ======
153c153
<   replicas: 3
---
>   replicas: 5
190c190
<           value: non-prod-user
---
>           value: new-non-prod-user
192c192
<           value: non-prod-password
---
>           value: new-non-prod-password
```

Argo CD Local diff

This approach is very promising, as you could, in theory, use it inside a CI system with the following process:

- Open a pull request with the suggested changes
- Have your CI system check out the pull request
- Run inside a CI pipeline `argocd diff —local` against the cluster where you intend to deploy the changes in the pull request (It also uses again the built-in support for kustomize/Helm within Argo CD)
- Present the diff to the user to make decisions about the pull request

This CI-based approach sounds great in theory, but it has several shortcomings in practice.

You must provide your CI pipeline credentials to access the cluster where you installed Argo CD. However, providing the CI pipeline with cluster credentials forfeits one of GitOps's main benefits - the pull mechanism, where the credentials stay within the cluster.

An even bigger concern, however, is what happens when you have multiple clusters. Which cluster should you pick to compare against? What if the chosen cluster has CRDs or other resources that are custom to it?

This process can also become very complex with remote or secure cluster instances. For example, if you have an Argo CD cluster in Asia and your CI system runs in the US, connectivity between the two might be very slow or impossible.

In summary, `argocd diff—local` is great for local experimentation and quick ad hoc checks, but there is a better way to achieve the same result for a production deployment process (spoiler: it doesn't involve the Argo CD CLI at all, nor does it need cluster access).

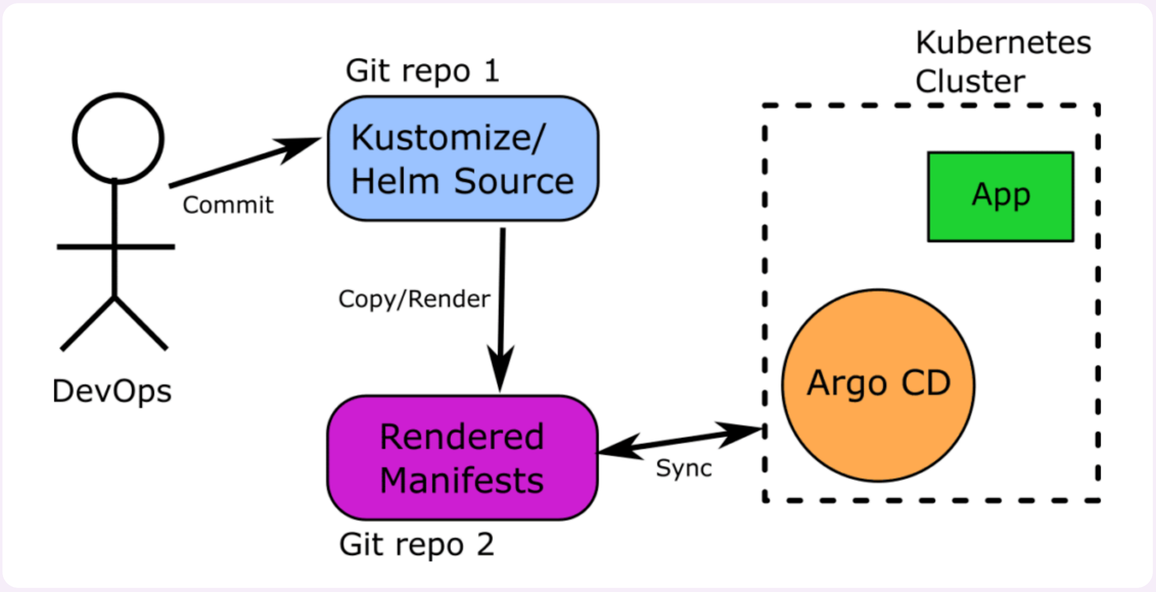
Pre-rendering manifests in a second Git repository

Let's take a step back. We have been looking for ways to show an enhanced diff as part of a pull request and ignore the existing diff provided by the Git provider, as we have seen that this doesn't work with the final manifest.

However, there is a way to enhance the built-in diff and make it work on the final manifests.

The solution is to use 2 GitOps repositories, for each application/cluster. One Git repository has the manifests in their unprocessed form (e.g., as Kustomize overlays) as before. There is now a second repository that has the final rendered manifests, and Argo CD is pointed at the latter.

Here is how it would look:



PreRender manifests in second Git repository

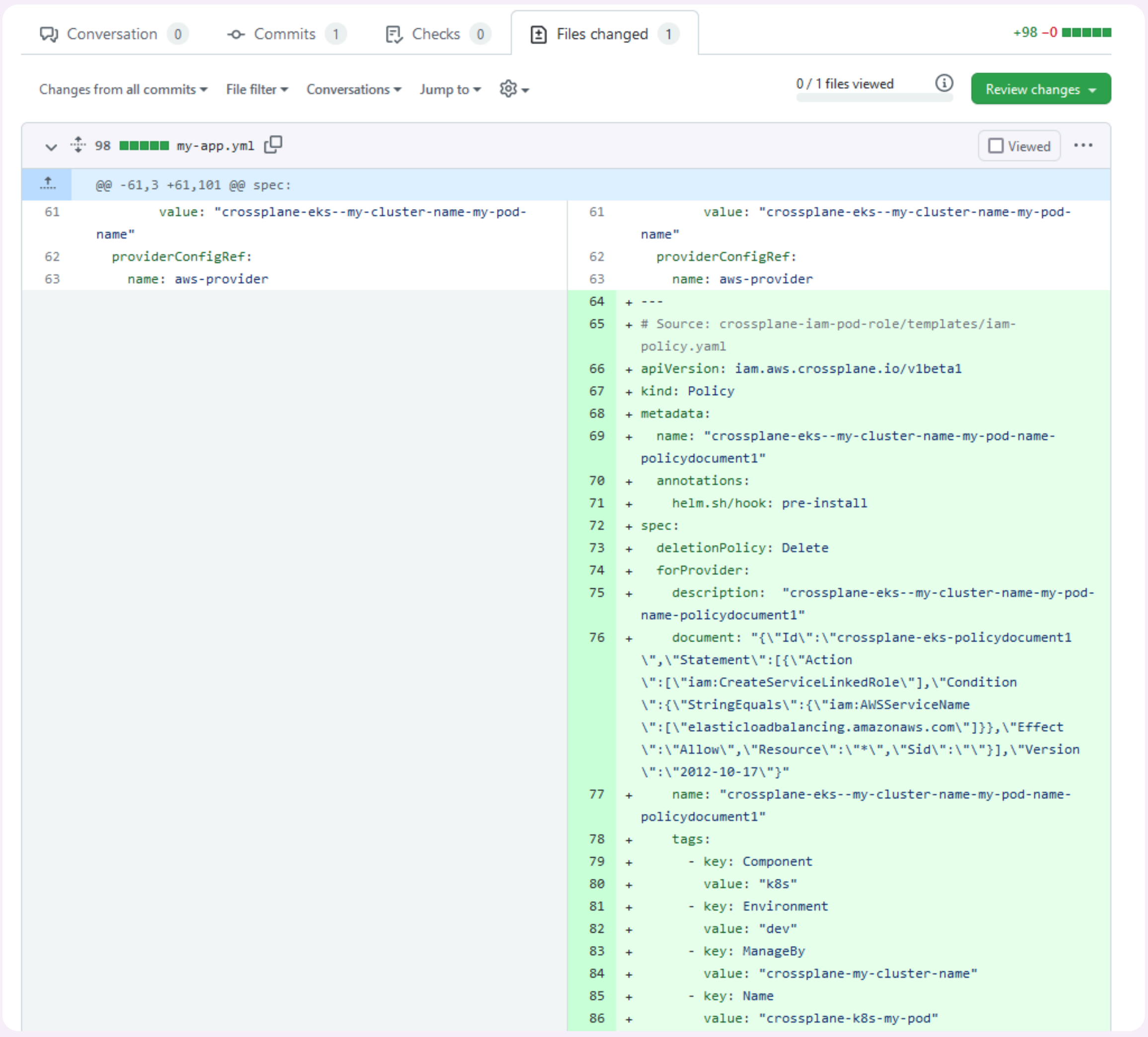
If you have ever used a preprocessor or code generator, this process should be familiar. Essentially, there is an automated process like a CI system that does the following:

1. A human creates a pull request on the "Source" Git repo with the suggested change.
2. A "copy" process takes the contents of the pull request and applies the respective template tool (i.e., Helm/Kustomize) to create the final rendered manifest.
3. A second pull request is opened automatically on the "Rendered" Git repo with the contents of the manifests.
4. A human sees the diff of the second pull request, and this time the diff is between rendered manifests and not snippets/segments.
5. If the pull request is approved, it is merged into both Git repositories, meaning the second repository has always rendered manifests.
6. Argo CD monitors the second repository and applies the changes (the integrated support for Helm and Kustomize within Argo CD itself is not used at all; Argo CD is only syncing raw manifests).

The rendered manifests approach is a valid process, and I have seen it successfully used in several companies.

The big advantage is that **the diff you get in the Git provider**[41] provides you with the full information about what will change in the application *after* all manifests are processed.

Here is the same example with the Helm chart, but this time, we are using a second Git repository that stores the rendered manifest:



