Flux CD
D2 Reference Architecture Guide

Rationale and security considerations for the adoption of Flux CD D2 reference architecture on Kubernetes

Table of Contents

Executive Summary

Flux CD, a cloud native Continuous Deployment tool, is a crucial component in modern GitOps pipelines. In this guide, we delve into the intricacies of deploying a Flux CD reference architecture specifically designed to achieve a git-less GitOps installation.

This document extends the Flux CD Reference Architecture 1 (D1). The revision is intended for those with some previous Flux knowledge, especially regarding the complexities behind multi-tenant setups. The D1 architecture document equips the reader with the general knowledge to understand these complexities, including a technical explanation of Kubernetes manifest organisation in different git repositories, a definition of the main user personas and roles involved, and considerations on security requirements.

These fundamental concepts remained unchanged between D1 and D2 and are assumed to be shared knowledge. This D2 documentation delves into low-level technical improvements, setup simplification, and the advantages brought by reconciling configuration using Open Container Initiative (OCI) Artifacts instead of git repositories.

Finally, readers will find additional information about the subscription-based ControlPlane Enterprise for Flux, which supports the full-time employment of core maintainers to work on the upstream Flux CD project, and an appendix containing solutions to the most common administration tasks.

Introduction and Objective

In this iteration of our reference architecture, we introduce the concept of *Gitless GitOps*: the desired state of the system is driven by OCI Artifacts stored in container registries that are built from manifests tracked in git repositories during the Continuous Integration process. Continuous synchronisation of desired and actual system state is achieved through automated workflows and reconciliation mechanisms.

OCI is the standard for containers, and so Flux's OCI Artifact reconciliation benefits from consistent tooling and security mechanisms already used by the applications deployed into a Kubernetes cluster. This unifies the technologies and practices around software distribution, reduces control duplication, and simplifies production system architecture.

After completing this guide, readers will clearly understand how to implement a Flux CD multi-tenant reference architecture managed through OCI Artifacts. They will be equipped with the knowledge and best practices to:

- Set up and manage secure, multi-tenant Kubernetes clusters using Flux Operator, without any additional third-party components
- Use CI pipelines to securely build and sign OCI Artifacts and push them to container registries
- Learn how a git-less GitOps workflow can be deployed using Flux to harden production environments

The reference architecture detailed within the next sections can be deployed with either Enterprise for Flux CD or the open source version of Flux, as both are guaranteed to have feature parity.

Background

OCI Artifacts and Container Registries

The D1 Architecture reconciles manifests hosted on git repositories. In contrast, D2 reconciles manifests stored in a container registry as OCI Artifacts. The OCI Image Specification allows OCI registries to store general-purpose non-image objects, so-called *OCI Artifacts/Objects*. The OCI Distribution specification and HOCI Artifact Guide provide further technical information on the composition and use of OCI Artifacts.

Using OCI Artifacts as configuration instead of git repositories brings multiple security advantages, notably an improvement to access control by leveraging on workload identities, and avoiding direct interaction between Kubernetes clusters and the manifests. OCI Artifacts support rich additional metadata, including signature, SBOMs and vulnerability scans, which enhance the fidelity of control and verification available to Flux at deployment time. We explain these improvements, and how Flux utilises them to enhance platform security, in the following paragraphs.

Flux references OCI Registries using the `OCIRepository` Custom Resource, both from a `Kustomization` CR and from a `HelmRelease` CR.

Signature and verification

Flux `OCIRepository` CRs can be configured to verify signed OCI Artifacts before reconciling them.

```
Unset
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: OCIRepository
metadata:
 name: podinfo
 namespace: flux-system
spec:
 interval: 5m
 url: oci://ghcr.io/stefanprodan/manifests/podinfo
 ref:
   semver: "*"
 verify:
   provider: cosign
```

```
    secretRef:
      name: cosign-pub
```

In this example, Flux uses *cosign* (with a public key `cosign-pub`) to verify OCI Artifacts coming from the `podinfo` OCI Repository. It is assumed the signing process is being executed during the CI phase by a trusted and hardened build system.

Flux can also verify `GitRepository` sources, using the public keys of the Git authors used to sign commits. Commit signing keys are usually linked to an individual developer, which implies further evaluation of the signing key hierarchy is needed.

There might be mainly two options:

- Developer signing keys are generated from a single master key, managed by the organization. The respective public key is added to the `verify.secretRef` field.
- Each developer generates their own key. Any new developer's public key must be added to the verification configuration.

Respectively, in case of key revocation or invalidation:

- If keys have been generated from a master key, invalidation of a single author subkey would not be possible unless a denylist system is used. Invalidating the master public key would invalidate all the other keys.
- If all keys are managed independently, invalidating a single key involves removing it from the list of trusted keys.

Signed OCI Artifacts benefit from decoupling the manifest signature verification process from source authenticity verification — the latter can then be performed by a properly hardened CI workflow. Flux is then only responsible for verifying whether artifacts are generated and signed by the CI system.

SBOM

A Software Bill Of Materials (`SBOM`) is a *"formal record containing the details and supply chain relationships of various components used in building software"* (NIST). Organisation-wide SBOM usage allows an organization to quickly understand which components are affected by a security advisory. SBOMs can be uploaded on OCI registries as OCI Reference Artifacts. Signatures should also be attached to SBOMs in this manner for provenance and veracity.

VEX

The Vulnerability Exploitability eXchange (`VEX`) document format *"allows a software supplier or other parties to assert the status of specific vulnerabilities in a particular product"*. A vendor can attach VEX documents to OCI Artifacts to inform consumers that a specific vulnerability is known, and how it is mitigated or managed. VEX documents can be automatically discovered from OCI Registries during the vulnerability scanning process, and signatures should be attached to VEXs for validation as per SBOMs.

Workload Identities

Container registry can authenticate workloads using "workload identities" — cryptographic authentication for machines without persistent secrets. This improves overall security and maintenance effort during CI (building OCI Artifacts) and deployment phases (downloading the OCI Artifacts), as no secrets are created, maintained (i.e. rotated or revoked), or stored in CI or in the cluster.

CI

As shown later in this document, Flux D2 GitHub "push" actions use workload identities behind the scenes. The action is configured to upload OCI Artifacts on GHCR, with no need for additional pipeline secrets that risk leak or compromise. This applies in cases where Artifacts are stored in the same registry path associated with the repo as the action workload is executed from (i.e. the `push-action` stored in *d2-infra* repository will be called in the context of the *d2-infra* repository, therefore the OCI Artifacts built can be pushed only on GHCR related to the *d2-infra* repository).

Other configurations require the use of a classic Personal Access Token (PAT), to be stored as a pipeline secret. This is not recommended as it is prone to unintended exposure.

A GitHub action run can be identified with the following contextual field:

- `{{github.actor}}`: *The username of the user that triggered the initial workflow run*
- `{{github.ref_name}}`: *The short ref name of the branch or tag that triggered the workflow run. This value matches the branch or tag name shown on GitHub*
- `{{github.repository}}`: *The owner and repository name*

Some contextual fields can also be made available to the OIDC token. This is useful for complex use-cases, signing and verifying signatures, or fine tuning registry access for supported registries.

For the sake of simplicity, D2 focuses on GitHub only. However, the same principles can be applied to GitLab (and other CI Platforms), using GitLab CI/CD to authenticate to the GitLab container registry. Examine your provider's documentation on CI and on registry authentication to determine their support for secret-less authentication to a container registry.

Manifest download

Flux `source-controller` can use a specific service account to connect to the container registry where OCI Artifacts are being stored (if supported by the environment and registry). That service account might be able to impersonate a cloud identity with grants on the registry, thus ensuring a secret-less authentication.

GHCR does not support workload identities for manifest downloads unless the identities are part of a GitHub action CI run. Therefore, Flux can only use classic PAT to login to GHCR. If you want your cluster to authenticate to a container registry using native workload identity, you might consider keeping the OCI Artifacts as close to the cluster as possible. When deploying a cluster in AWS, using Amazon ECR will guarantee support for workload identities.

