Containers

GitOps model for provisioning and bootstrapping Amazon EKS clusters using Crossplane and Flux

by Viji Sarathy | on 11 JAN 2022 | in Amazon Elastic Kubernetes Service, Containers, Expert (400), Technical How-to | Permalink | Share

In an earlier blog (Part 1 of the series), I discussed the adoption of the GitOps model as an efficient strategy for provisioning cloud provider-specific managed resources, such as, for example, Amazon S3 bucket and Amazon RDS instance, that application workloads depend on. The blog presented the details of implementing a use case where an Amazon EKS cluster was employed as the central management cluster to manage the task of provisioning workload EKS clusters and then deploying applications workloads onto them. The implementation used Crossplane to define the relevant AWS infrastructure resources using Kubernetes-style declarative semantics and ArgoCD to provision them and deploy workloads using the GitOps approach.

In this blog post (Part 2), I will present the details of implementing the same use case using Flux for GitOps-based deployment.

Source code

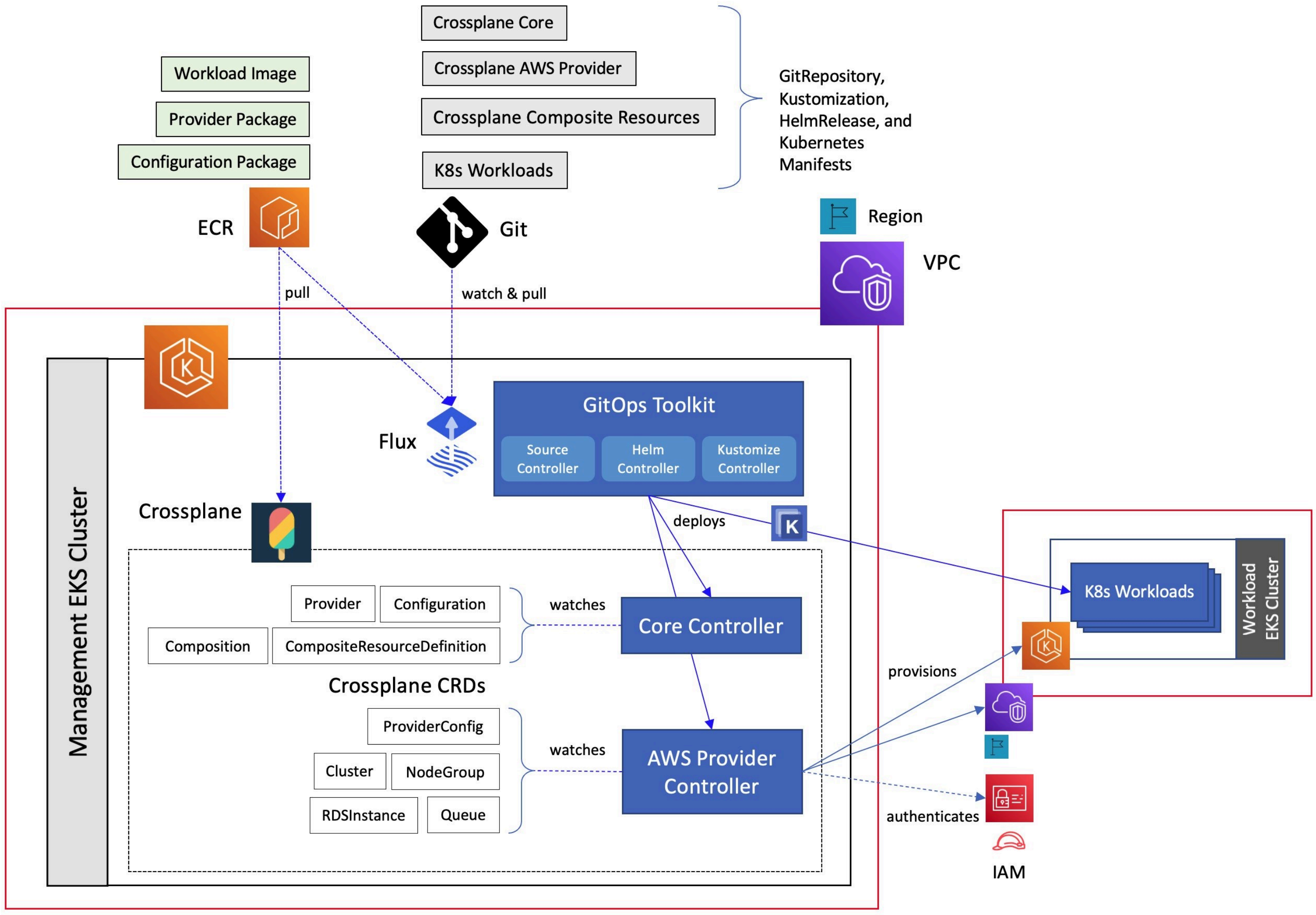
Deployment artifacts for the solution outlined in this blog are available in this GitHub repository for readers to try this implementation out in their clusters. The script flux.md provides the commands to install Flux in a cluster, deploy Crossplane components to provision remote clusters, and deploy workloads on them, all using the GitOps approach.