```
@@ -61,3 +61,101 @@ spec:
          value: "crossplane-eks--my-cluster-name-my-pod-name"
    providerConfigRef:
      name: aws-provider
+ ---
+ # Source: crossplane-iam-pod-role/templates/iam-policy.yaml
+ apiVersion: iam.aws.crossplane.io/v1beta1
+ kind: Policy
+ metadata:
+   name: "crossplane-eks--my-cluster-name-my-pod-name-policydocument1"
+   annotations:
+     helm.sh/hook: pre-install
+ spec:
+   deletionPolicy: Delete
+   forProvider:
+     description:  "crossplane-eks--my-cluster-name-my-pod-name-policydocument1"
+     document: "{\"Id\":\"crossplane-eks-policydocument1\",\"Statement\":[{\"Action\":[\"iam:CreateServiceLinkedRole\"],\"Condition\":{\"StringEquals\":{\"iam:AWSServiceName\":[\"elasticloadbalancing.amazonaws.com\"]}},\"Effect\":\"Allow\",\"Resource\":\"*\",\"Sid\":\"\"}],\"Version\":\"2012-10-17\"}"
+     name: "crossplane-eks--my-cluster-name-my-pod-name-policydocument1"
+     tags:
+       - key: Component
+         value: "k8s"
+       - key: Environment
+         value: "dev"
+       - key: ManageBy
+         value: "crossplane-my-cluster-name"
+       - key: Name
+         value: "crossplane-k8s-my-pod"
```

Helm diff with full context

However, I am against this process, as it greatly complicates things and increases the number of moving parts.

- It doubles the number of repositories for any given application (or branches if you use multiple branches).
- It introduces another point of failure, which is the copy process that converts source YAML to final manifests.
- It completely bypasses the effort put into Argo CD to process manifests on its own.
- It might be confusing for people who now have 2 Git repositories to work with and opens the possibility of mistakes on both ends (either committing stuff on the "source" repo that never makes it to the "rendered" repo or vice versa).

In general, this is an overkill solution for a problem that can be solved more elegantly, as we will see later in the article. Still, if you follow this approach and it works for you, ensure you have safeguards and monitoring in place, especially for the copy/commit automated process.

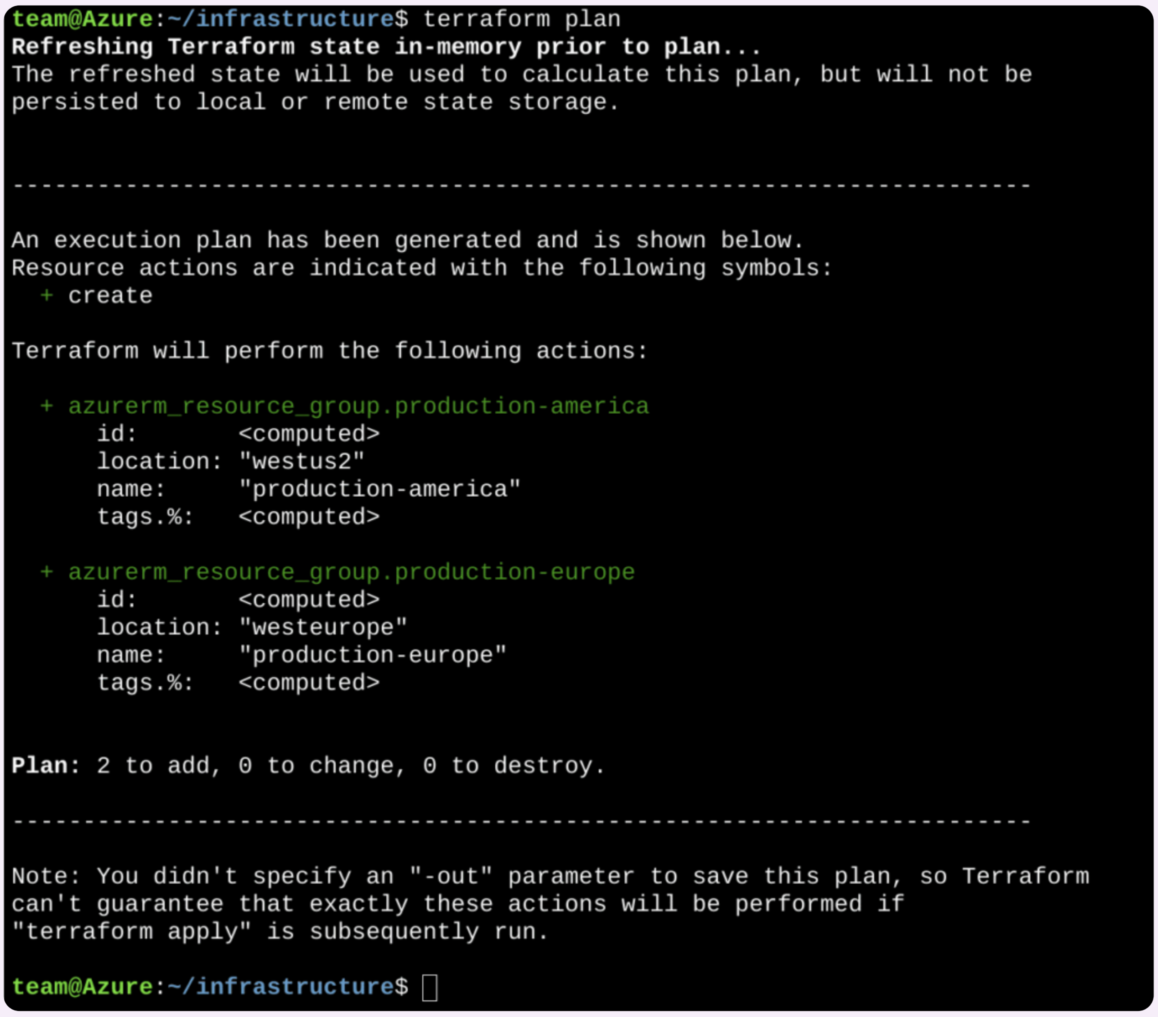
Intermission: Preview Terraform plans

You might think that previewing the full manifests for a pull request is a new problem that Kubernetes introduced. It isn't. Several tools before Kubernetes had to deal with the same issue, and it would make sense to look at what they do.

The most obvious candidate to examine is Terraform. If you are not familiar with Terraform, it is a declarative tool that lets you define your infrastructure in an **HCL file**[42] and then **"apply"**[43] your changes.

Terraform users have a very similar problem. A pull request that contains Terraform changes, especially in big projects, is not immediately clear for a human to understand, unless you are an expert on running Terraform mentally in your head.

To solve this issue, Terraform has a **"plan"**[44] command which reads the changes, decides what it will do, and prints a nice summary of all the proposed changes without actually doing anything.



```
team@Azure:~/infrastructure$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + azurerm_resource_group.production-america
      id:        <computed>
      location:  "westus2"
      name:      "production-america"
      tags.%:    <computed>

  + azurerm_resource_group.production-europe
      id:        <computed>
      location:  "westeurope"
      name:      "production-europe"
      tags.%:    <computed>


