Containers

GitOps model for provisioning and bootstrapping Amazon EKS clusters using Crossplane and Argo CD

by Viji Sarathy | on 20 OCT 2021 | in Amazon Elastic Kubernetes Service, Containers, Technical How-to | Permalink |
Share

Customers are increasingly using multiple Kubernetes clusters to manage their application delivery to different environments. Managed services like Amazon Elastic Kubernetes Service (Amazon EKS) help customers offload the onerous task of managing the Kubernetes control plane. But cluster operators face the challenge of managing the lifecycles of these clusters and deploying applications consistently to multiple clusters and environments.

GitOps is a way to do application delivery. It is an operational model that offers customers the ability to manage the state of multiple Kubernetes clusters by extending the best practices of version control, immutable artifacts, and automation.

Flux and Argo CD are both declarative, GitOps-based continuous delivery tools for Kubernetes that have gained a lot of traction amongst users. Both of them follow the pattern of using Git repositories as the source of truth for defining the desired state of a cluster. Both handle only the continuous deployment (CD) portion of a CI/CD pipeline and allow users to choose any continuous integration (CI) workflow provider of their choice. Users also have the flexibility of choosing their Git provider (GitHub, GitLab, BitBucket). The ability to manage application deployments to multiple remote Kubernetes clusters from a central management cluster, support for progressive delivery, and multitenancy are other notable features of both tool kits.

Customers adopt managed services not just for Kubernetes, but also for a host of other services such as messaging, relational databases, key-value stores, etc., in conjunction with their containerized workloads. It is very common for an application deployed on an EKS cluster to interact with managed services such as Amazon S3, Amazon DynamoDB, and Amazon SQS, to name a few. Ideally, cluster operators or application developers do not want to deal with setting up these managed service resources and administering them. They want to automate the management of the lifecycle of these resources using declarative semantics similar to how they manage applications deployed to a Kubernetes cluster. Kubernetes Operator is one approach to implement this automation. It provides a mechanism to extend Kubernetes functionality using Custom Resource Definitions (CRDs) and controllers with domain-specific knowledge needed to provision managed service resources using cloud-provider-specific tools, such as the AWS SDK.

AWS Controllers for Kubernetes (ACK) is a tool that lets you directly manage AWS-managed services from Kubernetes. It is a collection of Kubernetes CRDs and custom controllers working together to extend the Kubernetes API and manage AWS resources on your behalf. Crossplane is a similar tool that extends the Kubernetes API and enables teams to put together an opinionated platform comprising infrastructure services from all the major cloud providers. Crossplane's infrastructure provider for AWS relies on code generated by the AWS Go Code Generator, which is also used by ACK. Through the use of Kubernetes-style APIs (CRDs and controllers), both ACK and Crossplane allow users to define AWS-managed service resources (such as an Amazon RDS instance, SQS queue, etc.) as Kubernetes custom resources using declarative configurations. Subsequently, they can all be provisioned and managed by `kubectl`, GitOps, or any tool that can talk with the Kubernetes API.

In addition to managing the so-called "Day Two" operations, the GitOps model can be used to automate cluster lifecycle management as well. In this blog post, we take a look at implementing a real-world use case by leveraging the strength of two open-source tools, namely, Crossplane and Argo CD. A production-ready Amazon EKS cluster is employed as the central management cluster and used to manage the tasks of both provisioning other Amazon EKS clusters and deploying workloads on to them.