Architecture

Please refer to this blog post for details about the mechanics of provisioning an Amazon EKS cluster using Crossplane. The companion GitHub repository for that blog provides the relevant installation scripts. The implementation here uses the same version of Crossplane (1.4.1), AWS Provider package (0.17.0), as well as the CompositeResourceDefinition and Composition types that are used to create the configuration package. The latter will provision the complete infrastructure for setting up an EKS cluster—VPC, subnets, internet gateway, NAT gateways, route tables, and the EKS cluster with a managed node group.

Here’s the high-level overview of the solution architecture.

- Start off with an Amazon EKS cluster created using any one of the approaches outlined here.
- Install and bootstrap Flux on this cluster to manage all deployment tasks, pointing to a Git repository containing the deployment artifacts.
- Deploy Crossplane components needed to manage the lifecycle of AWS-managed service resources.
- Deploy Crossplane composite resource to provision an Amazon EKS cluster.
- Deploy a set of workloads to the new cluster.



GitOps deployment with Flux

An imperative approach to provisioning an Amazon EKS cluster using Crossplane involved the following sequence of steps.

1. Deploy the core Crossplane controller as well as Custom Resource Definitions (CRD) under the *.crossplane.io groups.
2. Deploy an OCI image that contains the Crossplane provider package. This will install Crossplane's AWS provider-specific controller and CRDs under the *.aws.crossplane.io groups.
3. Configure the provider with AWS IAM credentials.
4. Build an OCI image that represents a Crossplane configuration package that comprises the CompositeResourceDefinition and Composition types needed for provisioning an Amazon EKS cluster with a managed node group. Push this OCI image into a registry.
5. Deploy the Crossplane configuration package using the OCI image.
6. Finally, deploy a Composite Resource that triggers the provisioning of an Amazon EKS cluster.

Deploying Crossplane with Flux

Let's take a look at how this may be done using a declarative approach using Flux. Flux follows the GitOps pattern of using Git repositories as the source of truth for defining the desired state of a cluster. As I did in the earlier blog, I will again assume that the reader is familiar with the core concepts of implementing a continuous deployment (CD) workflow using Flux. I will elaborate on some of the implementation details of leveraging the GitOps workflow in Flux to manage both cluster provisioning with Crossplane as well as remote cluster management.

Flux GitOps Toolkit is installed on the management cluster per the steps outlined in this script and points to the eks-gitops-crossplane-flux Git repository as the source of truth, also referred to as the *config repository*. This bootstrapping process will create the directory hierarchy *clusters/$CLUSTER_NAME* starting at the top level of this config repository.

In the simplest repository structure, Kubernetes deployment artifacts may be stored directly under this directory, and Flux will ensure that the state of the cluster is kept in sync with those artifacts. Making use of the GitRepository and Kustomization custom resources allows us to set up a better repository structure with separation of concerns. GitRepository instances define sources for artifacts coming from the config repository and other remote Git repositories. Kustomization instances define where to fetch Kubernetes manifests from under these sources, patch them with *kustomize*, and then apply them to the target cluster.