The same principles can be applied to:

- AKS with ACR
- GKE with Google Artifact Registry

Complex use cases and RBAC fine-tuning

Workload identity federation supports using different cloud providers to store artifacts and run Kubernetes clusters. A cluster running on one cloud provider can authenticate to a container registry hosted in another cloud provider.

This can be applied to CI pipelines, git repositories, and container registries. For example, a GitHub Action can push OCI Artifacts to Amazon ECR. AWS must trust GitHub's OIDC as a trusted federated identity.

Access granularity depends on the container registry being used. For example, Amazon ECR can use an IAM role per repository to limit a GitHub action. This enables a repository and reference to be restricted to a specific repository on the container registry.

For example, one could only run in the `main` branch to write to the `latest-stable` tag. This prevents tampering attempts on any branch other than the `main` branch, which will result in the repo rejecting OCI artifacts.

Fine-tuning container registry RBAC models may increase the number of OCI Artifacts and/or repositories, so the granularity of control is proportional to the day two management effort.

Flux Operator

The `Flux Operator` is a Kubernetes CRD controller that enhances the functionality of CNCF Flux CD and the ControlPlane Enterprise distribution. It extends Flux with lifecycle and operational management capabilities, enhanced preventative security controls, developer self-service enablement, and automated pull request preview environments for GitLab and GitHub workflow verification.

This documentation describes only the operator features used in D2. The GitHub repository and documentation expand upon the comprehensive list of features:

- Autopilot for Flux CD
- Advanced Configuration
- Deep Insights
- Self-Service Environments
- Enterprise Support

FluxInstance CR

D2 relies on clusters bootstrapped with the `FluxInstance` CR, an upgrade to the `flux bootstrap` command. `FluxInstance` is a declarative API for the installation, configuration and automatic upgrade of the Flux distribution, and assists in configuring Flux CD with complex multi-tenancy functionality.

multitenant flag

Multitenancy must be manually enabled in Flux D1 using patches defined in the `Kustomize` overlay. This configuration can be enabled with a single flag (`.spec.cluster.multitenant: true`) defined in the `FluxInstance` CR. Further information is available in the documentation.

copyFrom annotation

D1 reference architecture utilised Kyverno policies to clone `Secrets` and runtime information across namespaces to avoid using cross-namespace references. Flux Operator is able to generate `ConfigMaps` and `Secrets`, as defined in the `ResourceSet` CR, with data copied from existing objects, using only metadata defined in the `fluxcd.controlplane.io/copyFrom` annotation.

```
Unset
...
spec:
  inputs:
    - tenant: "team1"
    - tenant: "team2"
  resources:
   - apiVersion: v1
     kind: ConfigMap
     metadata:
       name: runtime-info
       namespace: << inputs.tenant >>
       annotations:
         fluxcd.controlplane.io/copyFrom: "flux-system/runtime-info"
...
```

Additional details on this functionality can be found in the ResourceSet documentation.

Application tenants can access OCI Artifacts stored in GHCR via a `Secret` stored in the application (i.e. backend/frontend) namespaces. The D2 model requires the Platform team to create these `Secrets` (via *flux create secret git* command) in the `flux-system` namespace, and await the Flux Operator sync to the tenant's application namespaces. The `Secret` stores a single PAT with GHCR read permissions, only used to download the OCI Artifacts for reconciliation. This is a relevant difference between D2 and D1 - D1 uses a single fine-grained PAT for both reconciliation and update automation.

The `flux-runtime-info` `ConfigMap` can be used by `Kustomization`, and therefore by workloads (infra or application). D2 multitenancy lockdown does not permit Flux controllers to make cross-namespace references. As with PAT `Secrets`, the Flux Operator duplicates the `ConfigMap` to all namespaces, making the content of the `ConfigMap` potentially available to every workload managed by `ResourceSets`. As per Kubernetes best practice, no confidential information should be stored in a `ConfigMap`.

ResourceSet CR

D2 reference architecture relies on a new `ResourceSet` Custom Resource part of the Flux Operator API. `ResourceSet` is a declarative API for generating a group of Kubernetes objects based on a matrix of input values and a set of templated resources. This `ResourceSet` API integrates with GitLab and GitHub pull requests to create ephemeral environments for testing and validation.

Using the new `ResourceSet` APIs drastically reduces this architecture's lines of code compared to D1. It defines and utilises reusable templates as most components are reconciled from a multitenant locked-down Flux installation that uses the same Kubernetes object templates.

In an application context, these resources should be templated, parametrized and deployed as a single unit in a self-service environment:

- `Namespace` `<< inputs.tenant >>`
- `ConfigMap` `flux-runtime-info`: for the runtime cluster information that Flux is going to use to customize the application installation (e.g. the Cluster domain when setting up an Ingress)
- `OCIRepository` `apps`: for Flux to reference the OCI Artifact
- `Kustomization` `apps`: for Flux to start reconciling the application
- `ServiceAccount` `flux`: which Flux uses to manage the application and to access supported registries
- `Secret` `ghcr-auth`: for Flux to download the referenced OCI Artifact from a private registry, assigned as `.spec.imagePullSecrets` to `flux` `ServiceAccount`
- `RoleBinding` `flux`: to assign the `ClusterRole` `admin` grant to the `flux` `ServiceAccount`. The grant is bound to the namespace where the application is deployed to

Inputs configuration

An input value is a key-value pair of strings and structs, where the key is the input name which can be referenced in the resource templates using `<< inputs.name >>` syntax. Once the template has been defined, inputs are then used to generate the final objects to be applied to the cluster. Inputs are defined in `.spec` of the `ResourceSet`.

Static Inputs

Inputs are hardcoded in the `ResourceSet` `.spec.inputs`.

```
Unset
spec:
  inputs:
    - tenant: "frontend"
      tag: "${ARTIFACT_TAG}"
      environment: "${ENVIRONMENT}"
    - tenant: "backend"
      tag: "${ARTIFACT_TAG}"
      environment: "${ENVIRONMENT}"
```

As with any other Kubernetes resources managed by Flux, some values can be replaced at runtime while the Kustomization is being applied. In this case, `${ARTIFACT_TAG}` and `${ENVIRONMENT}` will be replaced with the data stored in the `flux-runtime-info` `ConfigMap`, as specified in the `tenants` `Kustomization` CR `.spec.substituteFrom` field.

Dynamic Inputs

Inputs are automatically fetched by the operator at runtime from the `ResourceSetInputProvider` CRs:

```
Unset
spec:
  inputsFrom:
    - apiVersion: fluxcd.controlplane.io/v1
      kind: ResourceSetInputsProvider
      name: podinfo-pull-requests
```

At runtime, the operator fetches the input values when the `ResourceSetInputProvider` reconciler detects a change in the upstream source.