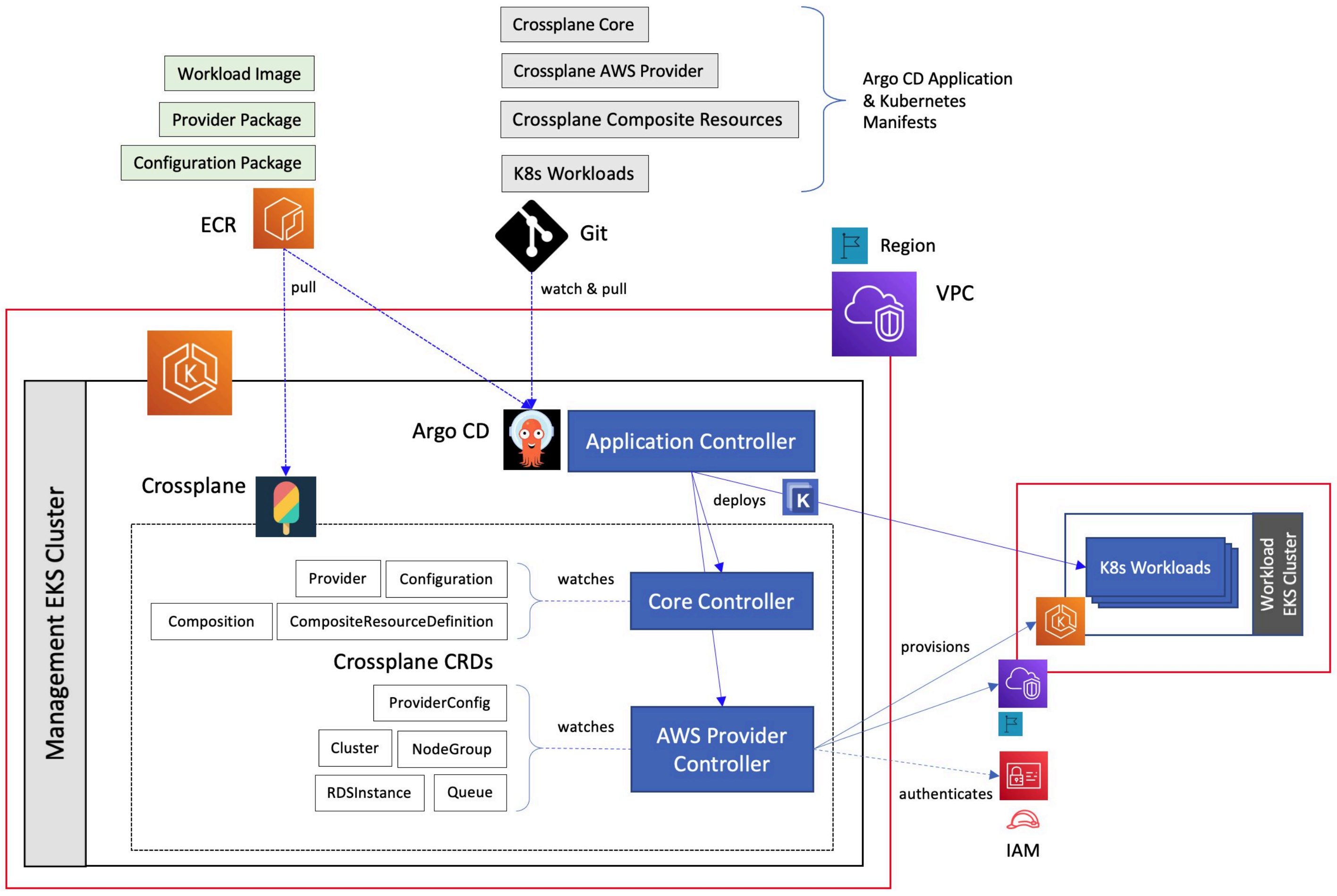
Source code

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 shell script crossplane.sh provides the commands to install Crossplane in a Kubernetes cluster as well as build and deploy the Crossplane Configuration package needed for provisioning an EKS cluster. The shell script argocd.sh provides the commands to install Argo CD in a cluster, then deploy Crossplane to the same cluster and deploy workloads to a remote cluster, all using the GitOps approach.

Architecture

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

- Start with an EKS cluster that was created using any one of the approaches outlined here.
- Install Argo CD on this cluster to manage all deployment tasks and point it to a Git repository containing the deployment artifacts.
- Deploy Crossplane components that are needed to manage the lifecycle of AWS-managed service resources.
- Deploy Crossplane-specific custom resources to provision an EKS cluster
- Deploy a set of workloads to the new cluster



Crossplane concepts

Crossplane has the concept of a Managed Resource (MR) which is a Kubernetes CRD that represents an infrastructure resource in a cloud provider. Crossplane's AWS provider packages the following resources.

- CRDs such as RDSInstance, Cluster, Queue, to name just a few which model resources under AWS-managed services such as Amazon RDS, Amazon EKS, and Amazon SQS, respectively
- Controller to provision these resources in AWS when a user deploys custom resources that these CRDs define in order to provision a Kubernetes cluster.

Crossplane Managed Resources match the APIs of the external system they represent as closely as possible. They expose the same set of parameters that are provided by the corresponding API group in AWS SDK. For example, Queue.v1beta1.sqs.aws.crossplane.io , which represents an Amazon SQS queue instance, exposes parameters such *DelaySeconds*, *MaximumMessageSize*, *MessageRetentionPeriod*, *VisibilityTimeout* and others that make up the complete set of request parameters one would use to create an SQS queue using CreateQueue API. Managed Resources in Crossplane are considered low-level custom resources that can be used directly to provision external cloud resources for an application.

Crossplane introduces the concept of Composition, which allows platform teams to define a new custom resource called a Composite Resource (XR). The XR is composed of one or more Managed Resources. Crossplane uses two special resources to define and configure this new type of custom resource:

- A CompositeResourceDefinition (XRD) that defines the schema for an XR, akin to a CRD.
- A Composition that specifies which Managed Resourcesan XR will be composed of and how they should be configured.

One or more CompositeResourceDefinition and Composition types may be combined into a Configuration Package which is an OCI image containing a stream of YAML that can be pushed into any OCI-compatible registry such as Amazon ECR and Docker Hub. Subsequently, these packages can be installed on a Kubernetes cluster and then used to provision instances of an XR.

Amazon EKS cluster provisioning using Crossplane

Let’s get into the details of provisioning an EKS cluster using Crossplane. First, we need to address the chicken-and-egg problem of needing a Kubernetes cluster in order to create one.

Crossplane provides two options for addressing this. The first one is to use a hosted Crossplane service like Upbound Cloud. The second option is to use your own Kubernetes cluster, which I will refer to as the *management cluster*. The Kubernetes cluster that is created by Crossplane will be referred to as the *workload cluster.* In this blog post, I will be using an existing instance of an EKS cluster running Kubernetes 1.20. The management cluster may also be provisioned using Kind or Minikube.

The CLI commands used in this implementation to install Crossplane and provision an EKS cluster are outlined in this script. First, we need to extend the functionality of *kubectl* CLI in order to build, push and install Crossplane packages as well as be able to easily query Crossplane artifacts in the cluster.

Bash
```bash
curl -sL https://raw.githubusercontent.com/crossplane/crossplane/master/install.sh | sl
sudo mv kubectl-crossplane /usr/local/bin
```

Using Helm 3, install a stable release of Crossplane in the management cluster. I am using version 1.4.1. Crossplane does not have any specific requirements regarding the Kubernetes version. In general, it tries to support all Kubernetes versions until they reach end of life.

Bash
```bash
kubectl create namespace crossplane-system
helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update
helm install crossplane --namespace crossplane-system crossplane-stable/crossplane --v
```

In addition to Configuration package that we discussed earlier, Crossplane defines a Provider package which is used to install Crossplane providers on the management cluster. Crossplane’s AWS provider is installed using the following manifest, which references an OCI image that was built from the source code in the provider-aws repository and is stored in an ECR public registry. I am using version 0.17.0 of the provider.

YAML
```yaml
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: crossplane-provider-aws
spec:
  package: "public.ecr.aws/awsvijisarathy/crossplane-provider-aws:v0.17.0"
```

Deploying this Provider.v1.pkg.crossplane.io custom resource will trigger the core Crossplane controller that was deployed in the previous step to install provider-specific controllers and CRDs related to AWS- managed resources. The provider controllers require credentials to authenticate against AWS Identity and Access Management (IAM). The credentials are provided by deploying the ProviderConfig.v1beta1.aws.crossplane.io custom resource with the following manifest, which references a Kubernetes Secret.