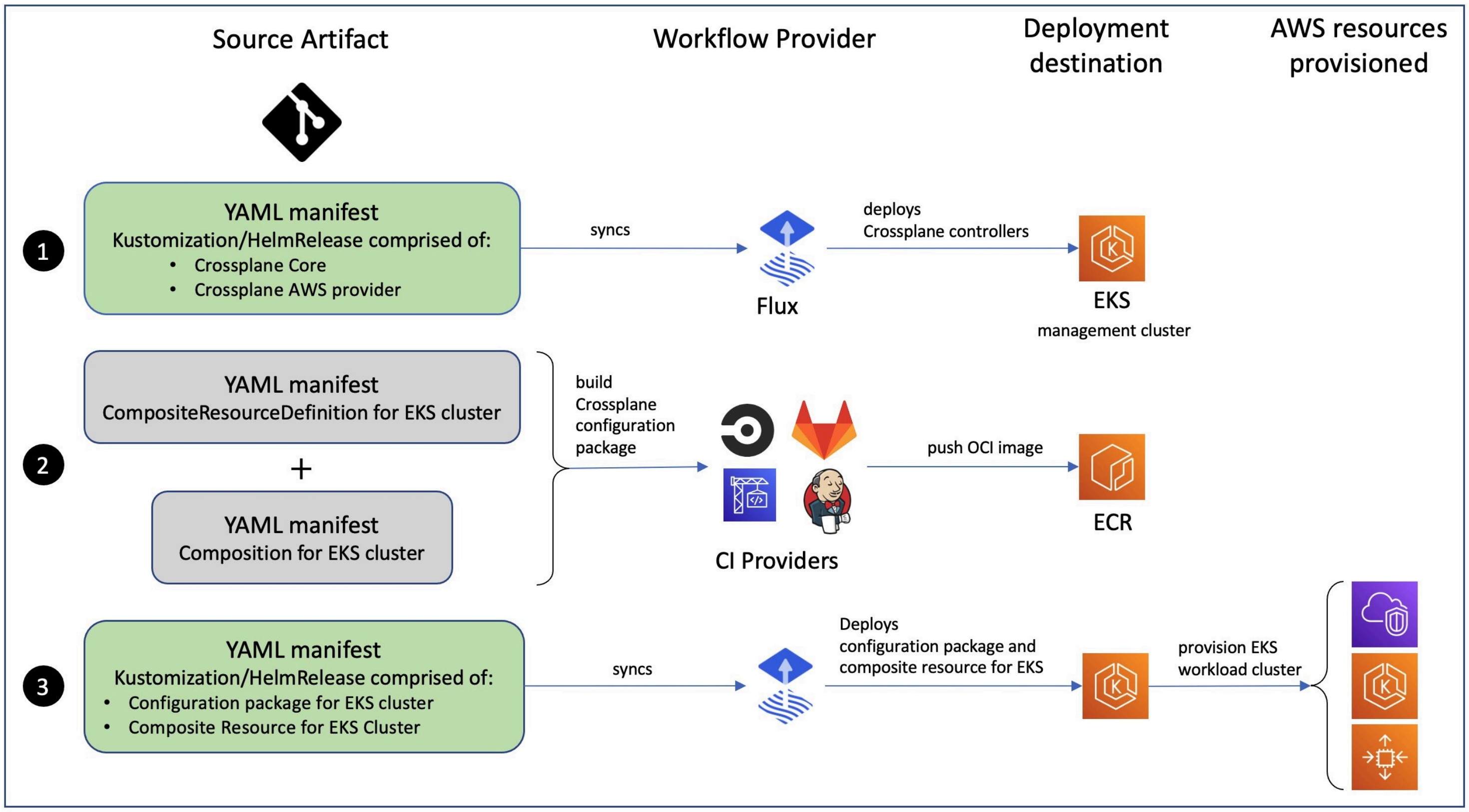
For the GitOps workflow, Crossplane components required for executing steps 1–6 above are packaged into four separate Kustomization resources and deployed to the management cluster in the following sequence:

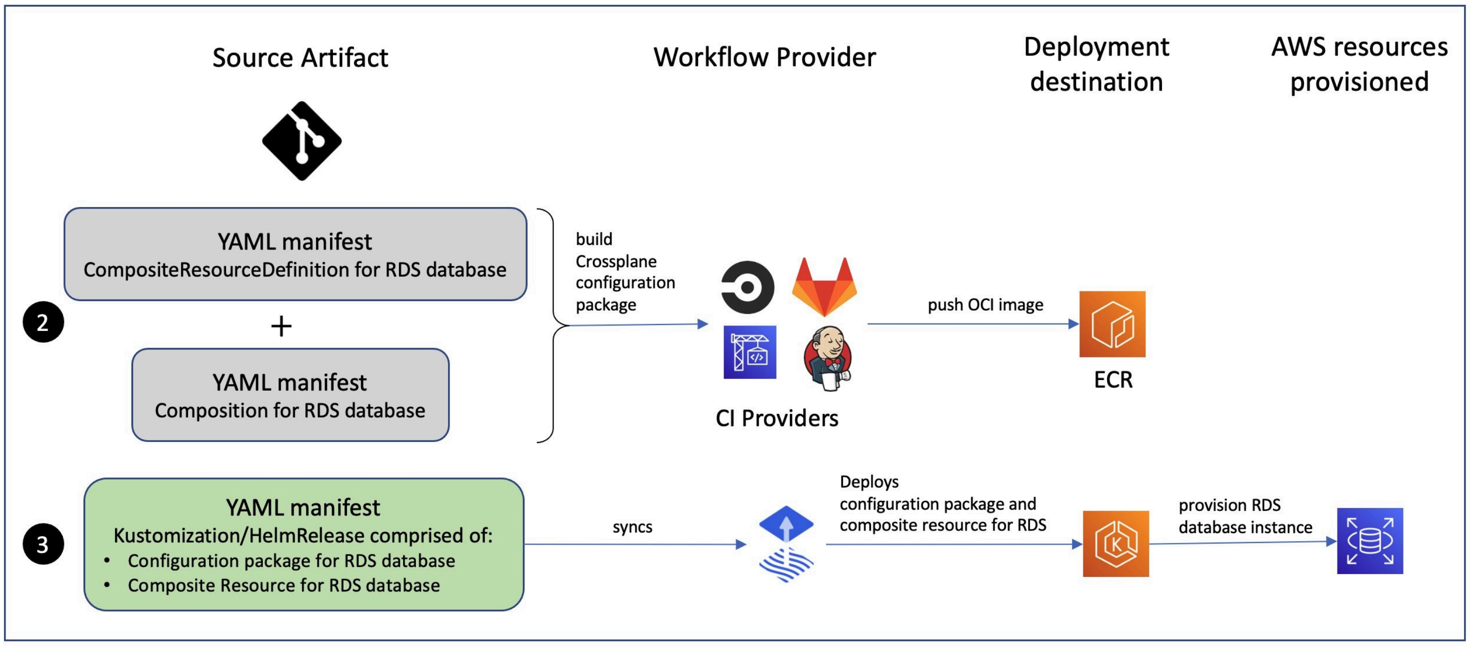
- crossplane-helmrelease.yaml deploys core Crossplane components (version 1.4.1) from a Helm chart using the HelmRelease custom resource.
- crossplane-provider-package.yaml deploys the AWS provider package (version 0.17.0) and the SealedSecrets controller.
- crossplane-configuration-package.yaml deploys the AWS credentials needed by the providers and the Crossplane configuration package that provides the CompositeResourceDefinition and Composition types.
- crossplane-composite-resources.yaml deploys the CompositeResource that triggers the provisioning of the Amazon EKS cluster.