Low Level Design Description

D2 Architecture Repositories Structure

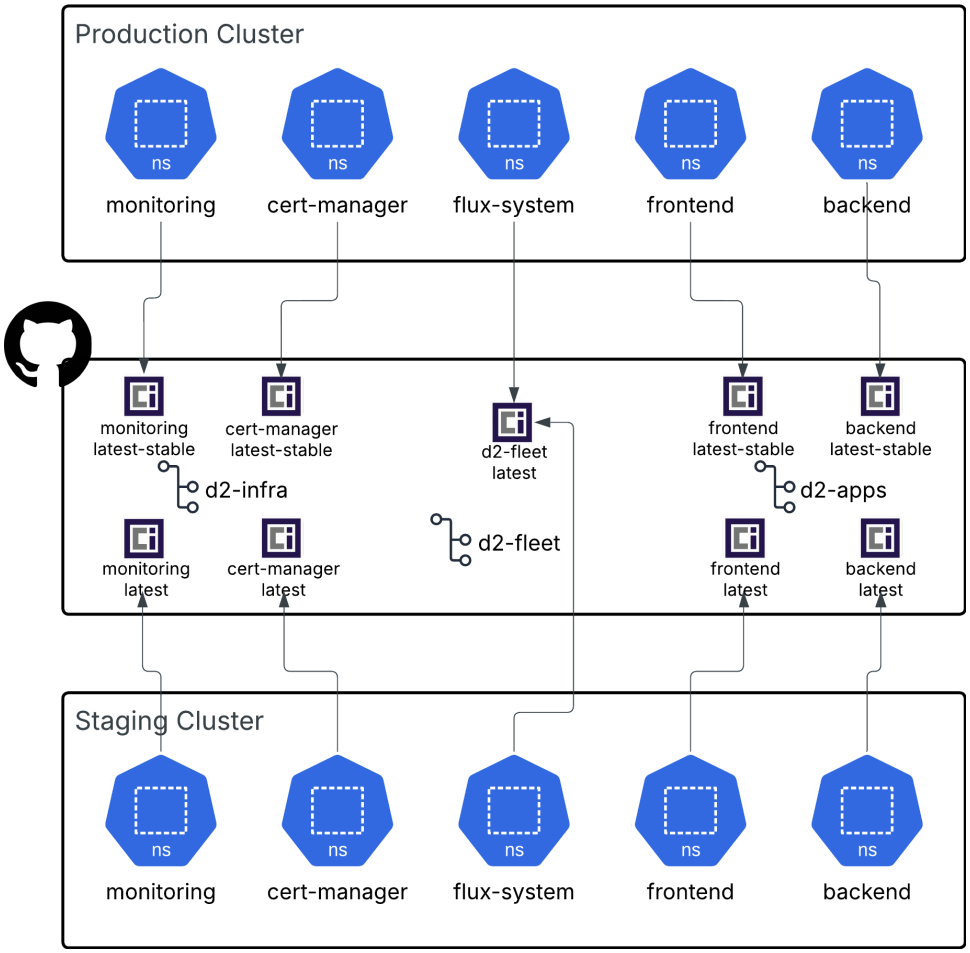

Figure 1: D2 Architecture - repository structure

D2 employs the following three repositories, in the same structure as D1.

d2-fleet Repository

Utilised by the Platform team only, who are admins on all clusters, in order to:

- Reconcile Flux Operator and a new `FluxInstance` with multi-tenancy restrictions on fleet clusters
- Configure the delivery of platform components (defined in *d2-infra* repository)
- Configure the delivery of applications (defined in *d2-apps* repository)

d2-infra Repository

This repository is managed by the Platform team, responsible for the Kubernetes infrastructure. This repository is used to define Kubernetes infrastructure components such as:

- Cluster add-ons (CRD controllers, admission controllers, monitoring, logging, etc.)
- Cluster-wide definitions (`Namespace`, `IngressClass`, `StorageClass`, etc.)
- Pod security standards
- Network policies

In larger organisations and operational models, where dedicated teams may be responsible for services (e.g., a security team managing policies, a secrets team managing secrets operators, or an observability team managing Prometheus), these teams will configure their components within this repository.

d2-apps Repository

Each Application team responsible for the delivery of an application running on the Kubernetes cluster fleet will be allocated an Application Repository, hosting application components such as:

- Flux `HelmRepository/OCIRepository` CR pointing to the application Helm charts in container registries
- Flux `HelmRelease` CR for the applications
- Per-environment Kustomize overlays

This repository is reconciled on the cluster fleet by Flux as the namespace admin, and can't contain Kubernetes cluster-wide definitions such as CRDs, `ClusterRole`s, `Namespace`s, etc.

Access to this repository is restricted to the dev teams and the Flux bot account. The Flux bot must be the only account with direct push access to the *main* branch.

OCI Artifacts and CI

Path and tags

The D2 architecture git repositories are packaged and distributed as OCI Artifacts, signed and published to GitHub Container Registry using the `push-artifact` and `release-artifact` GitHub Actions workflow.

Artifact	Git Repository	Environments
oci://<base_path>/d2-fleet	d2-fleet	Staging / Production / GHA image update automation cluster
oci://<base_path>/d2-infra/cert-manager	d2-infra	Staging / Production
oci://<base_path>/d2-apps/backend	d2-apps	Staging / Production
oci://<base_path>/d2-apps/frontend	d2-apps	Staging / Production

`<base_path>` is ghcr.io/controlplaneio-fluxcd.
Staging and Production environments are going to be reconciled from different OCI tags, respectively:

- *latest* for Staging
- *latest-stable* for Production

CI

Manifests are packaged and uploaded on artifact repositories using a CI workflow.

push-artifact

OCI Artifacts are currently being built during the CI phase using GitHub Actions. A custom action `push-artifact` will perform the following steps:

- Check out the code
- Install cosign
- Install flux CLI
- Login on GHCR using the implicit GitHub secret
- Prepare the tags
- Package and push the involved components (using the input matrix input matrix) as OCI Artifacts on the registry, using Push Flux Artifact action
- Sign the artifact with cosign, without using any explicit key

The action will need the following permissions:

```
Unset
    permissions:
      contents: read
      packages: write # for pushing
      id-token: write # for signing
```

This action is cloned across all repositories, with a slightly different input matrix.

release-artifact

A different action `release-artifact`, will only trigger when a tag is pushed. The action will build the specific OCI Artifact associated with that tag (i.e. `monitoring/v0.0.2`), then push it to the registry, and add the `latest-stable` tag (i.e. `ghcr.io/controlplaneio-fluxcd/d2-infra/monitoring:latest-stable`). This is particularly useful if we consider that production clusters will only reconcile from the `latest-stable` tag.

This action will use the same Push Flux Artifact GitHub action as `push-artifact` behind the scenes.

Keyless signing

These actions will sign the Artifact. No key will be explicitly used during the signing process, as the `id-token` permission will enable the job to create a new token to impersonate this workflow. The provider will use two variables that are implicitly injected by the CI pipeline when that permission is added:

- `ACTIONS_ID_TOKEN_REQUEST_TOKEN`
- `ACTIONS_ID_TOKEN_REQUEST_URL`

The provider will then use these two variables to get a GitHub OIDC Token. Cosign will then use that token to sign the artifacts as that specific CI run.

Managing different Environments

In D1, different branches in all three repositories were used to reconcile different environments:

- *main* for Staging clusters
- *production* for Production clusters

D2 uses a different approach: one *main* git branch is used across all three repositories. This branch generates artifacts that are pushed to each tenant's respective registry paths (e.g., *d2-apps/backend*). Different environments are reconciled using a combination of tags and artifact paths (see OCI Artifacts).

`path` and `tag` information is stored in the `ConfigMap` `flux-runtime-info`, which stores cluster information. Using `Kustomize` overlays, the base manifests are enriched/modified according to the environment's needs (i.e. adding limits/requests for production deployments).

As the example apps tenant, a staging cluster will use artifacts with the `latest` tag to download the `OCIRepository` (`.spec.ref.tag`).

```
Unset
    - apiVersion: source.toolkit.fluxcd.io/v1beta2
      kind: OCIRepository
      metadata:
        name: apps
        namespace: << inputs.tenant >>
      spec:
        interval: 5m
        serviceAccountName: flux
        url: "oci://ghcr.io/controlplaneio-fluxcd/d2-apps/<< inputs.tenant >>"
        ref:
          tag: << inputs.tag >>
```

Once the artifact is downloaded, a `Kustomization` CR pointing to a specific path (`.spec.path`) `./staging` will be deployed.

```
Unset
    - apiVersion: kustomize.toolkit.fluxcd.io/v1
      kind: Kustomization
      metadata:
        name: apps
        namespace: << inputs.tenant >>
      spec:
        ...
        sourceRef:
          kind: OCIRepository
          name: apps
        path: "./<< inputs.environment >>"
```

`Kustomization` will then build all the manifests, considering the base manifests and additional overlays, if any.

Policies

D1 uses Kyverno to ensure `ConfigMaps` and `Secrets` are replicated across the needed namespaces, via `ClusterPolicy` CRs. Kyverno `controllers` and `configs` are deployed before any other components, to ensure policies are applied before any other reconciliation is started. This explicit dependency had to be implemented as part of the reconciliation flow, and failure in policy reconciliation would halt the reconciliation of all the other components in the cluster.

D2 reduces third-party dependencies, so these objects are now synced using Flux Operator features, removing the need for Kyverno.

D2 uses Common Expression Language (CEL) native support in Kubernetes API (> v1.30) to define additional rules, as shown in the policies tenant. With CEL, complex validation rules can be deployed on the cluster via the `ValidatingAdmissionPolicy` resource.