YAML
```yaml
apiVersion: aws.crossplane.io/v1beta1
kind: ProviderConfig
metadata:
  name: default
spec:
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: aws-credentials
      key: credentials
```

The **credentials** key in this Secret should specify IAM credentials shown in the following:

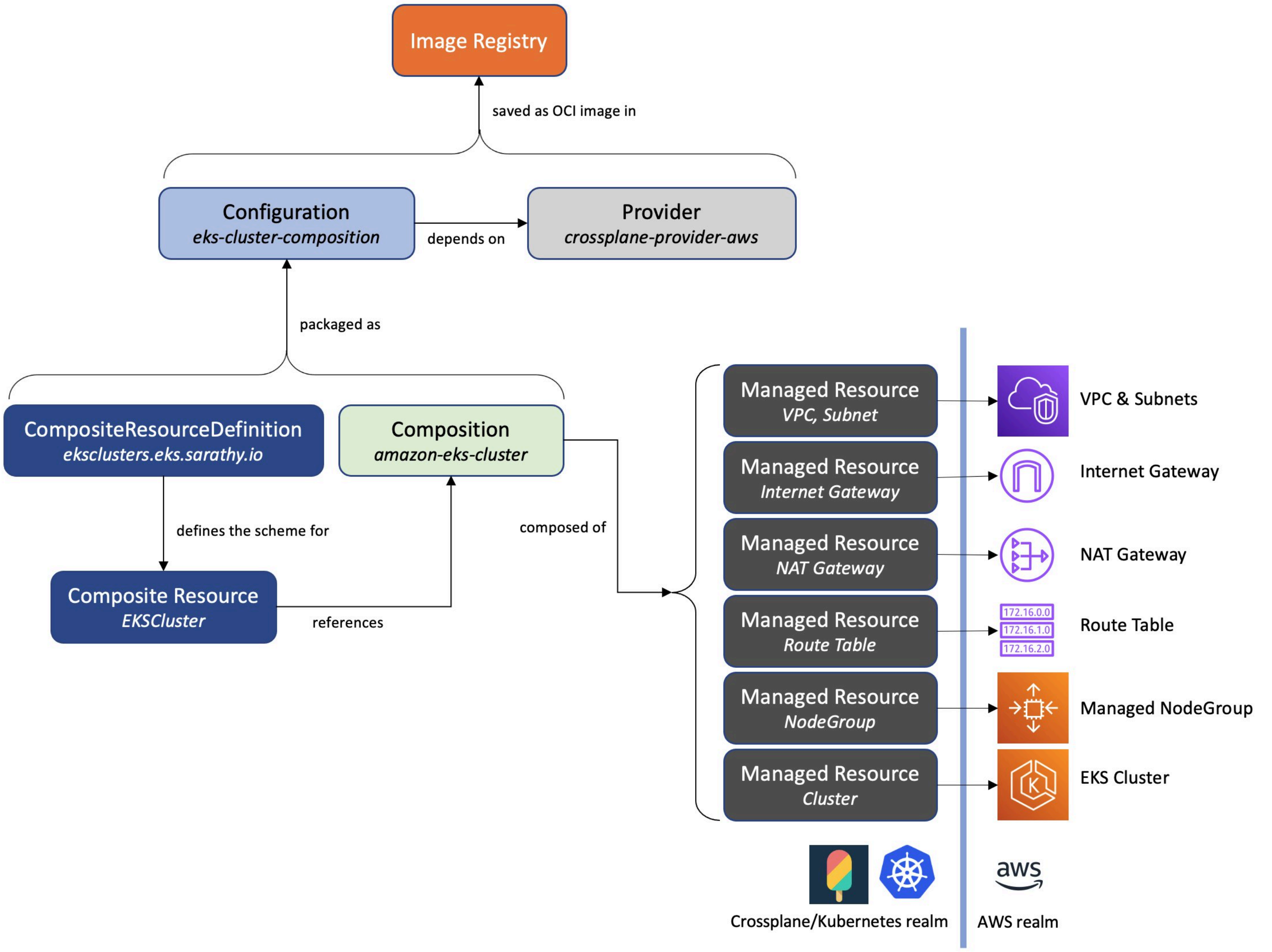
Bash
```bash
[default]
aws_access_key_id =ABCDEFGHIJ0123456789
aws_secret_access_key = Ow3HUaP8BbqkV4dUrZr0H7yT5nGP5OPFcZJ+
```

In this implementation, we are using the credentials of an IAM user that has permissions to only manage Amazon EKS clusters. The IAM permission policy associated with this user is show below. Note that these are exactly the same minimal set of permissions that will be needed in order to create an EKS cluster using the AWS CLI command `aws eks create-cluster`. The IAM role, namely, **EKS-Cluster-Role**, referenced in this policy is the Amazon EKS cluster IAM role which is needed by EKS to make calls to other AWS services on your behalf to manage the resources that you use with the service in the workload cluster.