When using ArgoCD, Resource Hooks and Sync Phases and Waves were used to ensure certain resources were deployed and were in a healthy state before subsequent resources were synced to the cluster. With Flux, Kustomization dependencies are used to guarantee that these components are deployed in the specific sequence listed above. This will ensure that each Kustomization is applied only after all of its dependencies are ready. In addition, Kustomization health assessments are used to determine the rollout status of workloads deployed with each Kustomization.

The health check entries support both Kubernetes built-in types such as Deployment and DaemonSet as well as custom resources that are compatible with kstatus. The health status condition reported by Crossplane custom resources is not fully compatible yet with kstatus. Hence, as a stopgap solution, this implementation uses the health status of the underlying built-in types that land on the cluster when these Crossplane custom resources are deployed.

I have again simplified the GitOps workflow for this implementation by assuming that the OCI image for the Crossplane configuration package already resides in a registry. When starting off from a clean slate, this workflow will have to be broken up into multiple segments, as shown in the first illustration below. The first segment (Step 1), which needs to be done only once, will be a GitOps workflow that deploys core Crossplane and the AWS provider package. This will be followed by a CI workflow (Step 2) to build the OCI image for the configuration package and push it to a registry. The final segment (Step 3) will be a GitOps workflow that deploys the configuration package and the Composite Resource to the management cluster. Using this approach to provision other AWS-managed resources such as, say, an RDS database instance, merely requires repeating Steps 2 and 3, as shown in the second illustration below.


Source Artifact
Workflow Provider
Deployment destination
AWS resources provisioned
1
YAML manifest
Kustomization/HelmRelease comprised of:
• Crossplane Core
• Crossplane AWS provider
syncs
Flux
deploys Crossplane controllers
EKS
management cluster
2
YAML manifest
CompositeResourceDefinition for EKS cluster
+
YAML manifest
Composition for EKS cluster
build Crossplane configuration package
CI Providers
push OCI image
ECR
3
YAML manifest
Kustomization/HelmRelease comprised of:
• Configuration package for EKS cluster
• Composite Resource for EKS Cluster
syncs
Deploys configuration package and composite resource for EKS
provision EKS workload cluster


Source Artifact
Workflow Provider
Deployment destination
AWS resources provisioned
YAML manifest
CompositeResourceDefinition for RDS database
2
+
YAML manifest
Composition for RDS database
build Crossplane configuration package
CI Providers
push OCI image
ECR
3
YAML manifest
Kustomization/HelmRelease comprised of:
• Configuration package for RDS database
• Composite Resource for RDS database
syncs
Deploys configuration package and composite resource for RDS
provision RDS database instance

Bitnami's Sealed Secrets is used to secure the AWS credentials required by the Crossplane AWS provider. The Kubernetes Secret that contains the AWS credentials is encrypted into a SealedSecret custom resource that is safe to store in a Git repository. The encryption keys or sealing keys are securely stored in AWS Secrets Manager. They are deployed to the cluster outside the GitOps workflow so that they are readily available to the controller that decrypts the SealedSecret.

Connecting to Remote Clusters with Flux

Flux supports deploying workloads to remote clusters using either a Kustomization or a HelmRelease resource. In both scenarios, it expects a *kubeConfig* field in the resource manifest, which should reference a Kubernetes Secret that provides KubeConfig data needed to connect to the remote cluster. The Secret must expose this data using either the *value* or *value.yaml* key and must reside in the same namespace as that of the Kustomization or HelmRelease resource, which, in this implementation, is *flux-system*. There are two approaches to implementing remote cluster management with Flux.

When Crossplane has successfully provisioned the Cluster.eks.aws.crossplane.io resource, it creates a Kubernetes Secret that exposes the cluster's API server endpoint, certificate authority, and KubeConfig data using the keys *endpoint*, *clusterCA*, and *kubeconfig* respectively. Crossplane allows a CompositeResource, such as the EKSCluster.eks.sarathy.io used in this implementation, to expose its own Kubernetes Secret as shown by the *writeConnectionSecretToRef* section of the YAML manifest below.