Plan: 2 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

team@Azure:~/infrastructure$ 
```

Terraform plan

The plan functionality in Terraform is crucial to Terraform teams as it removes the guesswork about what Terraform will do when the changes are applied.

With this summary at hand, the next step is obvious. We can attach the output of the plan command to the pull request. Humans can now look at both the diff of the HCL files and the plan summary and decide if the change is valid or not.

This workflow is so common that an open-source project - **https://www.runatlantis.io/**[45] - does exactly that.

1. You make your changes to the Terraform files.
2. You create a pull request.
3. Atlantis runs the "plan" command and attaches the result to the PR.
4. You can then approve/comment on the PR as a human.
5. Atlantis then runs the "apply" command to modify your infrastructure.
6. Atlantis locks the workspace until you merge the PR, preventing a second PR from overriding the changes before you merge the first PR.

This workflow is very effective for end-users but has several security drawbacks similar to those of the argo CLI diff approach:

- You need bidirectional communication between your Git provider and the server that runs Atlantis.
- The server that runs Atlantis will run Terraform on its own and therefore needs **all credentials that Terraform has**[46] in your organization.
- The Atlantis server also needs access to your remote Terraform state. Essentially, Atlantis has the keys to your kingdom.

Another downside of Atlantis is that it's pull request based, whereas proprietary Terraform CD tools on the market feature a dashboard where every project/workspace can be browsed, similar to how you have a dashboard in Argo CD where you can see your applications and whether or not Argo CD has synced them.

In theory, we could create a similar system for `argocd diff –local`, but given the security implications, there is a much better approach that GitOps principles help enable.

Still, attaching a diff in a pull request for greater context offers several advantages for the end user that we cannot overstate.

Render Kubernetes manifests on the fly

One of the most important principles of GitOps is that the cluster state is always the same as what is described in the git state. We really like how Terraform Atlantis works, but there is a way to improve the workflow and make it more secure and robust by taking advantage of the Argo CD guarantee for GitOps.

The Terraform CLI that runs on the Atlantis instance needs credentials to your infrastructure because it must both read the **Terraform state**[47] and also create the actual infrastructure once changes are "applied".

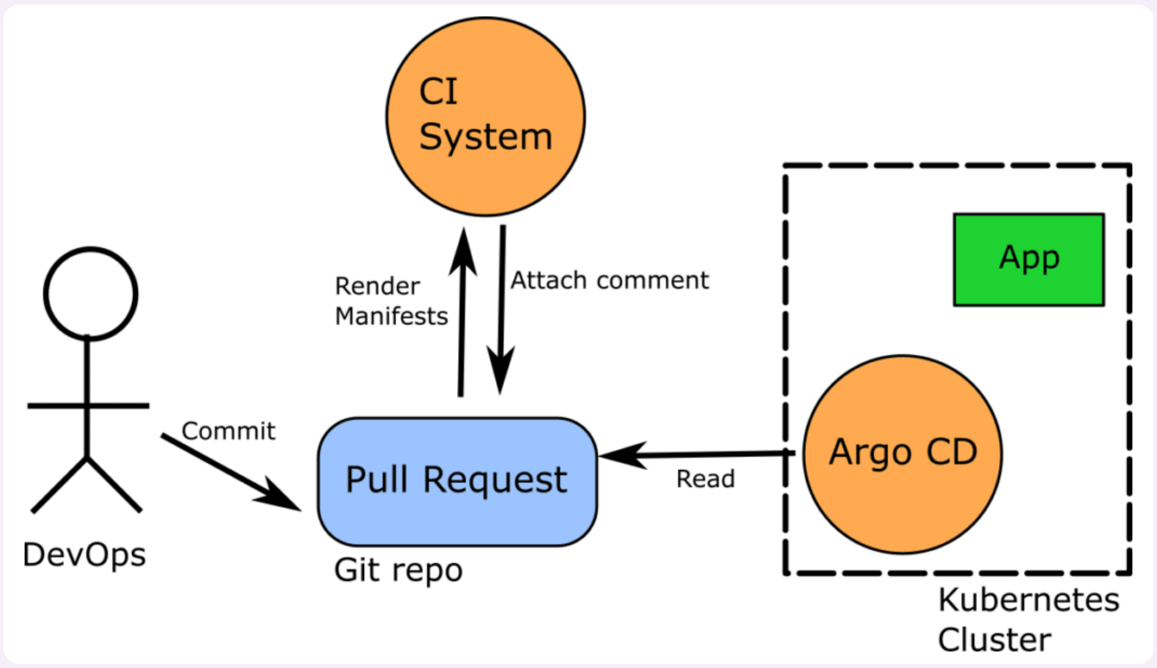
With Argo CD, we can completely bypass this limitation because we already have the cluster state. The pull request target stores it!

Having the cluster state in Git means we don't need any credentials for the cluster or Argo CD. We can run a diff between the files of the pull request and the branch it is targeted at. We will also run a preprocessing step for the template solution (Helm or Kustomize) to get the full manifest.

So the full process is as follows:

1. Somebody opens a pull request to the manifest repo.
2. We check out the code of the pull request and run Kustomize, Helm, or other templating tools to have the final rendered manifests of what is changed.
3. We check out the code of the branch that the Pull Request targets (e.g. main) and find the same environment, and again run the same templating tool to get the final manifest
4. We run a diff between the final manifests from the two previous steps.
5. We show the diff to the human operator, who will decide if they will merge the pull request or not.

The beauty of this approach is that **unlike Atlantis**[48], we never access the Argo CD cluster. All information is coming from Git (an advantage of using GitOps). The Git-based approach means that your Argo CD cluster could be in China with a very slow or even isolated connection, and your CI server doesn't need to know anything about it. Your Argo CD server's location and security access are now irrelevant, as we don't interact with it in any way.



Render manifests on the fly

Notice in the diagram above that, unlike Atlantis, our CI server has a direct connection only to the Git repository. The Argo CD instance still cares only about the Git repository it monitors.

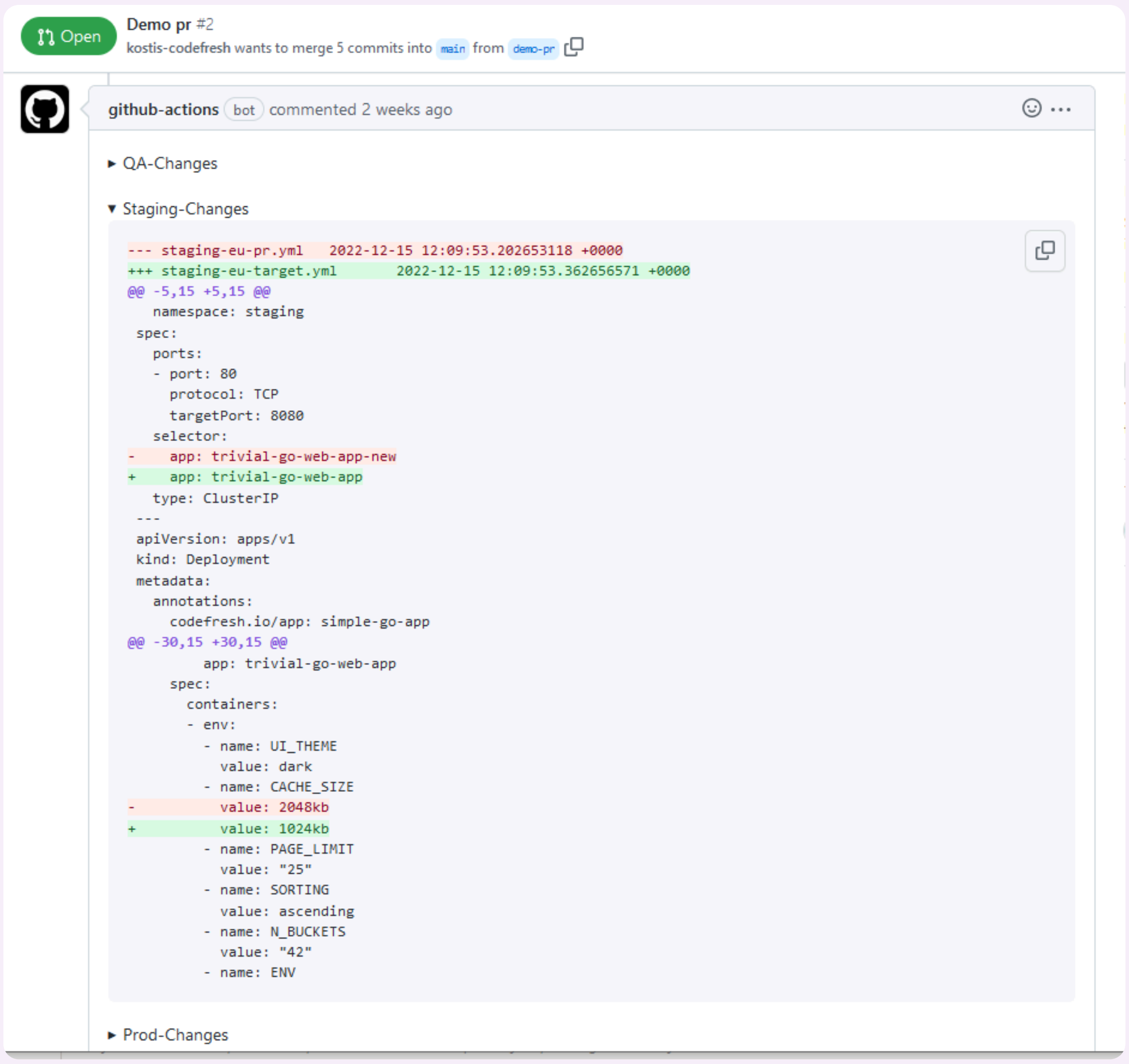
This makes our approach much more secure as the cluster's credentials remain within it, and we only interact with the Git repository.

One thing to notice here is that, unlike Atlantis or the `argocd diff` command, we are not comparing the desired state in Git to the actual state (acquired from the cloud provider's API or Kubernetes API); we are comparing two versions of the desired state stored in different branches of a Git repository. While this approach is a good enough approximation, it is not 100% equivalent to the `argocd diff` one.

A corner case scenario would be Helm Capabilities - built-in variables populated by querying the K8s cluster for API version and available resources. Some Helm templates use this information to render correct resource versions appropriate for a specific cluster's version and available CRDs. This information has to be supplied manually to the `helm template` command to achieve parity with `argocd diff`.

Attaching the full manifest diff to a pull request

The icing on the cake is that we will also attach the full manifest diff to the pull request as Atlantis does. This is how it would look:



Open Demo pr #2
kostis-codefresh wants to merge 5 commits into main from demo-pr

github-actions bot commented 2 weeks ago

▸ QA-Changes

▾ Staging-Changes

```
--- staging-eu-pr.yml   2022-12-15 12:09:53.202653118 +0000
+++ staging-eu-target.yml       2022-12-15 12:09:53.362656571 +0000
@@ -5,15 +5,15 @@
   namespace: staging
 spec:
   ports:
   - port: 80
     protocol: TCP
     targetPort: 8080
   selector:
-    app: trivial-go-web-app-new
+    app: trivial-go-web-app
   type: ClusterIP
 ---
 apiVersion: apps/v1
 kind: Deployment
 metadata:
   annotations:
     codefresh.io/app: simple-go-app
@@ -30,15 +30,15 @@
         app: trivial-go-web-app
     spec:
       containers:
       - env:
         - name: UI_THEME
           value: dark
         - name: CACHE_SIZE
-          value: 2048kb
+          value: 1024kb
         - name: PAGE_LIMIT
           value: "25"
         - name: SORTING
           value: ascending
         - name: N_BUCKETS
           value: "42"
         - name: ENV
```

▸ Prod-Changes

Environment diff

We now need to explain to users that they should no longer look at the automatic diff of the pull request because it only tells part of the story. Instead, they should look at our attached diff, get the complete picture of what has changed, and make decisions accordingly.

The attached diff is especially important for people who use Helm, as you can see a diff between plain YAML instead of trying to run manually Golang templates in your head.

Enforcing changes during environment promotion

If you follow this diff approach where the full manifests' changes are shown in the pull request, it will be much easier for you and your team to collaborate on GitOps changes as everybody will have the full context of each incoming change.

However, a secondary benefit of this diff approach is knowing what is *not* changed.

I wrote an **article about promotions between GitOps environments using folders**[49]. A lot of people asked about how you can guarantee that extracting a common setting from downstream Kustomize overlays and promoting it your base overlay can be safely executed in a single step.

I was really puzzled by this query until I realized that most people asking this question looked at the simpler diff of the pull request and thus lacked the full context of the change.

Let's take an example. You have two environments **qa** and **staging** with the following settings:

```
UI_THEME=light
CACHE_SIZE=2048kb
SORTING=ascending
N_BUCKETS=42
```

You want to add a new setting called `PAGE_LIMIT=25` and gradually promote it, first to QA, and then to staging.

You modify/commit the **qa environment**[50].