JSON
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": [
                "arn:aws:iam::111111111111:role/EKS-Cluster-Role"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "eks:DeleteCluster",
                "eks:CreateCluster",
                "eks:DescribeCluster",
```

At this point, Crossplane is ready to provision a workload cluster using Amazon EKS. Before proceeding further, we can run the `kubectl get crossplane` command to check if the Crossplane AWS provider has been installed properly and is in a healthy state.

The next step is to create a Crossplane Configuration package that contains the CompositeResourceDefinition and Composition types with which we can manage the lifecycle of workload clusters. In this implementation, I am creating a Configuration package that 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. The following figure shows the relationship between various Crossplane custom resources used in this Configuration package.



The YAML manifest at this link creates a Composition type named `amazon-eks-cluster` which will be used to configure an XR that is composed of the following Managed Resources – VPC, Subnet, InternetGateway, NATGateway, RouteTable, ElasticIP, Cluster and NodeGroup. A Composition such as this allows a cluster operator to take an opinionated approach to how certain complex AWS-managed resources are provisioned. It abstracts the underlying Crossplane-managed resources and hides many of the low-level parameters required to provision the corresponding AWS resources.

The `amazon-eks-cluster` Composition is an opinionated way of provisioning an EKS cluster. It stipulates that the cluster will run inside a newly created VPC with a fixed number of public/private subnets and managed worker nodes. It hides details such as EC2 instance type, OS image, AMI, etc., to be used for the worker nodes.

The YAML manifest creates the CompositeResourceDefinition type named **eksclusters.eks.sarathy.io,** which defines the schema for an XR that uses the above Composition. The end user will typically use only the XR defined by this schema and work with the high-level parameters that it exposes, such as the CIDR range, AWS Region and Availability Zones, etc.

To create a Configuration package, we must have a manifest named crossplane.yaml**.** This manifest defines a Configuration type that declares semantic version constraints of Crossplane and the AWS provider with which the package is compatible. Here, we are stipulating that the package needs a version of Crossplane later than v1.0.0 and a Crossplane AWS provider later than v0.14.0.

YAML
```
apiVersion: meta.pkg.crossplane.io/v1alpha1
kind: Configuration
metadata:
  name: eks-cluster-composition
  annotations:
    provider: aws
spec:
  crossplane:
    version: ">=v1.0.0"
  dependsOn:
    - provider: crossplane/provider-aws
      version: ">=v0.14.0"
```

Next, we place these three YAML manifests in a directory and run the following set of commands to build the package as an OCI image. Then, we push it to a repository in the public Amazon ECR registry and install it in the management cluster.

YAML
```
kubectl crossplane build configuration
kubectl crossplane push configuration public.ecr.aws/awsvijisarathy/crossplane-eks-com
kubectl crossplane install configuration public.ecr.aws/awsvijisarathy/crossplane-eks-
```

All that remains now is to deploy an XR that will trigger Crossplane's provider-specific controller to create an EKS cluster. The following manifest conforms to the schema defined by the **eksclusters.eks.sarathy.io** CompositeResourceDefinition and uses the `amazon-eks-cluster` Composition. This will create the following list of AWS resources:

- VPC with two private and two public subnets
- Internet gateway and two NAT gateways
- Elastic IP addresses for the NAT gateways
- Route table for the public subnets and a separate route table for each private subnet
- EKS cluster running Kubernetes 1.20, comprising a managed node group of two instances of type `m5.large` in the US-WEST-2 (Oregon) region.

YAML
```
---
apiVersion: eks.sarathy.io/v1beta1
```

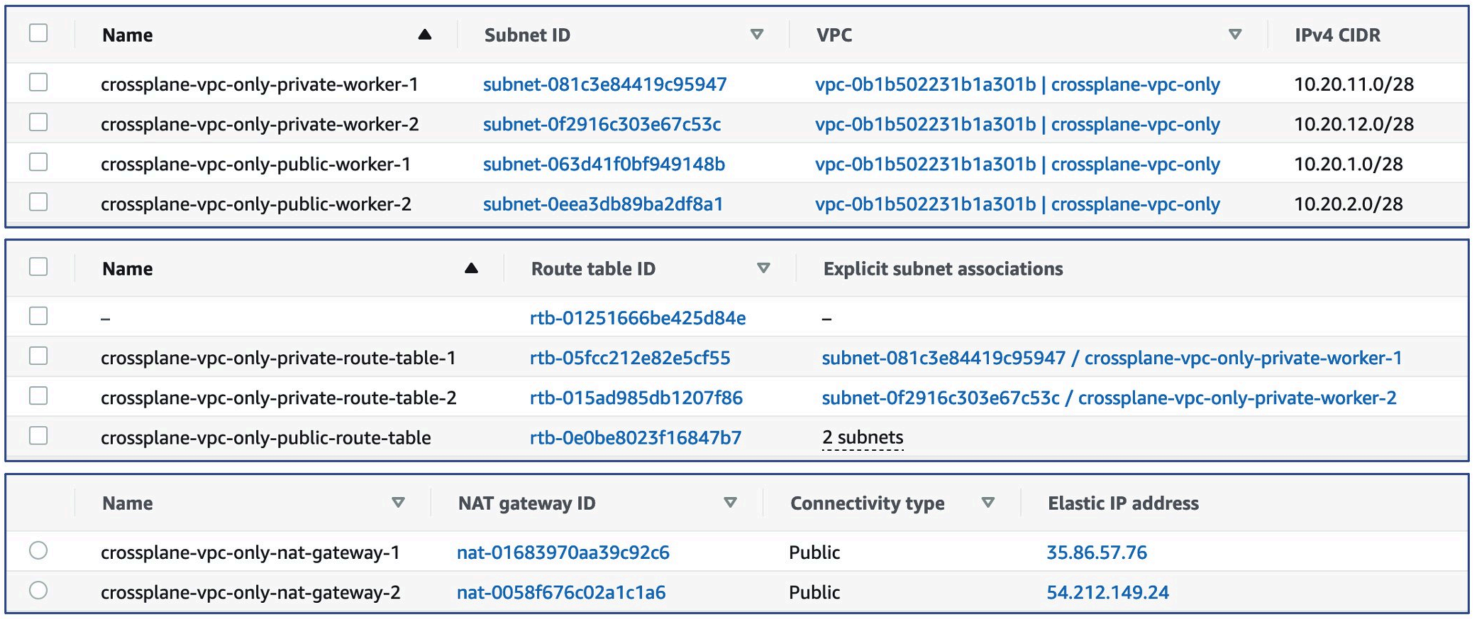
```
kind: EKSCluster
metadata:
  name: crossplane-prod-cluster
spec:
  parameters:
    region: us-west-2
    vpc-name: "crossplane-vpc-only"
    vpc-cidrBlock: "10.20.0.0/16"