YAML
```
---
apiVersion: eks.sarathy.io/v1beta1
kind: EKSCluster
metadata:
  name: crossplane-flux-cluster
spec:
  parameters:
    region: us-west-2
    vpc-name: "crossplane-flux-vpc"
    vpc-cidrBlock: "10.20.0.0/16"

    subnet1-public-name: "public-worker-1 "
    subnet1-public-cidrBlock: "10.20.1.0/28"
    subnet1-public-availabilityZone: "us-west-2a"

    subnet2-public-name: "public-worker-2"
    subnet2-public-cidrBlock: "10.20.2.0/28"
    subnet2-public-availabilityZone: "us-west-2b"
```

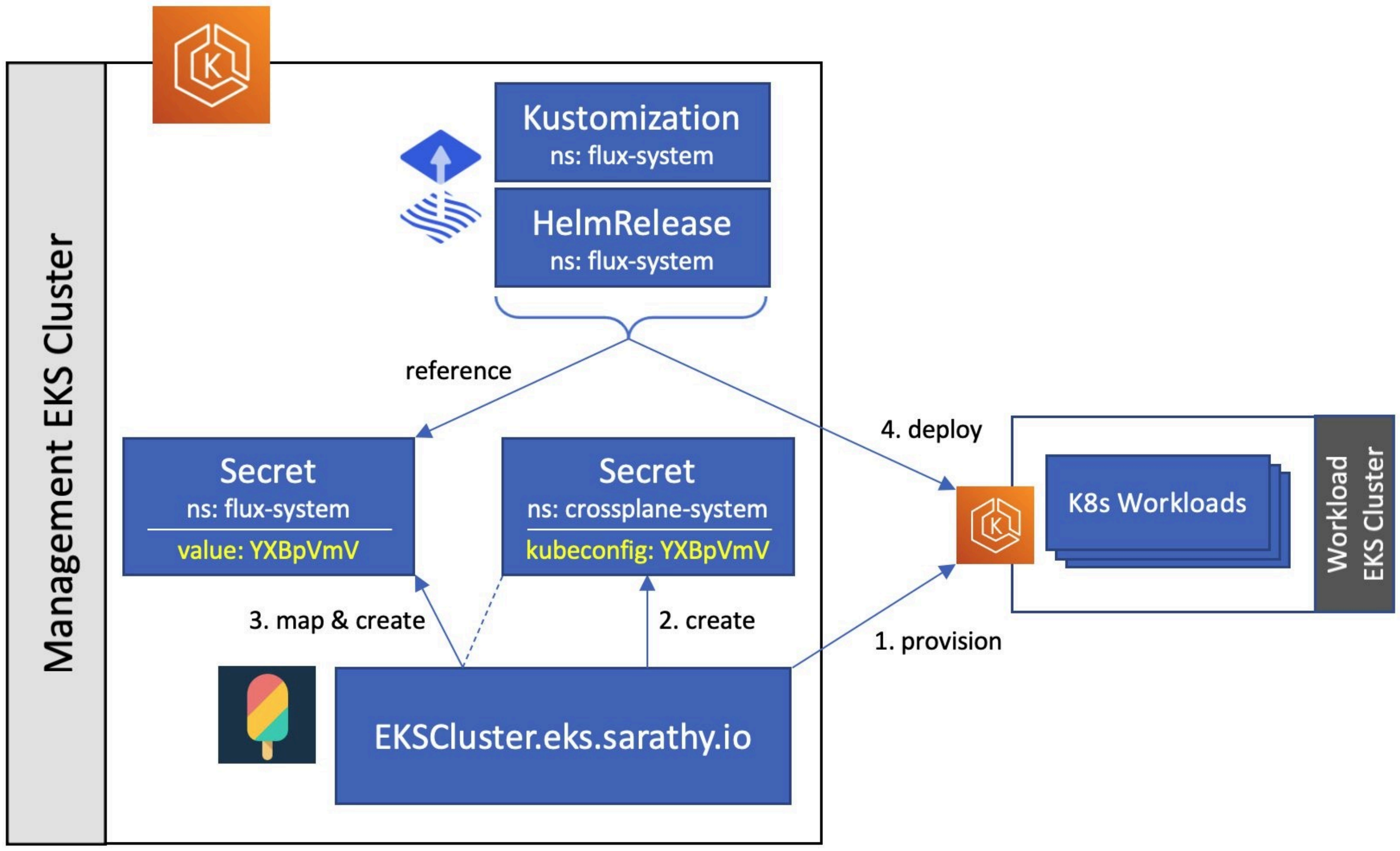
The keys in this Secret can be mapped to the keys in Secrets exposed by the composed managed resources in the underlying Composition. As shown below, these mappings are provided in the *connectionDetails* section of configuration for the Cluster.eks.aws.crossplane.io managed resource that makes up the *amazon-eks-cluster* Composition.