The rules are part of the `policies` `ResourceSet` template that employs the following resources:

- `ConfigMap` `flux-allowlist`: specifies a list of valid OCI repository addresses
- `ValidatingAdmissionPolicy` `source.policy.fluxcd.controlplane.io`: only allows `GitRepository`, `OCIRepository` and `HelmRepository` if their `spec.url` adheres to the allow list `flux-allowlist`
- `ValidatingAdmissionPolicyBinding` `flux-tenant-sources`: a previously declared policy will cause a “Deny” response if violated, and will only apply to objects with a target namespace with the following label `toolkit.fluxcd.io/role: tenant`

If Kyverno is being used already, note that complex CEL policies can also be deployed by Kyverno and may be preferable.

Update Automation

Flux enables continuous deployment through image automation controllers so new application versions can be deployed to any environment.

A dedicated kind cluster, `update-1`, is being used to manage manifest updates. This cluster is bootstrapped weekly in the runners by a specific GitHub Action `e2e-update`.

e2e-update GitHub Action

```
Unset
name: e2e-update
on:
  workflow_dispatch:
  schedule:
    - cron: 0 8 * * 1

jobs:
  image-update:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Setup Flux
      uses: controlplaneio-fluxcd/distribution/actions/setup@main
    - name: Setup Helm
      uses: fluxcd/pkg/actions/helm@main
    - name: Setup Kubernetes
      uses: helm/kind-action@main
    - name: Bootstrap Flux
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}       # GHCR access
        GH_UPDATE_TOKEN: ${{ secrets.GH_UPDATE_TOKEN }} # GitHub write access
      run: |
        make bootstrap-update
    - name: Run image automation
      run: |
...
```

Automation components should not be deployed to Production clusters or in any long-lasting cluster.

update-1 FluxInstance

As you can see from the `update-1` FluxInstance CR, the update cluster will be shipped with `image-reflector-controller` and `image-automation-controller`.

```
Unset
apiVersion: fluxcd.controlplane.io/v1
metadata:
  name: flux
  namespace: flux-system
spec:
  distribution:
    version: "2.x"
    registry: "ghcr.io/fluxcd"
    artifact:
"oci://ghcr.io/controlplaneio-fluxcd/flux-operator-manifests:latest"
  components:
    - source-controller
    - kustomize-controller
    - helm-controller
    - notification-controller
    - image-reflector-controller
    - image-automation-controller
```

These controllers work together to automatically push updated manifests on a specific `image-updates` branch.

Update flow

This update automation flow will then look like this:

1. Kind cluster is created weekly
2. Flux Operator installed into cluster
3. Secrets added for GitHub repos with read/write grants
4. Flux CD installed by Flux Operator through a `FluxInstance` CR

Then the following components are reconciled:

5. `ImageUpdateAutomation` CR: Git repository to write image updates to. The PAT associated with the `GitRepository` CR used by `ImageUpdateAutomation` must have enough privileges to create the `image-updates` branch (if not existing) and to push updates to it
6. `ImageRepository` CR: container registry to scan for new tags
7. `ImagePolicy` CR: semver range to use when filtering tags

On GitHub:

8. `image-updates` GitHub Action is fired when the `image-updates` branch is updated, and it opens a PR to `main`
9. The update cluster is destroyed
10. Once the branch is merged to main, the updated manifests are packaged and uploaded on the registry, tagged with `latest`
11. Tags can be used to promote the changes to production, firing the action that will upload the updated artifacts tagged as `latest-stable`

Flux will then notice a drift, download the new OCI Artifacts, and update the components.

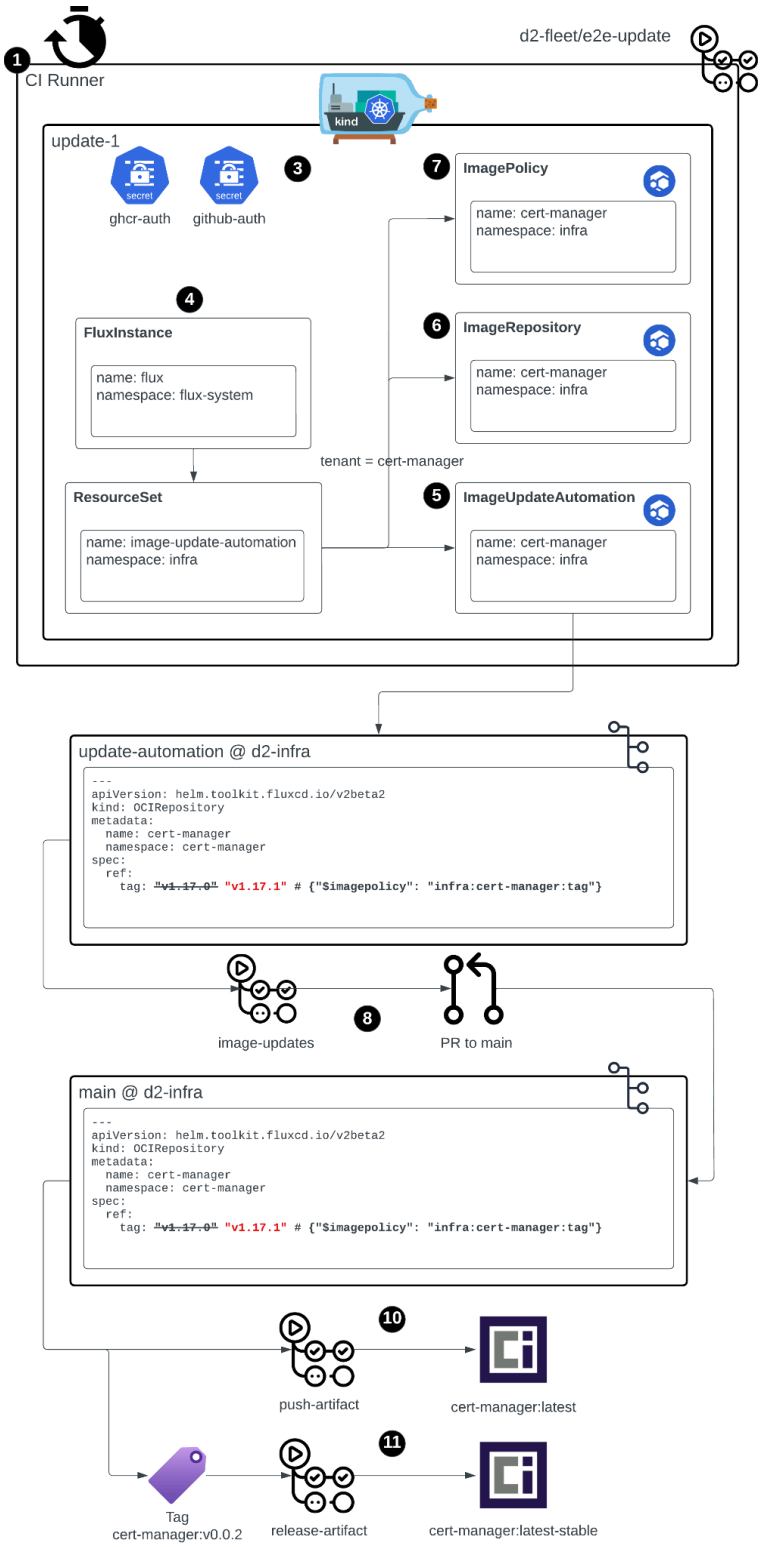

Figure 2: Update Automation

Example Workflows

Cluster Onboarding

The bootstrap procedure is a one-time operation to set up the Flux controllers on the cluster, and configure delivery of platform components and applications. For the sake of simplicity, the workflow focuses on onboarding a new cluster.

The bootstrap procedure requires the following:

- Flux CLI
- A Github Bot Account Classic Personal Access Token (PAT) with suitable permissions
- Cluster admin permissions on a provisioned Kubernetes cluster

The Flux CLI or Terraform Module can be called within a cluster provisioning pipeline or used from an administrator's machine. See the *flux bootstrap* command for further details, discussion on Flux CLI usage can be found in Flux CLI usage & Alternatives. The Terraform plan to install Flux Operator can be found in the *d2-fleet* repo.