    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"
```

It will take about 10 minutes for all the resources to be provisioned. Once it is done, running the `kubectl get crossplane` command will show a list of all Crossplane-managed resources provisioned.

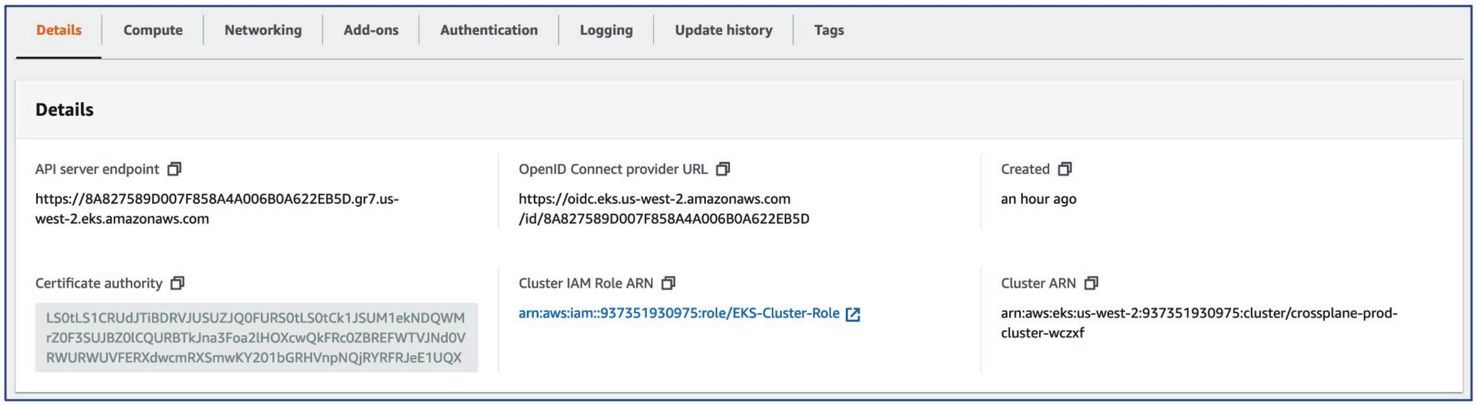
The following are the corresponding AWS-managed resources appearing on the AWS Management Console.



Name	Subnet ID	VPC	IPv4 CIDR
crossplane-vpc-only-private-worker-1	subnet-081c3e84419c95947	vpc-0b1b502231b1a301b | crossplane-vpc-only	10.20.11.0/28
crossplane-vpc-only-private-worker-2	subnet-0f2916c303e67c53c	vpc-0b1b502231b1a301b | crossplane-vpc-only	10.20.12.0/28
crossplane-vpc-only-public-worker-1	subnet-063d41f0bf949148b	vpc-0b1b502231b1a301b | crossplane-vpc-only	10.20.1.0/28
crossplane-vpc-only-public-worker-2	subnet-0eea3db89ba2df8a1	vpc-0b1b502231b1a301b | crossplane-vpc-only	10.20.2.0/28

Name	Route table ID	Explicit subnet associations
–	rtb-01251666be425d84e	–
crossplane-vpc-only-private-route-table-1	rtb-05fcc212e82e5cf55	subnet-081c3e84419c95947 / crossplane-vpc-only-private-worker-1
crossplane-vpc-only-private-route-table-2	rtb-015ad985db1207f86	subnet-0f2916c303e67c53c / crossplane-vpc-only-private-worker-2
crossplane-vpc-only-public-route-table	rtb-0e0be8023f16847b7	2 subnets

Name	NAT gateway ID	Connectivity type	Elastic IP address
crossplane-vpc-only-nat-gateway-1	nat-01683970aa39c92c6	Public	35.86.57.76
crossplane-vpc-only-nat-gateway-2	nat-0058f676c02a1c1a6	Public	54.212.149.24



Details | Compute | Networking | Add-ons | Authentication | Logging | Update history | Tags

Details

API server endpoint
https://8A827589D007F858A4A006B0A622EB5D.gr7.us-west-2.eks.amazonaws.com

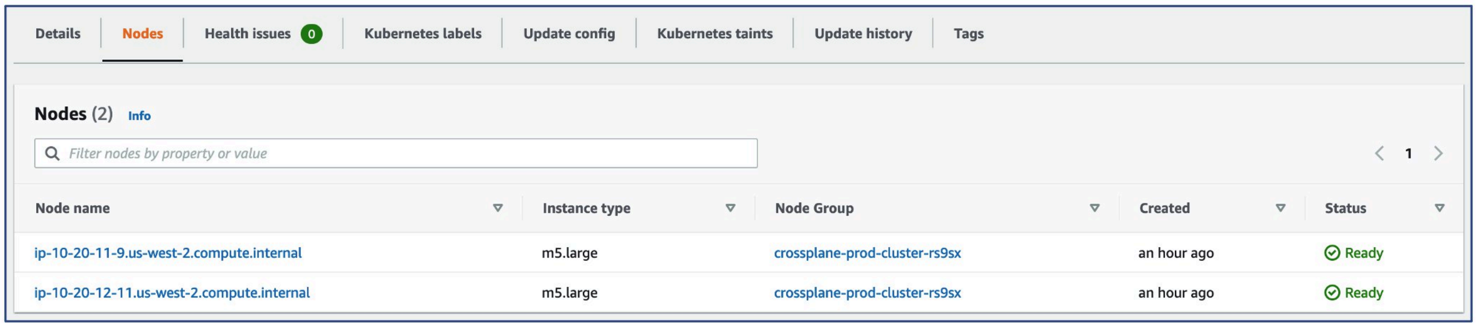
OpenID Connect provider URL
https://oidc.eks.us-west-2.amazonaws.com/id/8A827589D007F858A4A006B0A622EB5D

Created
an hour ago

Certificate authority
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM1ekNDQWM
rZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0ZBREFWTVJNd0V
RWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRJeE1UQX

Cluster IAM Role ARN
arn:aws:iam::937351930975:role/EKS-Cluster-Role

Cluster ARN
arn:aws:eks:us-west-2:937351930975:cluster/crossplane-prod-cluster-wczxf



Details | Nodes | Health issues 0 | Kubernetes labels | Update config | Kubernetes taints | Update history | Tags

Nodes (2) Info

Filter nodes by property or value

Node name	Instance type	Node Group	Created	Status
ip-10-20-11-9.us-west-2.compute.internal	m5.large	crossplane-prod-cluster-rs9sx	an hour ago	Ready
ip-10-20-12-11.us-west-2.compute.internal	m5.large	crossplane-prod-cluster-rs9sx	an hour ago	Ready

GitOps deployment with Argo CD