This feature is leveraged to expose a Kubernetes Secret named *crossplane-workload-cluster-connection* in the *flux-system* namespace that provides the KubeConfig data for the workload cluster using the *value* key. Subsequently, a Kustomization or HelmRelease can simply reference this Secret, and Flux will then target that remote cluster when deploying the workloads. An advantage of this approach is that it is seamless, requires no manual steps, and the cluster credentials are rotated at regular intervals. However, it should be noted that the credentials in this Secret pertain to that of the EKS cluster creator that is implicitly and irreversibly mapped to *system:masters* in Kubernetes RBAC. Such an elevated level of permission may not be necessary for deploying applications to the workload cluster.

YAML

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: amazon-eks-cluster
spec:
  compositeTypeRef:
    apiVersion: eks.sarathy.io/v1beta1
    kind: EKSCluster

  resources:
    - name: eks-cluster
      base:
        apiVersion: eks.aws.crossplane.io/v1beta1
        kind: Cluster
        spec:
          writeConnectionSecretToRef:
            namespace: crossplane-system
      connectionDetails:
```



The alternative approach is to manually create a Kubernetes service account in the workload cluster, granting it a minimal set of permissions needed to deploy the applications. The YAML manifest for a service account suitable for deploying the sample applications in this implementation is here. Following this step, a KubeConfig file has to be manually assembled using this service account's authentication token along with the API server endpoint and certificate authority data for the workload cluster. The last step is to create a Secret using this file as shown by the CLI commands in this script. Note that while this approach adheres to the principle of least privileges, the service account credentials don't expire and are valid for as long as the service account exists.