The Terraform run will perform the following actions on the cluster:

1. Creates a Kubernetes `Secret` named `flux-system` in the flux-system namespace that contains the bot Classic PAT, to be able to download the first OCI Artifact from the registry
2. Installs Flux Operator on the cluster
3. Creates a multi-tenancy enabled `FluxInstance` pointed to the *d2-fleet* OCI Artifact with tag `latest` and path `clusters/staging`

From this point on, the Flux controllers will:

4. Reconcile the cluster state with the desired state within the fleet OCI Artifact
5. The tenant folder within the fleet repository contains `OCIRepository` CRs that point to *d2-infra* and *d2-apps* repositories as further sources, as well as bootstrap configuration (namespace/RBAC) for those tenants
6. Reconcile the cluster state with the desired state within the *d2-infra* and *d2-apps* OCI Artifacts

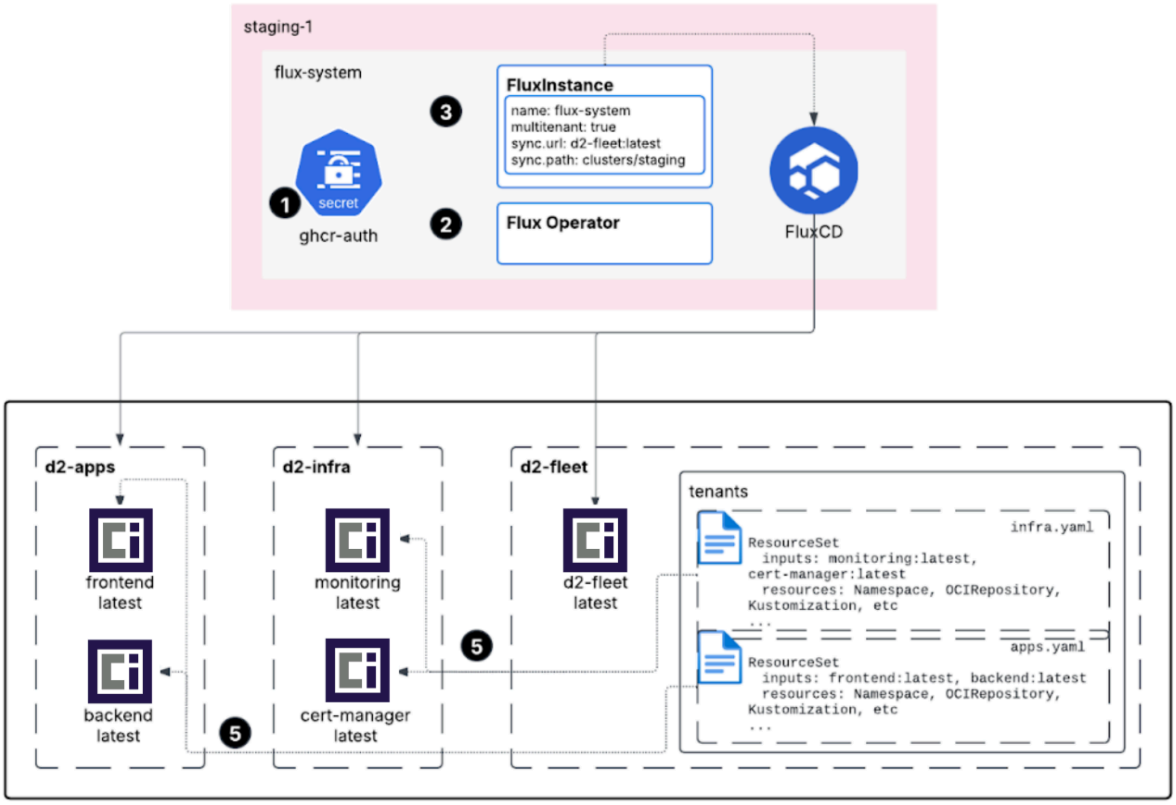

Figure 3: Example Workflow - Cluster onboarding

Adding a New Infra Component

The Platform team should follow this process to add new infrastructure components to the fleet. The diagram below shows where and when the manifests for a new component are pushed to the different repositories.

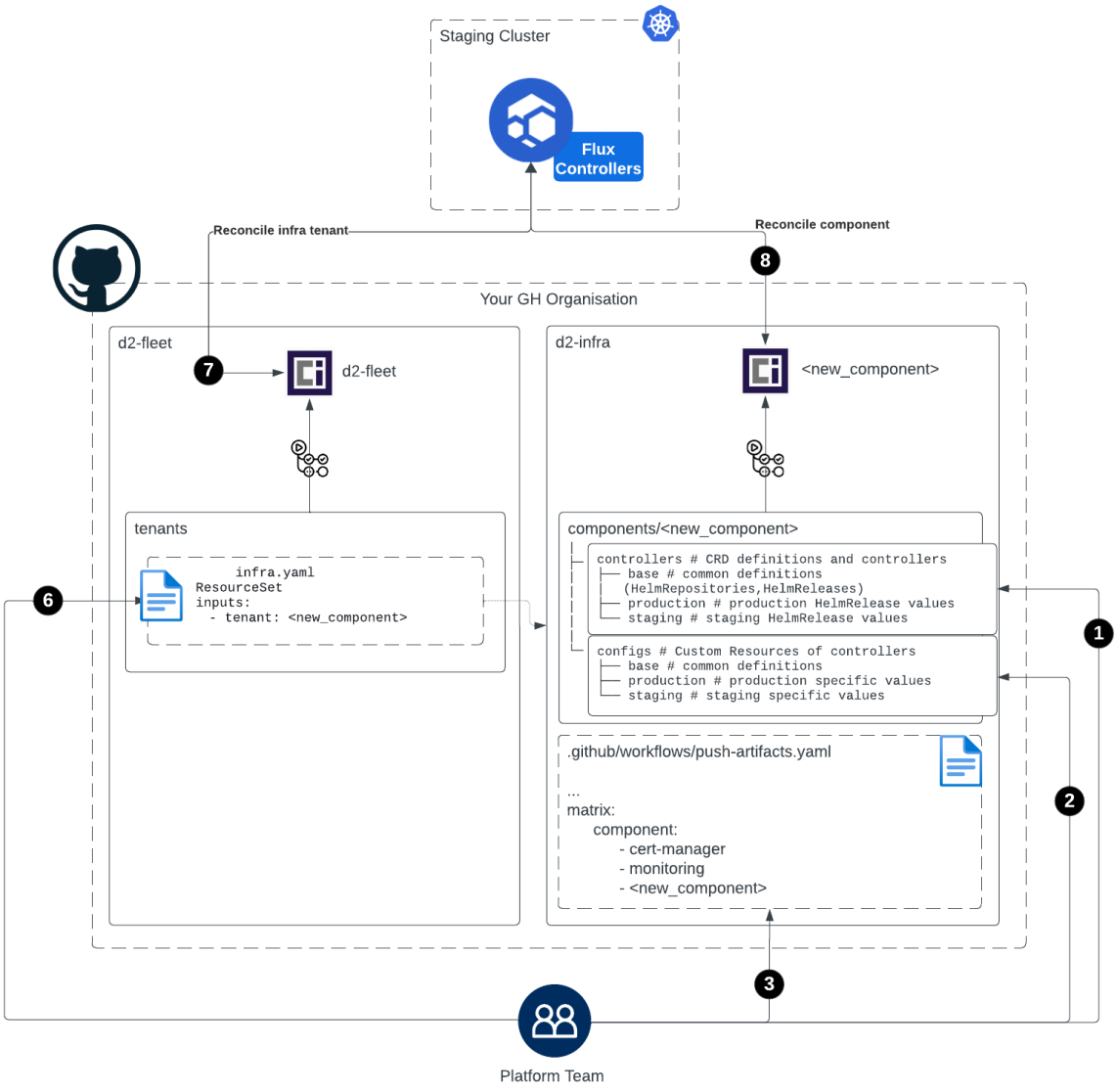

Figure 4: Example Workflow - Adding a new infrastructure component

To onboard a component from the *d2-infra* repository, the Platform team should fulfil the following requirements:

- Read-write access to *d2-fleet* repository
- Read-write access to *d2-infra* repository

The Platform team should then push on a short-lived branch of the *d2-infra* repository:

1. Add the infra component manifests and overlay to the *[d2-infra]/components/<new_component>/controllers* folder, in order to deploy the component itself (i.e. using `HelmRelease` and `HelmRepository/OCIRepository`) and any needed CRDs
2. Add the needed manifests and overlay to the *[d2-infra]/components/<new_component>/configs*, in order to deploy any configuration and custom resources used by that component
3. Add the *<new_component>* name to the input matrix of the push-artifact action, under `jobs.flux-push.strategy.matrix.component`
4. Automatize updates (optional):
    a. Add `ImageRepository` and `ImagePolicy` CR to *[d2-infra]/update* folder
    b. Add all the needed markers for the update automation workflow

The Platform team should open a PR and merge this short-lived branch to the `main` branch. This will trigger the creation of a new component artifact tagged with `latest`.