So far, we have looked at an imperative approach to cluster provisioning using Crossplane, which involved the following sequence of steps.

1. Deploy the core Crossplane controller as well as CRDs under the *.crossplane.io
2. Deploy Crossplane AWS provider-specific controller as well as CRDs under the *.aws.crossplane.io
3. Configure the provider with IAM credentials
4. Build an OCI image that represents a Crossplane configuration package that comprises the CompositeResourceDefinition and Composition types needed for provisioning an 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 an XR that triggers the provisioning of an EKS cluster.

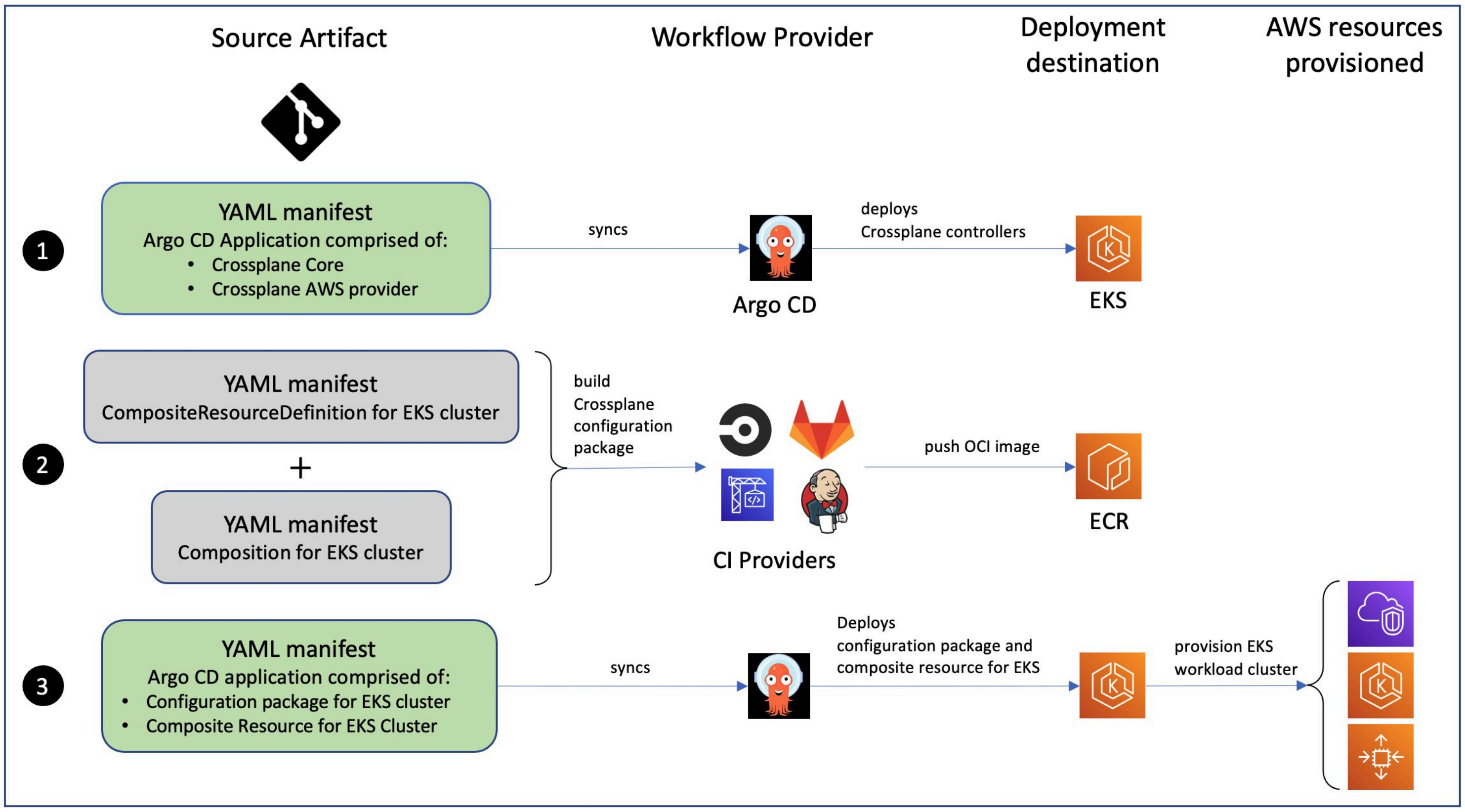
Next, let's take a look at how this may be done using a declarative approach with Argo CD. I will assume that you are familiar with the core concepts of implementing a Continuous Deployment (CD) workflow using Argo CD.

It follows the GitOps pattern of using Git repositories as the source of truth for defining the desired state of a cluster. Argo CD is installed in the management cluster and is configured to use the manifests in the eks-gitops-crossplane-argocd Git repository as the single source of truth to synchronize the state of both management and workload clusters.

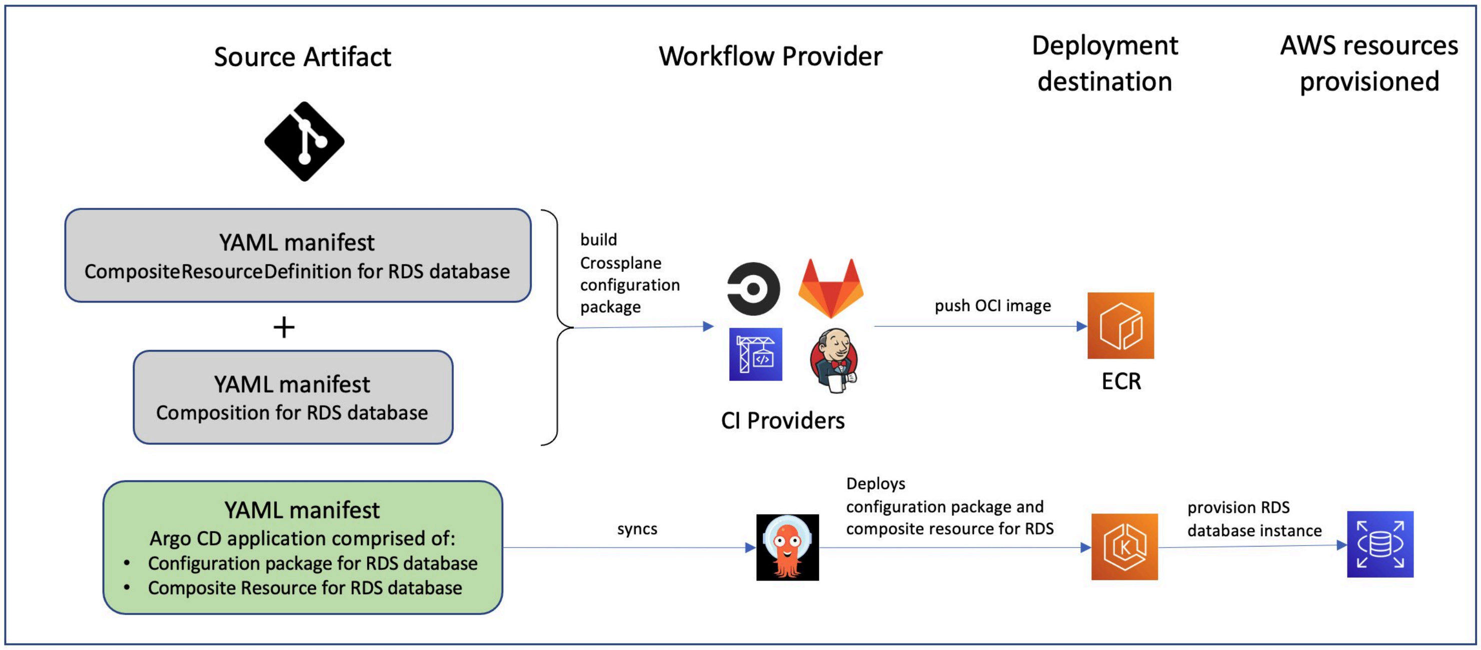
The steps involved in the installation and initial setup of Argo CD are outlined in this script. I will elaborate on some of the implementation details of leveraging the GitOps workflow in Argo CD to manage both cluster provisioning with Crossplane as well as remote cluster management.

- Workloads are deployed to a cluster by Argo CD using the Application custom resource, which represents a deployed application defined by one or more manifests. All Crossplane components required for executing Steps 1–6 above are packaged as a single Application whose destination is set to the management cluster. It deploys the core Crossplane and AWS provider-specific resources using a combination of a Helm chart and a set of manifests. Note that I have simplified the GitOps workflow for this implementation by assuming that the OCI image for the Crossplane Configuration package already resides in a registry. If you are starting off from a clean slate, then this workflow will have to be broken up into multiple segments, as shown in the following illustration.

  The first segment will be a GitOps workflow that deploys core Crossplane and the AWS provider (Steps 1–3). This needs to be done only once. This will be followed by a CI workflow to build the OCI image for the Configuration package and push it to a registry (Step 4). Any CI workflow provider may be used in this step. The final segment will be a GitOps workflow that deploys the Configuration package and the XR to the management cluster (Steps 5–6).