Deploying Applications to Workload Cluster

The applications to be deployed using Flux to the workload cluster are packaged using a set of Kustomization resources listed here. The workloads comprise a sample web application that exposes Prometheus metrics, namely, *http_requests_total* (Counter), and *request_duration_milliseconds* (Histogram). It is deployed along with an instance of Prometheus server that is configured to send metrics to a workspace in Amazon Managed Service for Prometheus. The Prometheus deployment makes use of the IAM Role for Service Account, which could also be created using Crossplane APIs. But in this implementation, the IAM role, as well as the IAM OIDC identity provider, are created outside the GitOps workflow.

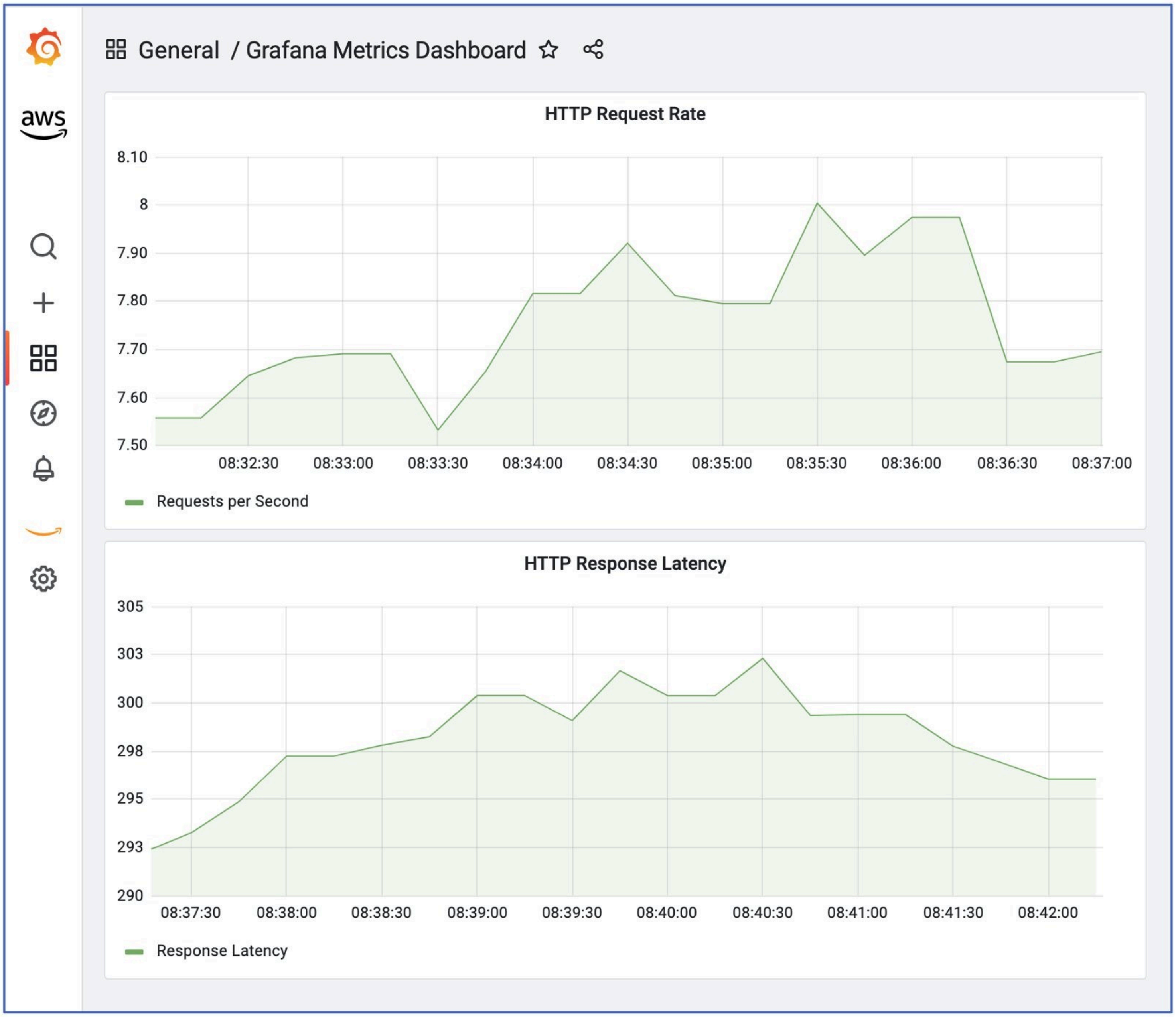
The Kustomization manifest that deploys the web application and the HelmRelease manifest that deploys the Prometheus server to the workload cluster are shown below. Both of them have the *kubeConfig* field that references the Secret that provides KubeConfig data for the workload cluster. This Secret is created by either of the two approaches discussed in the previous section.

YAML

```yaml
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta1
kind: Kustomization
metadata:
  name: application-webapp
  namespace: flux-system
spec:
  interval: 30s
  path: ./deploy/webapp
  prune: true
  sourceRef:
    kind: GitRepository
    name: flux-system
  kubeConfig:
    secretRef:
      name: crossplane-workload-cluster-connection
  validation: client

---
```

Shown below is a Grafana dashboard displaying the metrics scraped from the sample application on the workload cluster.



TAGS: Amazon EKS, Flux, Gitops