```
UI_THEME=light
CACHE_SIZE=2048kb
PAGE_LIMIT=25
SORTING=ascending
N_BUCKETS=42
```

The deployment goes ok, and you make the **same change to the Staging environment**[51]. It works fine there as well.

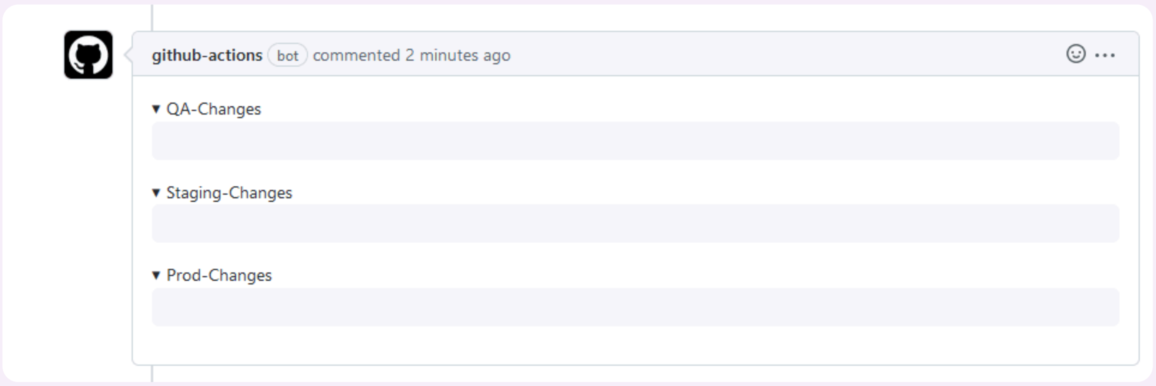
Now you decide that this new setting should be the same across both environments, and you decide to move it to **the parent overlay**[52], which is common to all non-prod environments.

So the actions you take are:

1. Delete the setting from the QA environment
2. Delete the setting from the Staging environment
3. Add the setting into the parent overlay that both environments depend on
4. Commit/push all the above in a single step

Many people were concerned about this process and asked how to ensure it would work without affecting the existing environments.

We can finally answer this question by simply looking at the **enhanced diff of the above commit**[53]


github-actions bot commented 2 minutes ago

▾ QA-Changes

▾ Staging-Changes

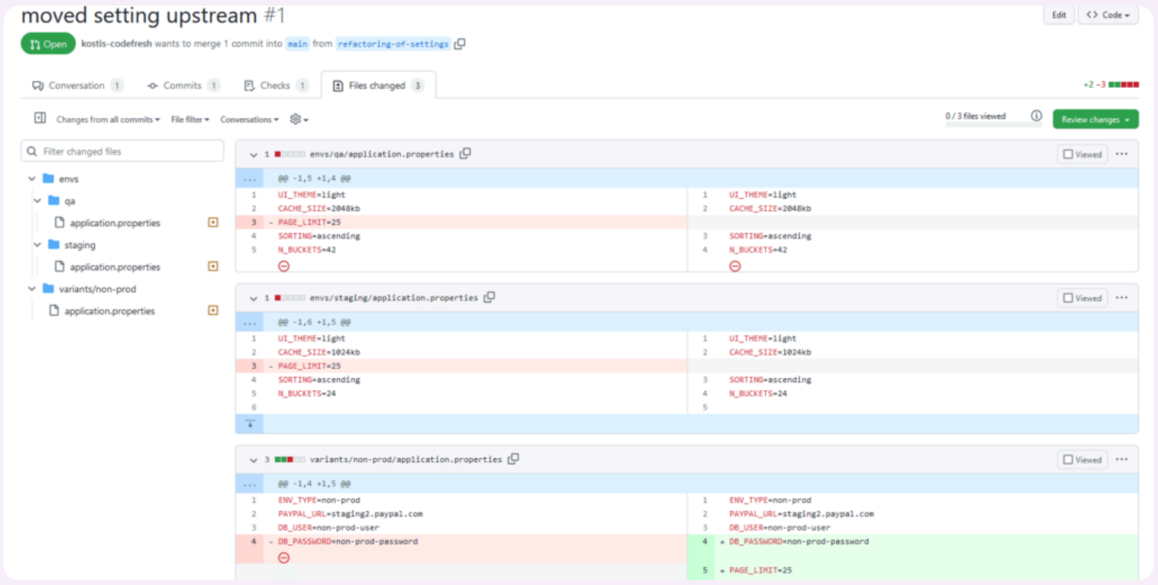
▾ Prod-Changes

No change in the final manifests

That's right. All diffs are completely empty. Even though there are changes in the individual Kustomize files, the result (the rendered manifests) is *exactly* the same.

This means that if you approve this pull request, Argo CD will do absolutely nothing, and you are sure that all environments will be oblivious to this refactoring.

Of course, the basic diff of the pull request is not that smart and shows the **diff changes in text in the individual files**[54].



moved setting upstream #1

Open kostis-codefresh wants to merge 1 commit into `main` from `refactoring-of-settings`

1 envs/qa/application.properties

```
@@ -1,5 +1,4 @@
  UI_THEME=light
  CACHE_SIZE=2048kb
- PAGE_LIMIT=25
  SORTING=ascending
  N_BUCKETS=42
```

1 envs/staging/application.properties

```
@@ -1,6 +1,5 @@
  UI_THEME=light
  CACHE_SIZE=1024kb
- PAGE_LIMIT=25
  SORTING=ascending
  N_BUCKETS=24
```

3 variants/non-prod/application.properties