Source Artifact
Workflow Provider
Deployment destination
AWS resources provisioned
1
YAML manifest
Argo CD Application comprised of:
• Crossplane Core
• Crossplane AWS provider
syncs
Argo CD
deploys Crossplane controllers
EKS
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
Argo CD application 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

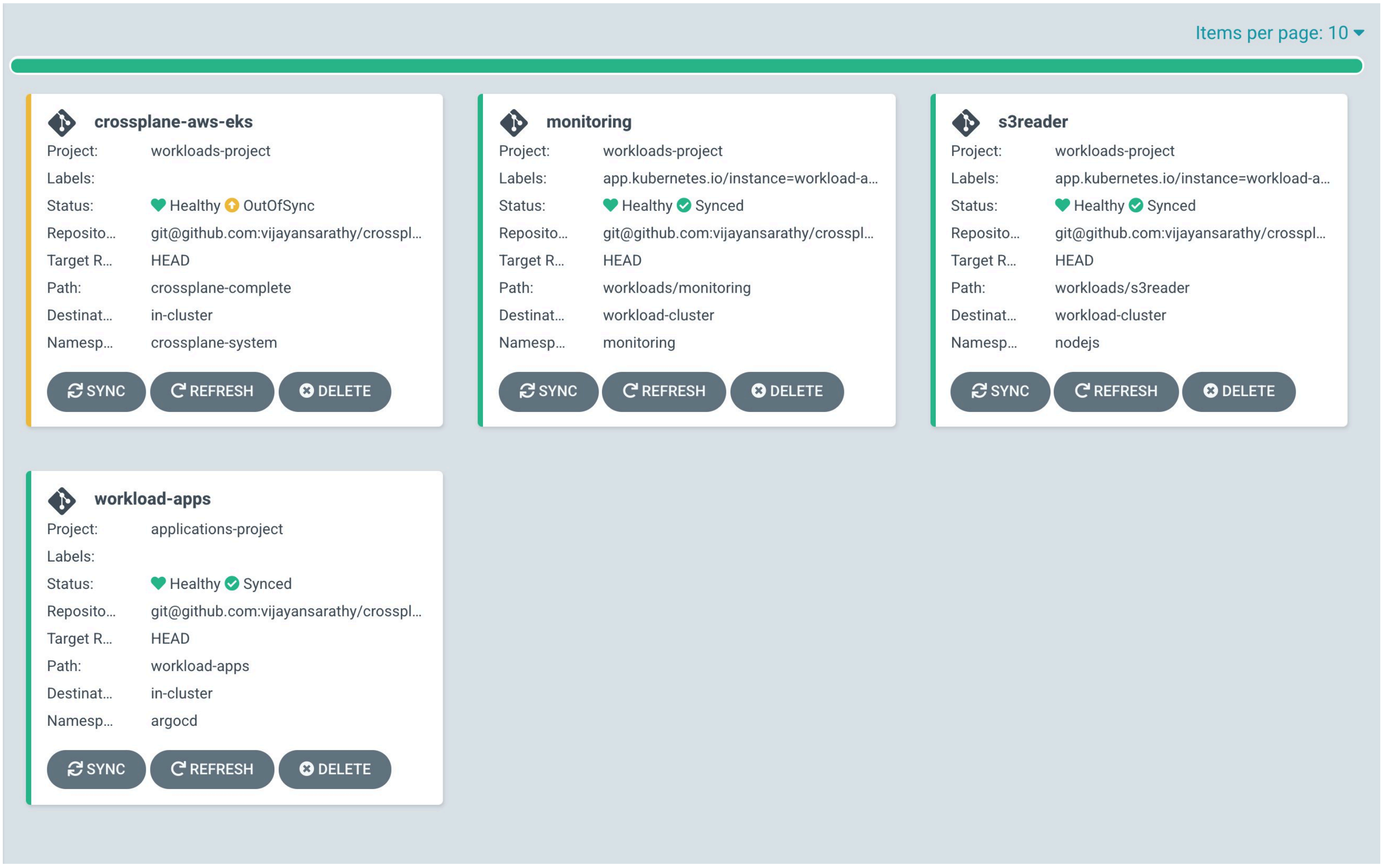
Using this approach to provision other AWS-managed resources such as, say, an RDS database instance, merely requires repeating Steps 2 and 3, as shown in the following illustration.


Source Artifact
Workflow Provider
Deployment destination
AWS resources provisioned
YAML manifest
CompositeResourceDefinition for RDS database
+
YAML manifest
Composition for RDS database
build Crossplane configuration package
CI Providers
push OCI image
ECR
YAML manifest
Argo CD application 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

- When deploying custom resources, Argo CD will perform a dry run to verify whether the CRDs that define those custom resources are known to the cluster. In this implementation, we make use of several Crossplane custom resources that are part of the same Application which contains the CRDs that define them. Hence, these CRDs will not be available in the cluster when Argo CD performs the dry run. Argo CD provides a workaround to circumvent this issue that is used in this implementation.
- Yet another issue is that the deployment of custom resources should be delayed until all the CRDs that define them have been deployed to the cluster and are ready. Argo CD provides two techniques, namely, Resource Hooks and Sync Phases and Waves, that allow you to ensure certain resources are healthy before subsequent resources are synced. The current implementation leverages both of them, as seen in this manifest, which uses a hook to pause the deployment pipeline for 10 seconds after the Crossplane AWS provider is deployed.
- When using GitOps workflow to deploy resources such as the Crossplane AWS provider that needs AWS credentials, we will have to tackle the issue of storing these credentials in a secure manner in a Git repository. The approach taken in this implementation is to use Bitnami's Sealed Secrets. The Kubernetes Secret, which contains the AWS credentials, is encrypted into a SealedSecret custom resource which is safe to store, even to a public repository. The encryption keys or sealing keys are securely stored in AWS Secrets Manager and are deployed to the cluster outside the GitOps workflow so that they are readily available to the controller that decrypts the SealedSecret.