To make the artifact available for production clusters, the Platform team will have to push a tag `<new_component>/<version>`. This will trigger the `release-artifact` GitHub action, that will build the OCI Artifact and upload it under with the `latest-stable` tag.

Similarly, the team should push changes on a short-lived branch of the *d2-fleet repository* in the following way:

5. (Optional) add any useful cluster variable to *[d2-fleet]/clusters/CLUSTER/runtime-info.yaml* `ConfigMap`
6. Add new `spec.inputs` in the infra `ResourceSet`, *[d2-fleet]/tenants/infra.yaml* — these inputs will be used to build the needed resources following the template defined in the `ResourceSet`

As soon as the short-lived branch is merged to main, GitHub will build the `latest` OCI Artifact. Similarly to the *d2-infra* workflow, pushing a tag on the repository will trigger the creation of fleet OCI Artifact with `latest-stable` tag.

When Flux detects a drift in the OCI Artifacts, the controllers will then:

7. Reconcile the `infra` tenant `ResourceSet` and then notice two new inputs
8. Download the new OCI Artifact required by the new infra tenant
9. Reconcile it

Adding a New Application Team/Tenant

The Platform team are responsible for adding a new tenant for each Application team, following the process below.

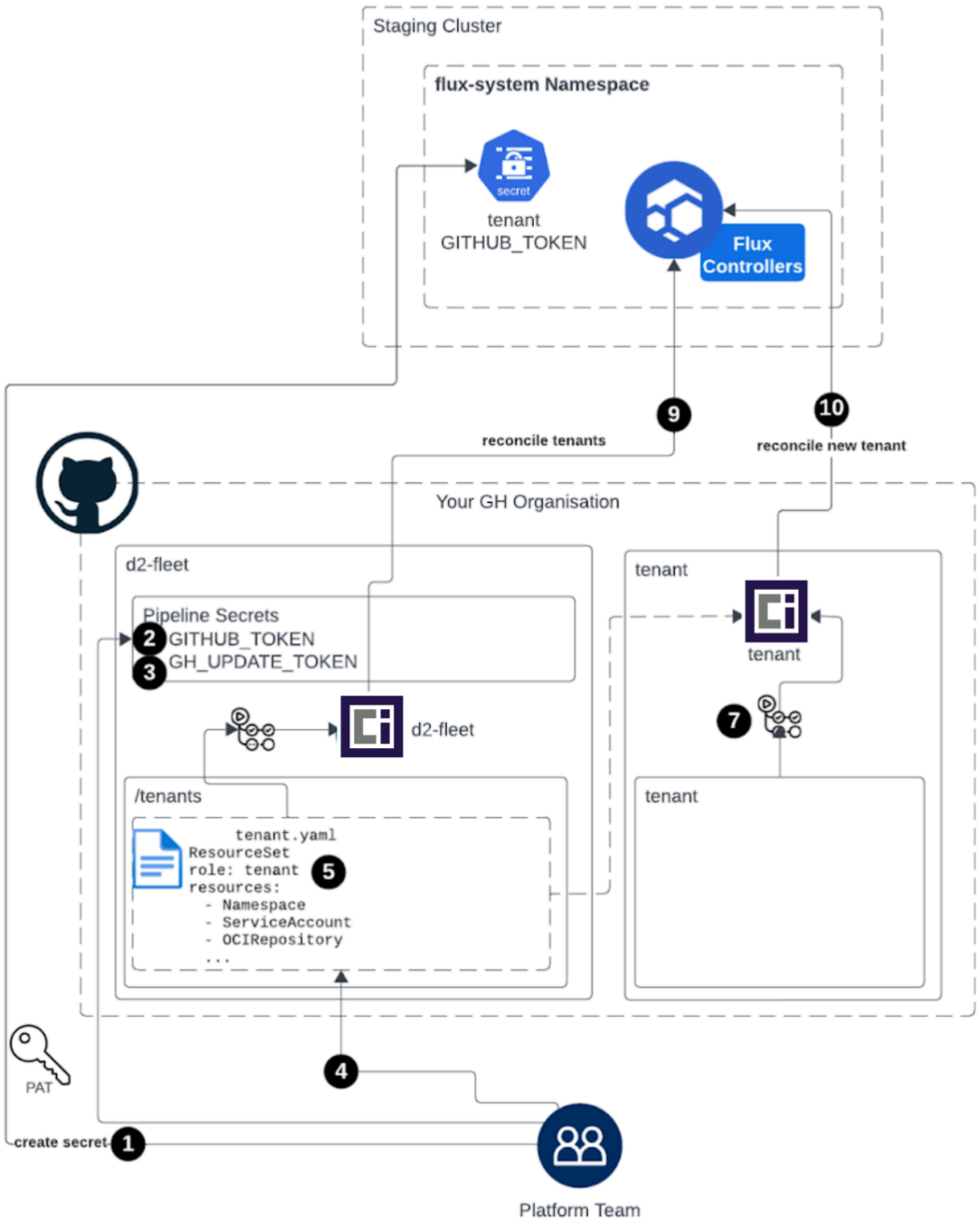

Figure 5: Example Workflow - Adding a new Application team or tenant

To onboard an application tenant, the platform should fulfil the following requirements:

- Read-write access to *d2-fleet* repository
- Knows the classic PAT for the new tenant

The Platform team should prepare the *d2-fleet* repo for the new tenant by adding Kubernetes manifests and preparing the CI actions:

1. Store `GITHUB_TOKEN` in all the clusters that will need to reconcile the new tenant. Ensure the PAT can access the new tenant OCI Artifact registry
2. (optional) Store `GITHUB_TOKEN` as a *d2-fleet* pipeline secret, for update automation purposes
3. (optional) Ensure the `GH_UPDATE_TOKEN` stored in *d2-fleet* CI secrets can access the new tenant git repository, for update automation purposes
4. Create a new `ResourceSet` *[d2-fleet]/tenants/NEW_TENANT.yaml* to define the template and the input set for the new tenant (e.g. *[d2-fleet]/tenants/apps.yaml* — this will be populated by resources for each application (`Namespace`, `GitRepository`, `Kustomization`, RBAC, etc). The `ResourceSet` deploys an `OCIRepository`, with a URL built like this: `oci://REPO/TENANT/<< inputs.tenant >>`
   Notice the `<< inputs.tenant >>` variable will affect which OCI Artifact is going to be downloaded
5. If needed, ensure the Namespace declared in the `ResourceSet` is labeled with `toolkit.fluxcd.io/role: "tenant"`. This applies validating policies to all the namespaces belonging to the tenant
6. Ensure the relevant `copyFrom` annotation is populated with the `ConfigMaps` and `Secrets` to be copied
7. Create the proper CI action to automatically build and push the OCI Artifacts to the registry following pushes and PRs
8. (optional) Create an action to automatically create a PR to main as soon as a new update branch is detected, for update automation purposes

Flux will then:

9. Reconcile the tenants and then notice a new `ResourceSet` for the new tenant
10. Start reconciling the new tenant `ResourceSet` *[d2-fleet]/tenants/NEW_TENANT.yaml*

The Application team responsible for the new application tenant must manage their manifests inside the newly pointed OCI Artifact.

Managing an Application

Manifests that deploy namespaced objects belonging to the app are managed in the proper app tenant. Adding a new app will require joint work between the Platform team and the Application team responsible for that application.