```
@@ -1,4 +1,5 @@
  ENV_TYPE=non-prod
  PAYPAL_URL=staging2.paypal.com
  DB_USER=non-prod-user
- DB_PASSWORD=non-prod-password
+ DB_PASSWORD=non-prod-password
+ PAGE_LIMIT=25
```

Plain diff shows changes

So in this scenario, we have the extreme case when the built-in diff of the pull request doesn't have the full context of what is going on because it doesn't understand the full manifests.

Understand the impact of your manifest changes

Previewing changes before applying them is a pillar of modern software automation, but in the case of Kubernetes applications, this is not always straightforward because the manifests are templated.

We have now seen several ways of previewing the changes in Argo CD applications:

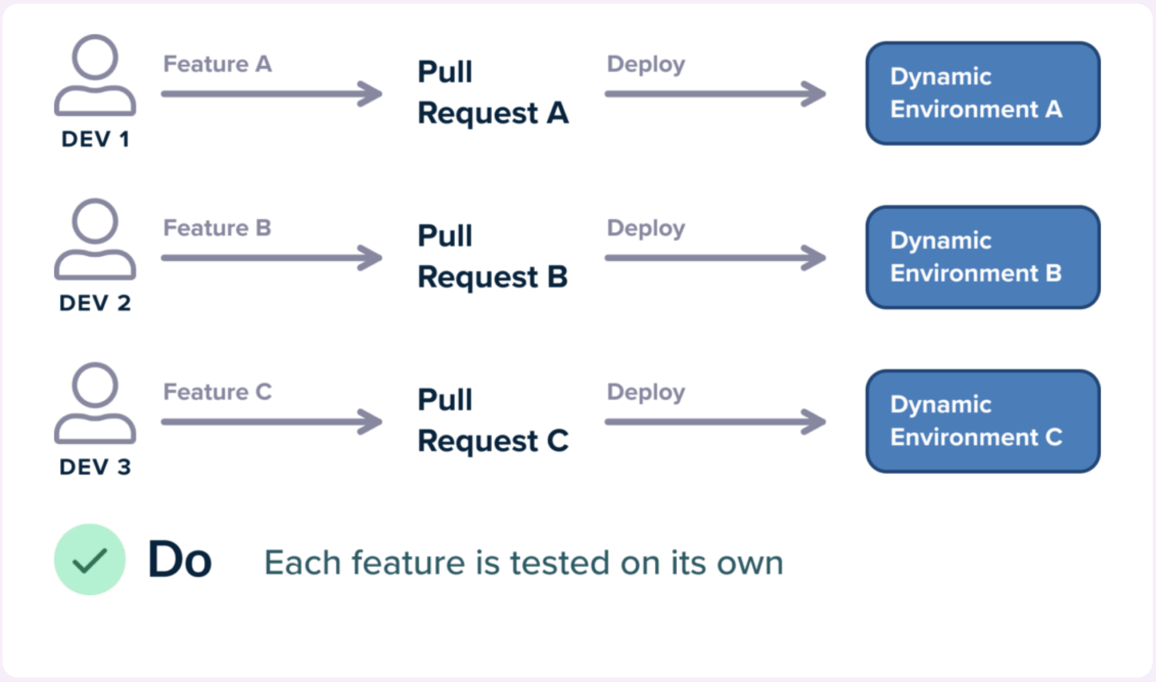
1. Basic diff of the Git platform (not recommended)
2. Native diff of the Argo CD UI
3. Diff local files with the Argo CD CLI
4. Pre-rendering manifests in a second Git repository
5. Rendering manifests on the fly for each Pull request (recommended)

We hope that this process is helpful for you and your team, especially when it is combined with static analysis, syntax validation, security scans, and other sanity checks that run against your Kubernetes manifests.

Creating preview environments for pull requests

The last use case we will explore in this whitepaper is temporary/preview environments.

In our **big guide for Kubernetes deployments**[55], we explained the benefits of using dynamic environments for testing.



Dynamic environment for each pull request

The general idea is that each developer gets a preview environment instead of having a fixed number of testing/QA environments. The environment gets created on the fly when you open a pull request. Typically, it gets destroyed when you merge the pull request, or after a specific amount of time.

Previously, we explained how to implement this pattern using just **Helm applications and an Ingress**[56] on a Kubernetes cluster with a traditional deployment pipeline.

If your organization has moved to GitOps, you might wonder if you can replicate this setup with Argo CD. The answer is yes!

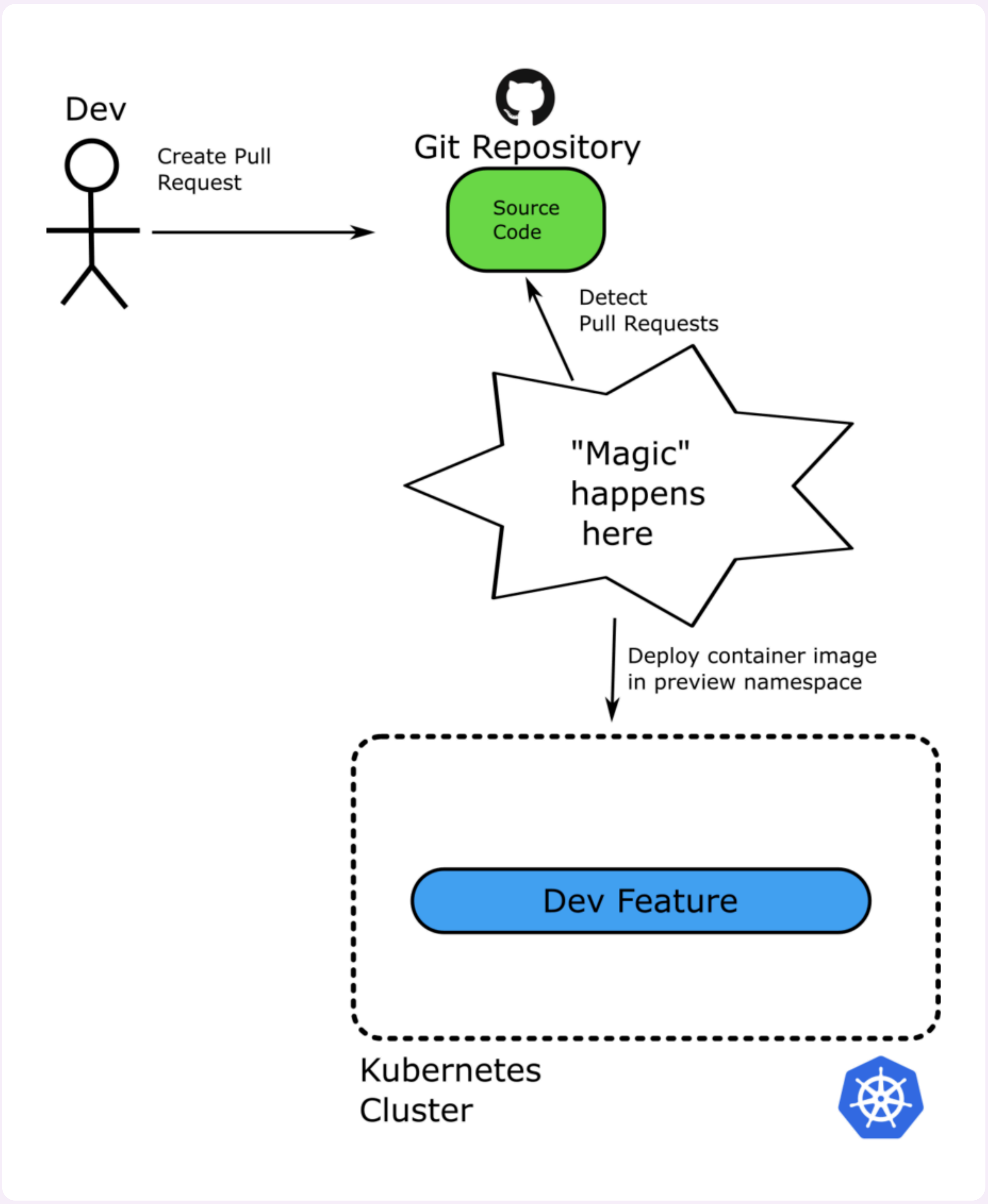
In this section, you'll learn how to create preview environments with Argo CD by using **the Pull Request generator**[57] for this scenario.

Keeping developers happy

There are several approaches for creating preview environments with Argo CD. One major decision is whether you want a preview environment created for a pull request in the application source code, or a pull request on the Kubernetes manifests. As a reminder, **it's always a good idea to separate source code**[58] from manifests in two separate Git repositories.

Both approaches are valid, but we will focus on pull requests created on the source code. Developers open these pull requests when they've implemented a feature and want to test or share it with their team.

We don't want to force developers to learn how Argo CD or Kubernetes works. So, as far as they're concerned, they open a pull request as usual, and after some time, a new preview environment gets created with the contents of the pull request.



Magic deployments reduce developer load

There are many ways to create preview environments for Kubernetes. For simplicity, we'll follow the same assumptions **as in the Helm article**[56].

1. We'll use a single cluster for all preview environments.
2. Argo CD will run inside the same cluster.
3. We'll deploy each pull request to its own namespace.
4. The name of the namespace will match the name of the pull request.

If two developers create two pull requests named "my-billing-feature" and "fix-queue-size", two new namespaces with the same names get created for the respective deployments. Each namespace contains the container from the respective Continuous Integration build, so both developers can test their features in isolation.

There are more advanced scenarios, like **creating a virtual cluster**[59] per pull request, but they are out of scope for this paper.

Generating Argo CD applications automatically

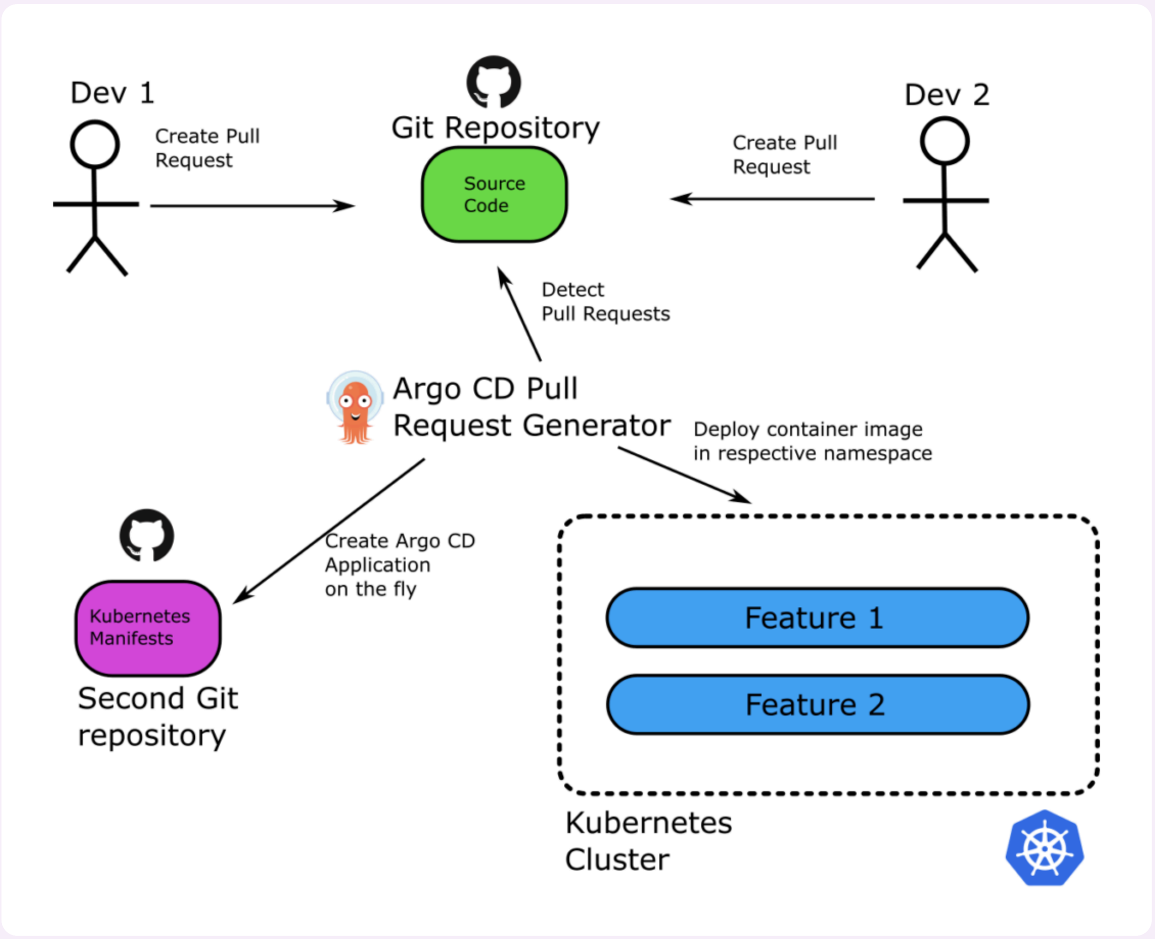
The basic concept of an Argo CD installation **is the application**[60], which is a link between a Git repository and a destination cluster. We explained how to create Argo CD applications using generators in our **article about Argo CD application sets**[11].

Things are a bit different in preview environments, as we don't have a folder in Git that represents the application. If we did, we could use the Git generator to create preview environments as more folders are created/destroyed.

Theoretically, we could instruct our Continuous Integration system to commit new folders on the manifest repo for each preview application. But this process is too cumbersome, and there's no need to go down that route. The **Pull Request generator**[57] solves this problem by creating applications directly from pull requests instead of folders.

Unlike other Argo CD generators, the PR Generator is special because it can monitor any Git repository for pull requests. It doesn't have to be the same one that has manifests. This lets you monitor the Git repository for source code while still deploying manifests from a different Git repository.

Here's our updated architecture where we've expanded the "magic" section from the previous picture.



Magic has been replaced by the Argo CD Pull Request generator

The whole process goes like this:

1. There's one Git repository with just the source code of the application. There's another Git repository with Kubernetes manifests (Helm, Kustomize, or plain manifests).
2. A developer creates a branch on the source Git repository and starts working on a feature.
3. When the developer finishes, they create a pull request on the branch.
4. Argo CD notices the pull request on the source code Git repository and creates a new application for it.
5. The Pull Request generator takes the manifests from the infrastructure repository and templates/modifies them according to the pull request in the source code repo.
6. A new application gets created and deployed in the Kubernetes cluster.
7. The developer accesses the new application, reviews it, runs unit tests, shares it, etc.
8. When the pull request gets merged/closed, Argo CD automatically discards the preview environment.
9. The process works in parallel for all developers working on features.

This is the heart of the process. We must consider several supporting factors to create a rock-solid experience for developers.

Coordinating with the Continuous Integration system

The main goal of the preview environment is to let developers test their new features in isolation. Before dealing with Argo CD deployments, we need a Continuous Integration system that covers two basic requirements:

1. It automatically builds a container every time you open a pull request.
2. It pushes the container using the Git hash of the source code as a tag.

We use Codefresh CI in our example application, but any CI system should be able to cover these requirements.

You can find the source code of the example application at **https://github.com/kostis-codefresh/preview-env-source-code**[61]. All the manifests are at **https://github.com/kostis-codefresh/preview-env-manifests**[62]. Note that we follow the **Argo CD best practices**[58] for splitting source code from Kubernetes manifests.

This example uses Kustomize. For an example with Helm, see **Piot's guide on the same subject**[63].

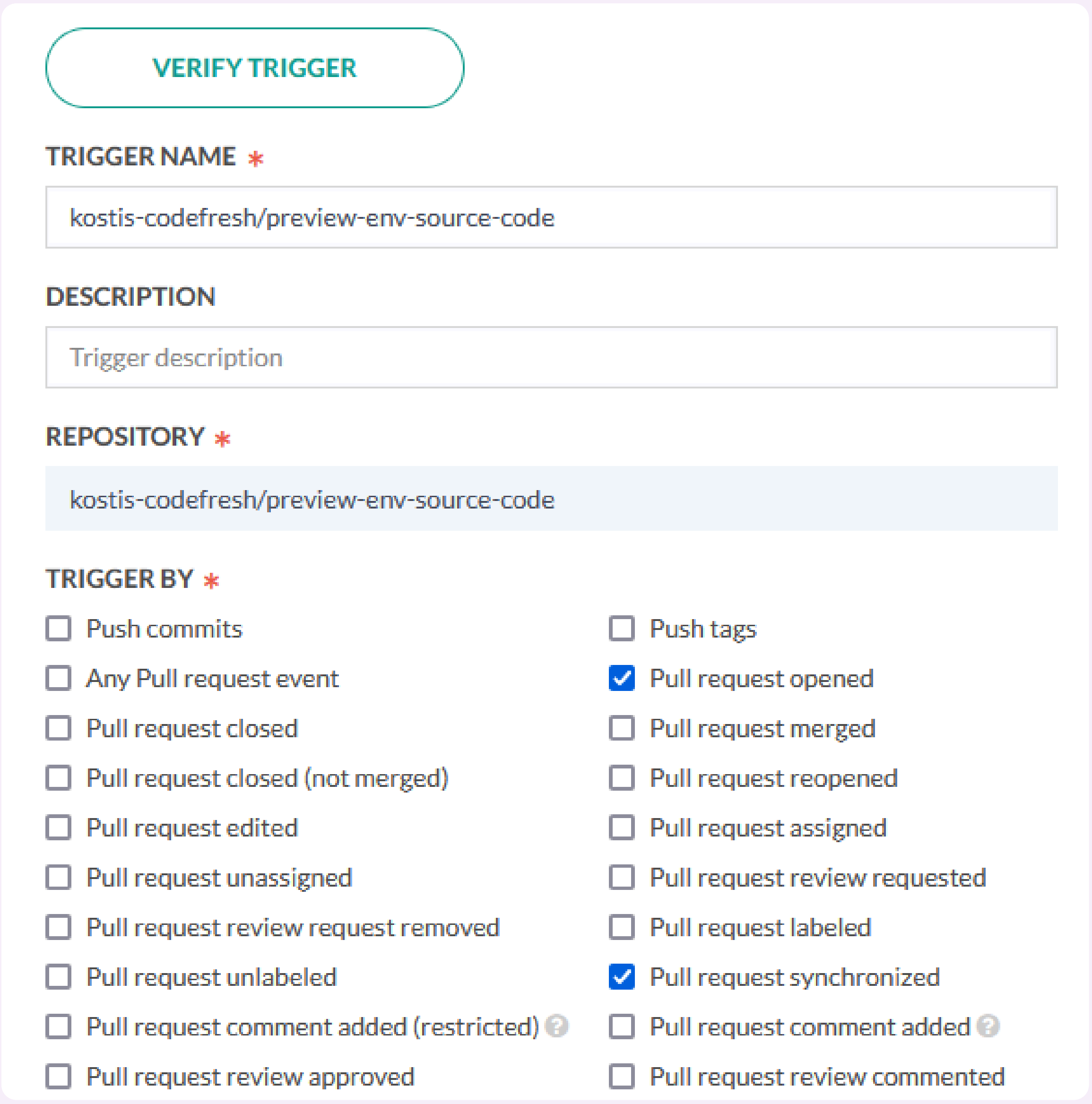
Here's a minimal Codefresh pipeline that builds a container image with the same tag as the source code:

```
version: "1.0"
stages:
  - "clone"
  - "build"
steps:
  clone:
    title: "Cloning repository"
    type: "git-clone"
    repo: "kostis-codefresh/preview-env-source-code"
    revision: "${{CF_REVISION}}"
    git: "github-1"
    stage: "clone"


  build:
    title: "Building Docker image"
    type: "build"
    image_name: "kostiscodefresh/my-preview-app"
    working_directory: "${{clone}}"
    tag: "${{CF_SHORT_REVISION}}"
    dockerfile: "Dockerfile"
    stage: "build"
```

The important part here is the "tag" line, which instructs Codefresh to tag the Docker image with the same hash as the source code. The short revision variable represents the Git hash and is automatically injected by Codefresh with the commit that we checked out.

The second requirement is to run this pipeline only for pull request events. This happens out-of-the-box with **Codefresh Git triggers**[64].



VERIFY TRIGGER

TRIGGER NAME *

kostis-codefresh/preview-env-source-code

DESCRIPTION

Trigger description

REPOSITORY *

kostis-codefresh/preview-env-source-code

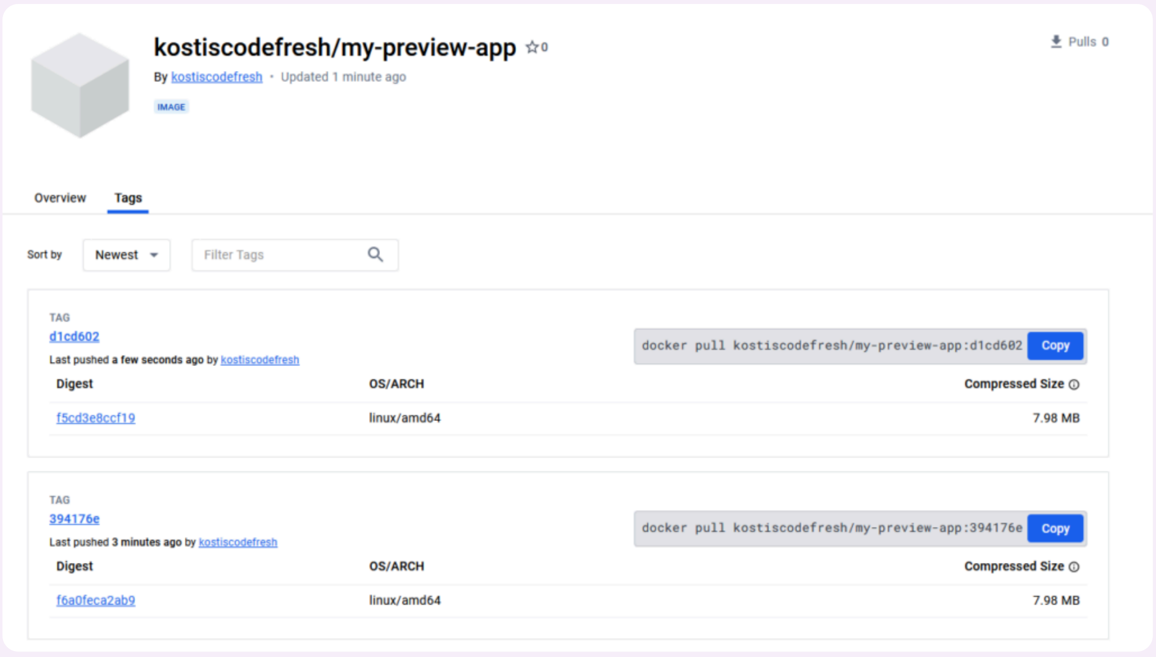
TRIGGER BY *

☐ Push commits	☐ Push tags
☐ Any Pull request event	☑ Pull request opened
☐ Pull request closed	☐ Pull request merged
☐ Pull request closed (not merged)	☐ Pull request reopened
☐ Pull request edited	☐ Pull request assigned
☐ Pull request unassigned	☐ Pull request review requested
☐ Pull request review request removed	☐ Pull request labeled
☐ Pull request unlabeled	☑ Pull request synchronized
☐ Pull request comment added (restricted)	☐ Pull request comment added
☐ Pull request review approved	☐ Pull request review commented

Trigger setting

We now have a pipeline that satisfies our requirements. If you create pull requests on the source code repository, Codefresh automatically builds them.

When the pipeline has finished, **you can see in Docker Hub**[65] (or whichever container registry you're using), the resulting images are tagged with the same Git hash as the source code branch:


kostiscodefresh/my-preview-app

By kostiscodefresh · Updated 1 minute ago

TAG
d1cd602
Last pushed a few seconds ago by kostiscodefresh

```
docker pull kostiscodefresh/my-preview-app:d1cd602
```

Digest	OS/ARCH	Compressed Size
f5cd3e8ccf19	linux/amd64	7.98 MB

TAG
394176e
Last pushed 3 minutes ago by kostiscodefresh

```
docker pull kostiscodefresh/my-preview-app:394176e
```

Digest	OS/ARCH	Compressed Size
f6a0feca2ab9	linux/amd64	7.98 MB

Docker Hub tags

This concludes the setup for the Continuous Integration part. Let's move on to the deployment process.

Performing the initial deployment

With the CI part in place, we can create our pull request Generator. **Here's the initial version**[66].

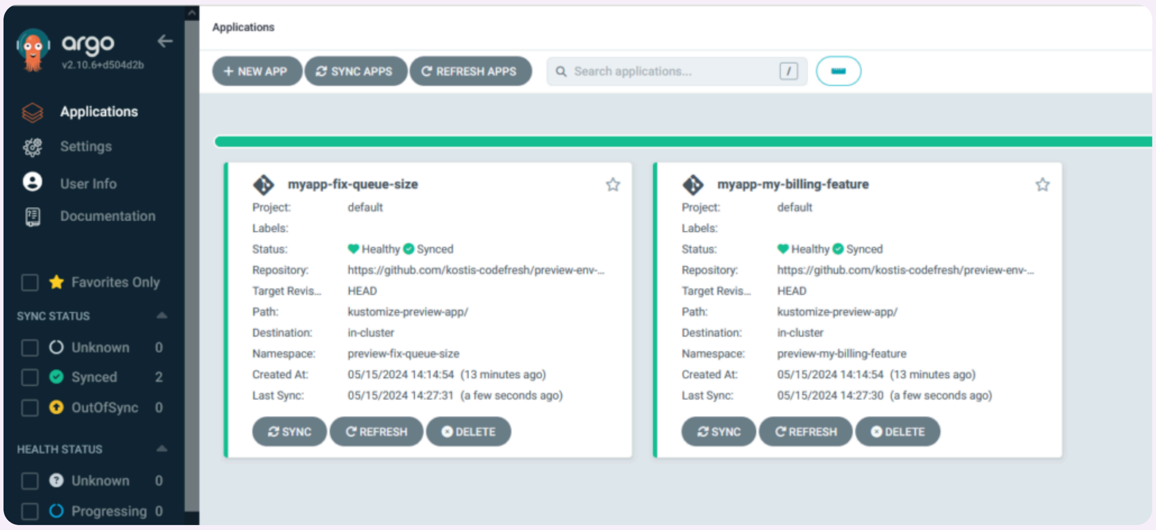
This Pull Request generator does the following:

1. Monitors the **kostis-codefresh/preview-env-source code**[61] repository for pull requests.
2. If it finds a pull request, it creates an Argo CD application named `myapp-<name of branch>`.
3. The Argo CD application syncs manifests found in the **kustomize-preview-app**[67] directory in the `preview-env-manifests` repo.
4. It deploys the manifests to the local cluster where Argo CD is installed, to a namespace with the same name as the branch of the pull request prepended with "preview".
5. It sets the container tag with the same hash as the source code, so that it matches the CI pipelines of Codefresh described in the previous section.

You can test this Pull Request generator by applying it to your local Argo CD instance.