The Application team should identify the needed resources that can't be deployed by a namespaced admin role to the Platform team. This is because they cannot be automatically reconciled by the Flux role assigned to the application's namespace.

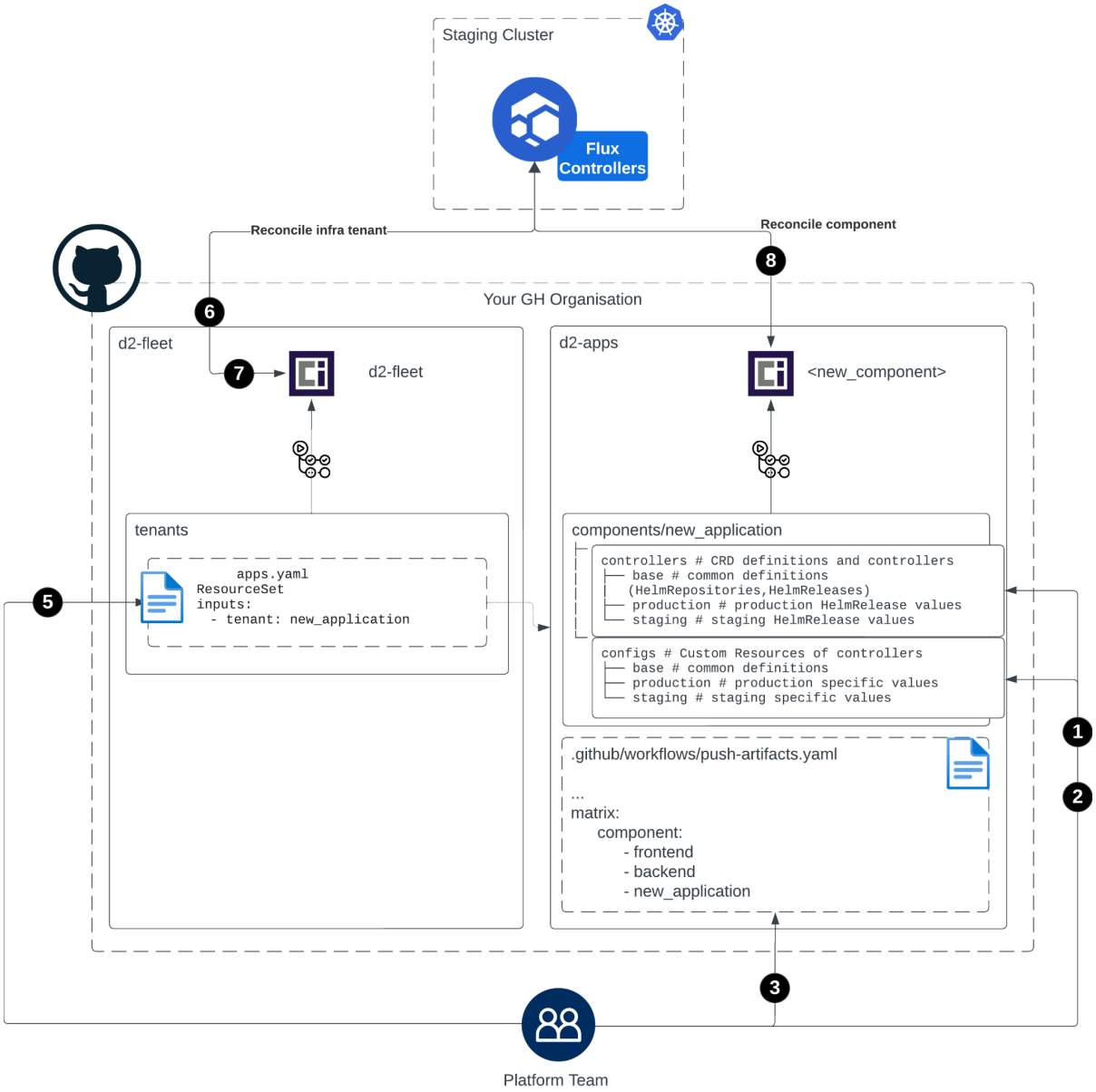

Figure 6: Example Workflow - Managing an application

To onboard a new application to an existing tenant the Platform team should fulfil the following requirements:

- Read-write access to *d2-fleet* repository, where the OCI Artifacts related to the fleet are built from

The Application team should fulfil the following requirements:

- Read-write access to the application tenant, where the OCI Artifact related to the application are built from

The Application team should push to the application tenant:

1. Push the needed manifests (i.e. `HelmRepository` and `HelmRelease`) on their repository
2. Push any needed environment overlays
3. Ensure the proper actions exist to build, tag, and push OCI Artifacts to the registry, and add the application name to the matrix input of the action
4. (optional) Push the manifests needed for the update automation (`ImagePolicy` and `ImageRepository`)
5. The Platform team should then add a new entry in the `spec.inputs` of the tenant `ResourceSet` in the *d2-fleet*

Flux will then:

6. Reconcile the new input in the application tenant `ResourceSet`
7. Deploy what is defined in the template (namespace, service account, `OCIRepository`, etc)
8. Start reconciling the OCI Artifact on the specific path where the `Kustomize` overlays are stored. This Artifact is pointed by the `Kustomization` and `OCIRepository` defined from the new `ResourceSet` input

This would be a possible modification for a new application folder in *d2-fleet/tenants/apps.yaml* `ResourceSet`:

```
Unset
...
  inputs:
    - tenant: "new_application"
      tag: "${ARTIFACT_TAG}"
      environment: "${ENVIRONMENT}"
      artifactSubjectWorkflow: "${ARTIFACT_SUBJECT_WORKFLOW}"
      artifactSubjectGitRef: "${ARTIFACT_SUBJECT_GIT_REF}"
...
```

Further Implementation Guidance

Guardrails on reconciliation

As shown in the flux-runtime-info section, the Flux D2 model ensures that OCI Artifacts are reconciled after their signatures have been verified. In workload-identity, we discussed how tampering with non-main branches results in their OCI Artifacts being rejected by the registry RBAC. However, this only applies if the Container Registry supports fine-tuned RBAC. As an additional guardrail, Flux D2 uses runtime variables to verify OCI Artifacts signatures come from the correct GitHub action run.

```
Unset
apiVersion: v1
kind: ConfigMap
metadata:
  name: flux-runtime-info
  namespace: flux-system
  labels:
    toolkit.fluxcd.io/runtime: "true"
  annotations:
    kustomize.toolkit.fluxcd.io/ssa: "Merge"
data:
  ARTIFACT_TAG: latest
  ENVIRONMENT: staging
  CLUSTER_NAME: staging-1
  CLUSTER_DOMAIN: preview1.example.com
  ARTIFACT_SUBJECT_WORKFLOW: push-artifact
  ARTIFACT_SUBJECT_GIT_REF: refs/heads/main
```

Flux replaces some placeholders in some manifests while applying `Kustomizations` using a specific `ConfigMap` `flux-runtime-info` deployed in each cluster.