```
kubectl apply -f pr-generator.yml -n argocd
```

Then, you can start creating pull requests on your source code repository and see applications created automatically. Isn't this great?



myapp-fix-queue-size

Project:	default
Labels:	
Status:	Healthy Synced
Repository:	https://github.com/kostis-codefresh/preview-env-...
Target Revis...	HEAD
Path:	kustomize-preview-app/
Destination:	in-cluster
Namespace:	preview-fix-queue-size
Created At:	05/15/2024 14:14:54 (13 minutes ago)
Last Sync:	05/15/2024 14:27:31 (a few seconds ago)

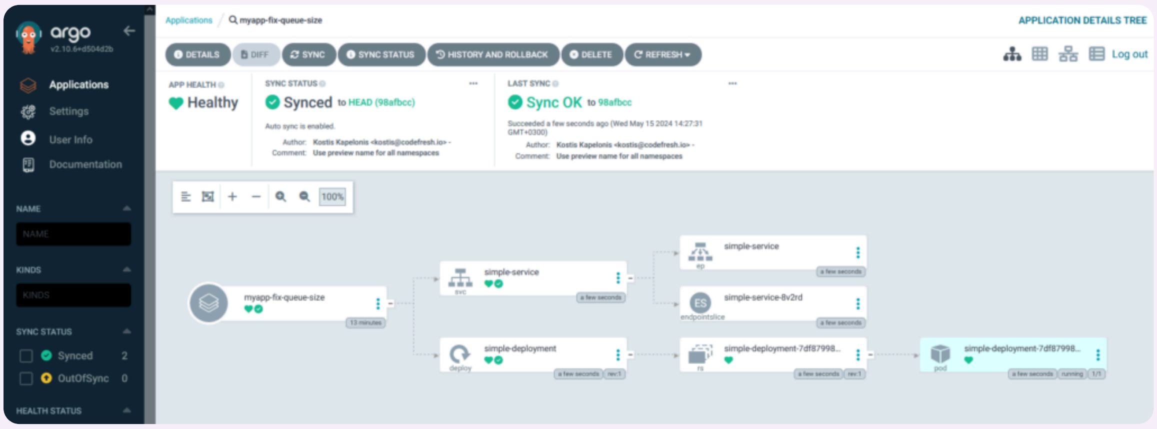
myapp-my-billing-feature

Project:	default
Labels:	
Status:	Healthy Synced
Repository:	https://github.com/kostis-codefresh/preview-env-...
Target Revis...	HEAD
Path:	kustomize-preview-app/
Destination:	in-cluster
Namespace:	preview-my-billing-feature
Created At:	05/15/2024 14:14:54 (13 minutes ago)
Last Sync:	05/15/2024 14:27:30 (a few seconds ago)

Preview apps in Argo CD

In our example, we instructed Argo CD to check for new pull requests every 3 minutes via the `requeueAfterSeconds` property. You need to fine-tune this property according to your organization's needs.

As always, you can click on any application to see its Kubernetes resources.



APP HEALTH
Healthy

SYNC STATUS
Synced to HEAD (98afbcc)
Auto sync is enabled.
Author: Kostis Kapelonis <kostis@codefresh.io> -
Comment: Use preview name for all namespaces

LAST SYNC
Sync OK to 98afbcc
Succeeded a few seconds ago (Wed May 15 2024 14:27:31 GMT+0300)
Author: Kostis Kapelonis <kostis@codefresh.io> -
Comment: Use preview name for all namespaces

Kubernetes resources

That's it!

Every time a developer creates a new pull request, their container image gets deployed automatically in the same namespace as the branch name. You can also inspect the applications manually in the cluster with the CLI.

```
➜  argocd git:(main) kubectl get ns
NAME                       STATUS   AGE
argocd                     Active   26d
default                    Active   26d
kube-node-lease            Active   26d
kube-public                Active   26d
kube-system                Active   26d
preview-fix-queue-size     Active   2m14s
preview-my-billing-feature Active   2m14s
```

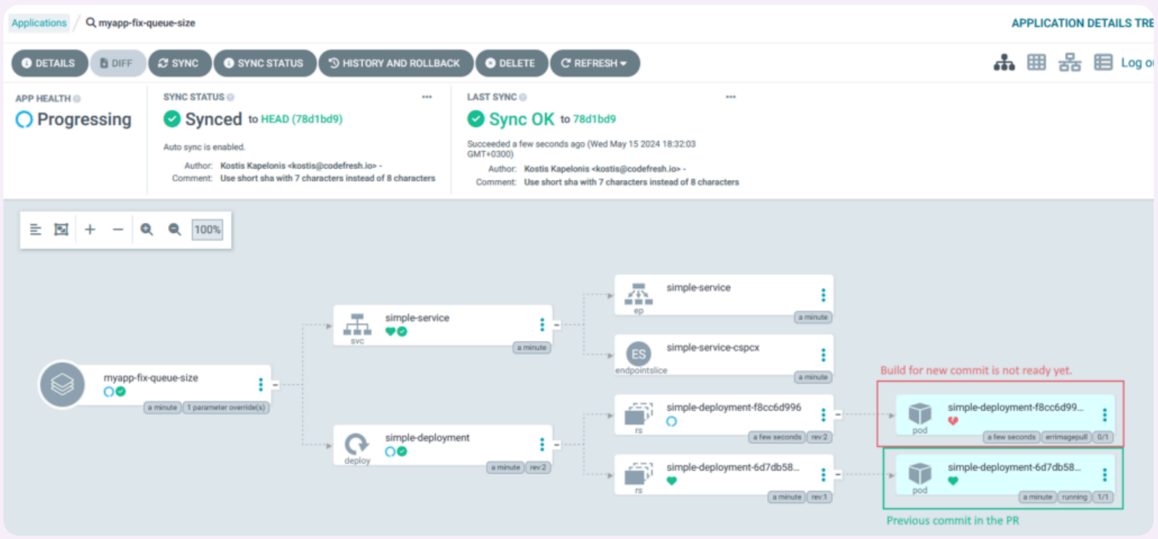
We use the "preview" prefix for all namespaces to make things easy. However, you can choose any naming convention for your organization.

Handling new commits in pull requests

Often, a developer notices something in a preview environment, fixes it, and wants to perform a redeployment. With our setup, this happens automatically. If the developer commits again, the Pull Request generator detects the commit and redeploys the application.

Note that we set the refresh period to 3 minutes in our example. Our application is straightforward, so typically when the Pull Request generator redeploys an environment, the container image is already built and pushed to our registry.

There's also the scenario where the image build is not yet complete. Argo CD will mark the deployment with an ImagePullBackOff error. Kubernetes will automatically retry the pull later, and the application will eventually succeed. However, note that this will work only if your application takes less than 5 minutes to build, as this is the default timeout for image pull operations.



Applications
myapp-fix-queue-size
APPLICATION DETAILS TRE
DETAILS
DIFF
SYNC
SYNC STATUS
HISTORY AND ROLLBACK
DELETE
REFRESH
APP HEALTH
Progressing
SYNC STATUS
Synced to HEAD (78d1bd9)
Auto sync is enabled.
Author: Kostis Kapelonis <kostis@codefresh.io> -
Comment: Use short sha with 7 characters instead of 8 characters
LAST SYNC
Sync OK to 78d1bd9
Succeeded a few seconds ago (Wed May 15 2024 18:32:03 GMT+0300)
Author: Kostis Kapelonis <kostis@codefresh.io> -
Comment: Use short sha with 7 characters instead of 8 characters
100%
myapp-fix-queue-size
simple-service
simple-deployment
simple-service
simple-service-cspcx
simple-deployment-f8cc6d996
simple-deployment-6d7db58...
Build for new commit is not ready yet.
simple-deployment-f8cc6d99...
errimagepull
simple-deployment-6d7db58...
running
Previous commit in the PR

New image is not built

If this is an issue for your developers, you can simply instruct the Pull Request generator to only deploy pull requests with a special label, for example "preview-ready". Then you have your CI system or developers add that label on the pull request after the CI build has finished. This way you guarantee the image is always there when Argo CD creates the temporary environment.

Destroying the temporary Argo CD application

Destroying an environment is straightforward. After you merge or reject/close a pull request, Argo CD detects it and automatically discards the respective environment.

Note that in our simple example, the preview namespace stays behind. To delete the namespace, you need to add a namespace resource in your Helm chart or Kustomize folder. Alternatively, you can set up a job that periodically removes unused namespaces.

Passing the name of the branch to a preview environment

Our setup works great for several common scenarios. One key point is that the application doesn't know if it's running in a preview environment.

Sometimes, however, you want to pass information about the pull request environment to the application, usually as a parameter with the name of the branch or the Git hash. This information can make the application adapt to the preview environment. An example scenario would be to set up an Ingress with the URL that matches the pull request. So in our example, we'd have URLs like this:

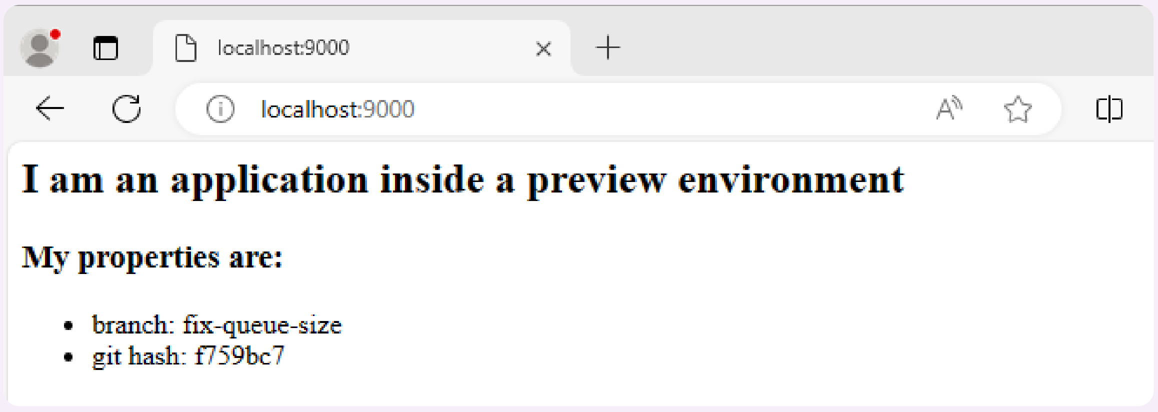
- **example.com/fix-queue-size**
- **example.com/my-billing-app**

Our example application source code already **accepts the Git hash and the branch name as environment variables**[68].

We can modify our Pull Request generator to pass these parameters in the created application.