This approach has already been used in D1. In D2, additional variables (in bold) are being used to introduce additional guardrails on how the reconciliation is performed:

- `ARTIFACT_TAG`: the tag Flux will reconcile from (e.g. latest)
- `ARTIFACT_SUBJECT_WORKFLOW`: the CI action that generated the OCI Artifact (e.g. push-artifact)
- `ARTIFACT_SUBJECT_GIT_REF:` the branch the workflow was started from (e.g. main)

These play an important role in conjunction with the `apps` tenant `ResourceSet`.

```
Unset
apiVersion: fluxcd.controlplane.io/v1
kind: ResourceSet
metadata:
  name: apps
  namespace: flux-system
...
spec:
...
  inputs:
    - tenant: "frontend"
      tag: "${ARTIFACT_TAG}"
      environment: "${ENVIRONMENT}"
      artifactSubjectWorkflow: "${ARTIFACT_SUBJECT_WORKFLOW}"
      artifactSubjectGitRef: "${ARTIFACT_SUBJECT_GIT_REF}"
...
  resources:
    - apiVersion: source.toolkit.fluxcd.io/v1beta2
      kind: OCIRepository
      metadata:
        name: apps
        namespace: << inputs.tenant >>
      spec:
        interval: 5m
        serviceAccountName: flux
        url: "oci://ghcr.io/controlplaneio-fluxcd/d2-apps/<< inputs.tenant >>"
        ref:
          tag: << inputs.tag >>
        verify:
          provider: cosign
          matchOIDCIdentity:
          - issuer: ^https://token\.actions\.githubusercontent\.com$
          subject:
^https://github\.com/controlplaneio-fluxcd/d2-apps/\.github/workflows/<<
inputs.artifactSubjectWorkflow >>\.yaml@<< inputs.artifactSubjectGitRef >>$
```

In this case:

- `ARTIFACT_SUBJECT_WORKFLOW: push-artifact`: the action being used to push pre-production Artifacts that Flux expects in the artifact signature
- `ARTIFACT_SUBJECT_GIT_REF: refs/heads/main`: the branch Flux expects in the artifact signature, which ensures that only Artifacts built as part of a GitHub action run in `main` will pass the verification process for this cluster

These variables will populate the input matrix for the `ResourceSet`, and the source-controller will verify the signature according to the `OCIRepository.spec.verify.matchOIDCIdentity` field. For the source-controller to trust the Artifact, the OIDC identity associated with the signature must fulfil the following requirements in order:

- Be a cryptographically valid identity
- Have a specific `subject` field, containing a specific action run a specific branch `ref`

Even though finer-grained RBAC is not available on all registries, Flux D2 affords some leeway to reject Artifacts pushed by tampered actions or un-approved branches, as demonstrated in the example above, where verification requires a very specific issuer and subject.

Github Role-Based Access Control (RBAC)

Two PATs (potentially generated from separate BOT accounts) to respectively manage OCI Artifacts download and update automation.

Classic PAT for GHCR access

D1 uses Fine-Grained PATs to access git repositories and reconcile manifests, as they allow to specify which git repositories the access is limited to. The same approach cannot be used to limit access to OCI Artifacts stored in GHCR, as the latter only supports classic Personal Access Tokens.

As per the Cluster Onboarding section, a new GitHub account is required for the Flux bot. The Flux CLI and the Flux controllers running on clusters use this account to authenticate with GitHub during cluster bootstrap, fleet (*d2-fleet* repo), infrastructure (*d2-infra* repo), and application (*d2-apps*) reconciliation.

For convenience, the newly created Flux bot account will be managed by the Platform team in your organisation with the following permissions:

- Read access to the *d2-fleet* repository (for cluster bootstrap)
- Read access to the *d2-infra* repository (for infra reconciliation)
- Read access to the *d2-apps* repository (for apps reconciliation)

For these permissions to be granted, the bot account must be part of the GitHub Organisation and also part of an Organisation Team that has read and write permissions to all relevant repositories.

Access to GHCR requires only the classic PAT to have `read:packages` permission. The reader should be aware that classic PAT generated from a bot account with `read:packages` permission will automatically inherit access to all the artifacts the bot account has access to. This may not be in line with the least privilege principle, especially if the bot account the PAT has been generated from accounts with access grants on the entire organization.

Fine-grained PAT for Update Automation

The Controllers must be able to update the manifests stored *d2-infra* and *d2-apps* repositories. Therefore, Flux will have to be bootstrapped with a fine-tuned Personal Access Token (PAT) with the proper grants. The PAT is saved as a pipeline secret in `secrets.GH_UPDATE_TOKEN`, and populates the `GH_UPDATE_TOKEN` environment variable during the workflow run.

The update automation process automatically creates a Kind cluster in CI, installs Flux and starts the update automation process that will check whether new versions are available, and pushes updates on a specific `image-updates` branch of *d2-infra* and *d2-apps*. The update workflow is executed in the context of *d2-fleet,* therefore it will need a specific fine-grained PAT saved as CI secret `secrets.GH_UPDATE_TOKEN` to be able to read and write on *d2-infra* and *d2-apps*. This variable will be used by the Flux bootstrap script to create a new `Secret` on the cluster, `github-auth`. This `Secret` will be used by `ImageUpdateAutomation` CRs to connect to GitHub..

For convenience, this PAT will be built from a separate Flux Update Automation bot account with the following permissions:

- Read and write access to the *d2-infra* repository (required to push infra updates)
- Read and write access to the *d2-apps* repository (required to push apps updates)

The PAT need the following permissions:

- *Administration* -> Access: Read-only
- *Commit statuses* -> Access: Read and write
- *Contents* -> Access: Read and write
- *Metadata* -> Access: Read-only

This PAT does not need any access to GHCR, as:

- OCI Artifacts are going to be built and uploaded on the registry by another CI run if the `image-updates` branch is merged to `main`
- Cluster reconciliation is managed by a dedicated fine-grained PAT, as specified in the previous section

GitHub App

Although the D2 automation workflow relies upon fine-tuned PATs, a GitHub App may be used as a viable alternative. Users can specify `GitHub` as the `.spec.provider` of the GitRepository referenced by the ImageUpdateAutomation CRs. You will need to register a new GitHub App with the proper read/write grants on the repos and generate a private key for the app. The controllers will then authenticate to GitHub as an app when cloning from and pushing to the git repository.

About ControlPlane Enterprise for Flux CD

In January 2024, we launched ControlPlane Enterprise for Flux CD (CPE-Flux CD), designed to enhance highly regulated organisations that run and manage Production Kubernetes environments. CPE-Flux CD provides a secure, stable, and efficient subscription-based offering for enterprise Kubernetes environments and is designed to ensure the long-term sustainability of the upstream project.

We provide this sustainability by employing core Flux CD maintainers and offering additional security and vulnerability remediation enhancements. If your organisation requires or benefits from these enhancements over the upstream Flux CD project, CPE-Flux is an option to consider. It features:

- Enhanced security: additional guardrails, hardened and distroless container images, and FIPS-compliant Flux CD builds
- Vulnerability management: continuous scanning for CVEs, patching and distribution of Flux CD, and SLAs for critical vulnerabilities with signed VEX documentation
- Seamless upgrades and maintenance: zero-downtime migration, full upstream compatibility, OCI-compliant image repositories, compatibility with the latest six Kubernetes releases, and SLAs for critical vulnerability remediation
- Dedicated support: developer support portal for tracking and managing resolution and feature requests

This additional support, and enhanced security feature set, makes ControlPlane Enterprise for Flux CD a great choice for companies aiming for operational excellence of their Kubernetes environments, especially those dealing with the traits of demanding environments such as:

- Security-critical workloads requiring a clear segregation of duties
- Enforced least privilege Service Accounts
- At-scale deployments with large numbers of pods and nodes
- CI/CD pipelines with low latency delivery requirements
- Availability-critical deployments of new features
- Lack of highly specialised personnel capable of debugging low-level Flux CD integration issues
- Standards or regulations mandating that third-party integrations have to be compliant with security best practices

CPE-Flux CD is backed by ControlPlane's Technical Account Architect (TAA) offering and professional services team, providing additional specialised resources and support as needed.

It is important to emphasise that while CPE-Flux CD offers these additional security and remediation enhancements, if they are not a requirement in your organisation you should use the upstream Flux CD project.

ControlPlane's team is available to support you in achieving secure, stable, and efficient Kubernetes Production environments, whether you choose to use CPE-Flux or the upstream Flux CD project. By offering CPE-Flux, ControlPlane aims to ensure the sustainability and continued development of the upstream Flux CD project.

About

ControlPlane is a global cloud native and open source cybersecurity consultancy operating in London, New York, and Auckland. We have industry-leading expertise in the architecture, audit, and implementation of zero trust infrastructure for regulated industries. With a deep understanding of secure-by-design and secure-by-default cloud, Kubernetes, and supply chain security we conduct threat modelling, penetration testing, and cloud native security training to the highest standard.

ControlPlane is trusted as the partner of choice in securing: multinational banks; major public clouds; international financial institutions; critical national infrastructure programs; multinational oil and gas companies, healthcare and insurance providers; and global media firms.

https://control-plane.io/

Team

Andrea Martino
Matheus Pimenta

Reviewers

Andrew Martin
Stefan Prodan
Leigh Capili