```yaml
...
patches:
  - target:
      kind: Deployment
      name: simple-deployment
    patch: |-
      - op: replace
        path: /spec/template/spec/containers/0/env/0
        value:
          name: GIT_BRANCH
          value: '{{.branch_slug}}'
      - op: replace
        path: /spec/template/spec/containers/0/env/1
        value:
          name: GIT_HASH
          value: '{{.head_short_sha_7}}'
...
```

Now, when you create a preview application, if you visit the application, you see that it knows which branch and hash it was created from.



I am an application inside a preview environment

My properties are:

- branch: fix-queue-size
- git hash: f759bc7

Previewing app parameters

Note that we *do not* recommend this override syntax for static and/or production environments. It mixes Argo CD resources with information from the Kubernetes manifests. However, we recognize that some developers like this capability, so we include it in this guide for completeness.

Limitations and future considerations

The approach we've seen – monitoring the source code repository for pull requests – is very simple to set up, but there are some limitations:

1. By default, it deploys the new manifests without waiting for the container image to be ready. Kubernetes will automatically retry to pull it for the first 5 minutes only. So if your CI system needs more than 5 minutes to build (and test) the application, this approach won't work for you.
2. Developers can only create a preview environment for a single application (the one with the pull request). If your organization has adopted microservices, a developer might want to create a common preview environment with different applications from different Git repositories. However, this scenario is not possible with the approach in this guide.
3. There are some cases where the preview environment is not just a new container tag. Still, the developer wants to preview a configuration change, for example, introducing a new variable or configuration file. Again, this scenario is not possible with the approach in this guide.

You can solve all these limitations by changing the behavior of the Pull Request generator to monitor the Kubernetes manifest repository for pull requests. However, this process is more complex, so we'll cover it in a future article.

Help developers test their changes in isolation with Argo CD

In this section, you've seen how to use Argo CD even for preview environments. Using the **ApplicationSet**[69] Pull Request generator, you can create temporary/ephemeral deployments so developers can test a feature in isolation. Updating the environment with new code when the developer performs additional commits in the pull request is also easy to configure.

The benefits of GitOps[70] can also apply to preview environments. For example, it's now possible to quickly find configuration differences between environments or even revert to a preview version of the environment using just a Git action.

With the power of the ApplicationSet Generator, you can also automatically delete all inactive temporary environments. This helps with cloud costs and long-term maintenance of temporary environments.

The process is fully transparent for developers. They can continue creating pull requests in the source code repository. After they merge/reject the pull request, Argo CD discards the temporary environment without human intervention.

We hope you now have a good starting point for your preview environment strategy.

Conclusion

In this whitepaper, we have covered several day-2 operations for Argo CD. We have seen how:

- to use Argo CD with microservices
- to enhance your pull requests with additional diff information
- to help developers test their features in isolated environments.

If you need more help during your Argo CD journey, Codefresh has **comprehensive support services**[71], **TAM experts**[72], and the **world's first GitOps certification**[73]. We would be delighted to help you.

References

1. **https://github.com/argoproj/argo-cd/blob/master/manifests/install.yaml**
2. **https://argo-cd.readthedocs.io/en/stable/getting_started/**
3. **https://artifacthub.io/packages/helm/argo/argo-cd**
4. **https://argo-cd.readthedocs.io/en/stable/operator-manual/upgrading/overview/**
5. **https://developer.hashicorp.com/terraform**
6. **https://www.pulumi.com/**
7. **https://www.crossplane.io/**
8. **https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs**
9. **https://registry.terraform.io/providers/hashicorp/helm/latest/docs**
10. **https://registry.terraform.io/providers/argoproj-labs/argocd/latest/docs**
11. **https://codefresh.io/blog/how-to-structure-your-argo-cd-repositories-using-application-sets/**
12. **https://argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/#manage-argo-cd-using-argo-cd**
13. **https://argocd-autopilot.readthedocs.io/en/stable/**
14. **https://github.com/argoproj-labs/argocd-autopilot**
15. **https://slack.cncf.io/**
16. **https://codefresh.io/blog/scaling-argo-cd-securely-in-2024/**
17. **https://codefresh.io/codefresh-signup/**
18. **https://github.com/argoproj/argo-cd/issues/7437**
19. **https://argo-cd.readthedocs.io/en/stable/operator-manual/cluster-bootstrapping/**
20. **https://argo-cd.readthedocs.io/en/stable/user-guide/sync-waves/**
21. **https://argo-cd.readthedocs.io/en/stable/operator-manual/health/**
22. **https://argo-cd.readthedocs.io/en/stable/operator-manual/health/#argocd-app**
23. **https://github.com/argoproj/argo-cd/issues/3781**

24. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
25. https://github.com/kostis-codefresh/application-dependency-argocd
26. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/apps/example-frontend.yml
27. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/apps/example-api.yml
28. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/apps/example-db.yml
29. https://github.com/kostis-codefresh/application-dependency-argocd/tree/main/apps
30. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/all-apps.yml
31. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/patch-argocd-cm..yml
32. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/manifests/frontend/deployment.yml
33. https://github.com/kostis-codefresh/application-dependency-argocd/blob/main/manifests/api/deployment.yml
34. https://argocd-applicationset.readthedocs.io/en/stable/
35. https://github.com/argoproj/argo-cd/issues/3517
36. https://github.com/argoproj/applicationset/issues/221
37. https://github.com/kostis-codefresh/argocd-preview-diff/blob/no-context-pr/envs/prod-us/replicas.yml
38. https://argo-cd.readthedocs.io/en/stable/user-guide/auto_sync/
39. https://www.cncf.io/blog/2020/12/17/solving-configuration-drift-using-gitops-with-argo-cd/
40. https://argo-cd.readthedocs.io/en/stable/user-guide/commands/argocd_app_diff/
41. https://github.com/kostis-codefresh/rendered-manifests/pull/1/files
42. https://developer.hashicorp.com/terraform/language/syntax/configuration
43. https://developer.hashicorp.com/terraform/cli/commands/apply

44. **https://developer.hashicorp.com/terraform/cli/commands/plan**
45. **https://www.runatlantis.io/**
46. **https://www.runatlantis.io/docs/provider-credentials.html**
47. **https://developer.hashicorp.com/terraform/language/state**
48. **https://www.runatlantis.io/docs/security.html**
49. **https://codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/**
50. **https://github.com/kostis-codefresh/manifest-refactoring/blob/main/envs/qa/application.properties**
51. **https://github.com/kostis-codefresh/manifest-refactoring/blob/main/envs/staging/application.properties**
52. **https://github.com/kostis-codefresh/manifest-refactoring/blob/refactoring-of-settings/variants/non-prod/application.properties**
53. **https://github.com/kostis-codefresh/manifest-refactoring/pull/1**
54. **https://github.com/kostis-codefresh/manifest-refactoring/pull/1/files**
55. **https://codefresh.io/blog/kubernetes-antipatterns-2/**
56. **https://codefresh.io/blog/unlimited-preview-environments/**
57. **https://argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/Generators-Pull-Request/**
58. **https://codefresh.io/blog/argo-cd-best-practices/**
59. **https://www.vcluster.com/**
60. **https://argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/#applications**
61. **https://github.com/kostis-codefresh/preview-env-source-code**
62. **https://github.com/kostis-codefresh/preview-env-manifests**
63. **https://piotrminkowski.com/2023/06/19/preview-environments-on-kubernetes-with-argocd/**
64. **https://codefresh.io/docs/docs/pipelines/triggers/git-triggers/**
65. **https://hub.docker.com/r/kostiscodefresh/my-preview-app/tags**
66. **https://github.com/kostis-codefresh/preview-env-manifests/blob/main/argocd/pr-generator.yml**

67. https://github.com/kostis-codefresh/preview-env-manifests/tree/main/kustomize-preview-app
68. https://github.com/kostis-codefresh/preview-env-source-code/blob/main/main.go#L21
69. https://codefresh.io/learn/argo-cd/argocd-applicationset-multi-cluster-deployment-made-easy-with-code-examples/
70. https://codefresh.io/blog/gitops-benefits-and-considerations/
71. https://codefresh.io/enterprise-argo-support/
72. https://codefresh.io/argo-technical-account-management/
73. https://learning.codefresh.io/

Further reading

If you enjoyed this, you might also enjoy some of these articles on Argo CD:

1. https://codefresh.io/blog/argo-cd-anti-patterns-for-gitops/
2. https://codefresh.io/blog/how-to-structure-your-argo-cd-repositories-using-application-sets/
3. https://codefresh.io/blog/argocd-clusters-labels-with-apps/