- After Crossplane has successfully provisioned the workload cluster, we will have to register this cluster with the Argo CD installation in the management cluster. The steps to do this are documented here. The `argocd-manager` ServiceAccount on the workload cluster that is used by Argo CD to perform its tasks is granted cluster level admin privileges. It can be restricted to a narrower list of namespaces and actions by modifying the `argocd-manager-role` ClusterRole and the `argocd-manager-role-binding` ClusterRoleBinding, which binds the former to the ServiceAccount. Argo CD also has the concept of a Project that each Application belongs to, and this can be leveraged to implement access control. A project can specify the list of Git repositories that an Application can source its manifests from as well as the set of destination namespaces and clusters it can deploy to. The project settings used for this implementation are seen here.

- For bootstrapping the workload cluster with a set of applications, I use the App of Apps pattern. The app of apps manifest used in this implementation packages two applications to be deployed to the workload cluster: a Node.js web service deployed as a replica set with a Deployment manifest and Prometheus and Grafana monitoring tools deployed with a Helm chart.

The following figure shows the Argo CD user interface displaying all the Applications deployed to the management cluster and the workload cluster.



crossplane-aws-eks

Project:	workloads-project
Labels:	
Status:	Healthy OutOfSync
Reposito...	git@github.com:vijayansarathy/crosspl...
Target R...	HEAD
Path:	crossplane-complete
Destinat...	in-cluster
Namesp...	crossplane-system

monitoring

Project:	workloads-project
Labels:	app.kubernetes.io/instance=workload-a...
Status:	Healthy Synced
Reposito...	git@github.com:vijayansarathy/crosspl...
Target R...	HEAD
Path:	workloads/monitoring
Destinat...	workload-cluster
Namesp...	monitoring

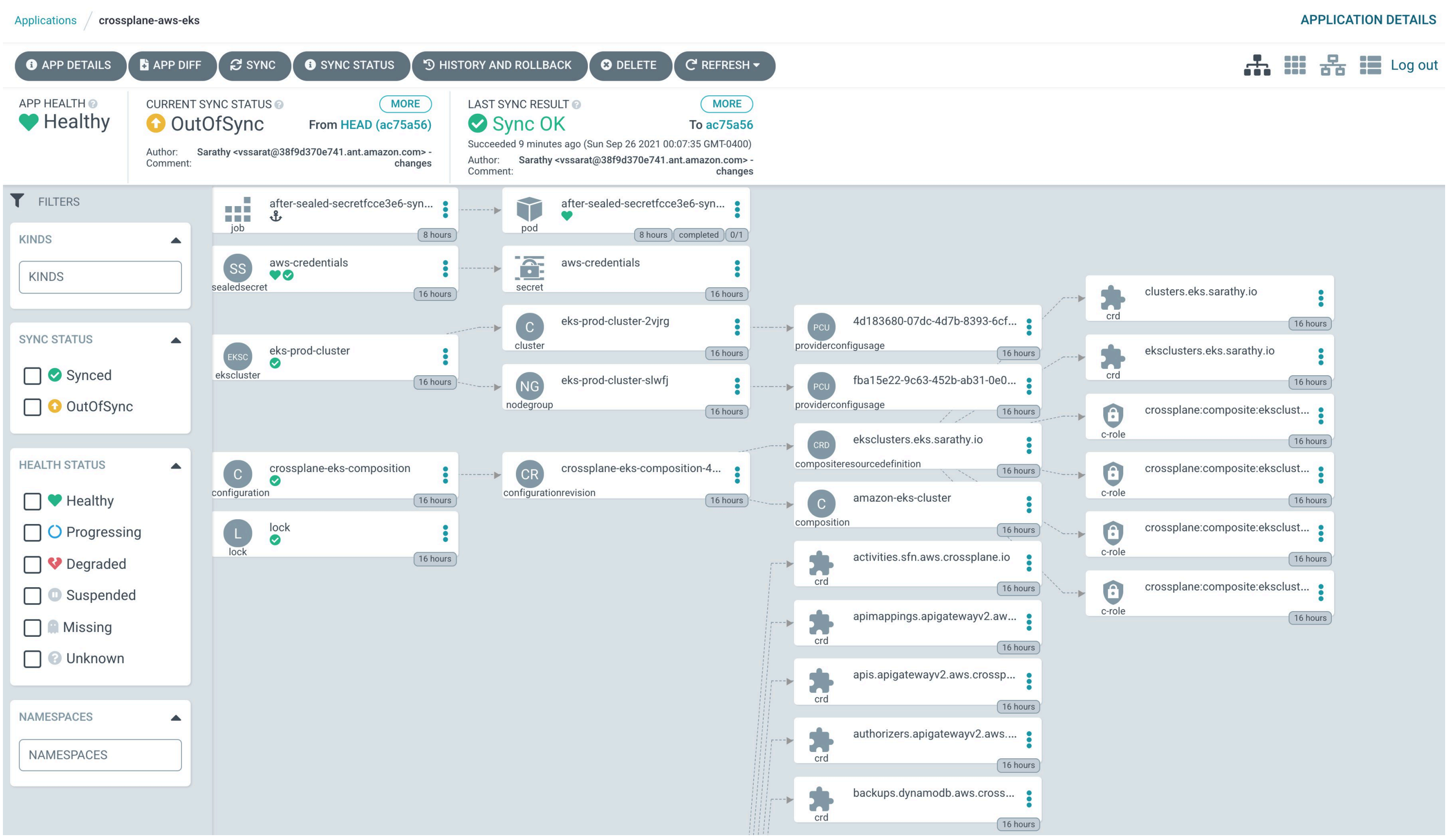
s3reader

Project:	workloads-project
Labels:	app.kubernetes.io/instance=workload-a...
Status:	Healthy Synced
Reposito...	git@github.com:vijayansarathy/crosspl...
Target R...	HEAD
Path:	workloads/s3reader
Destinat...	workload-cluster
Namesp...	nodejs

workload-apps

Project:	applications-project
Labels:	
Status:	Healthy Synced
Reposito...	git@github.com:vijayansarathy/crosspl...
Target R...	HEAD
Path:	workload-apps
Destinat...	in-cluster
Namesp...	argocd

Drilling down into the `crossplane-aws-eks` Application deployed to the management cluster, we see the deployment status of various CRDs in Crossplane's AWS provider as well as that of the EKSCluster Composite Resource.

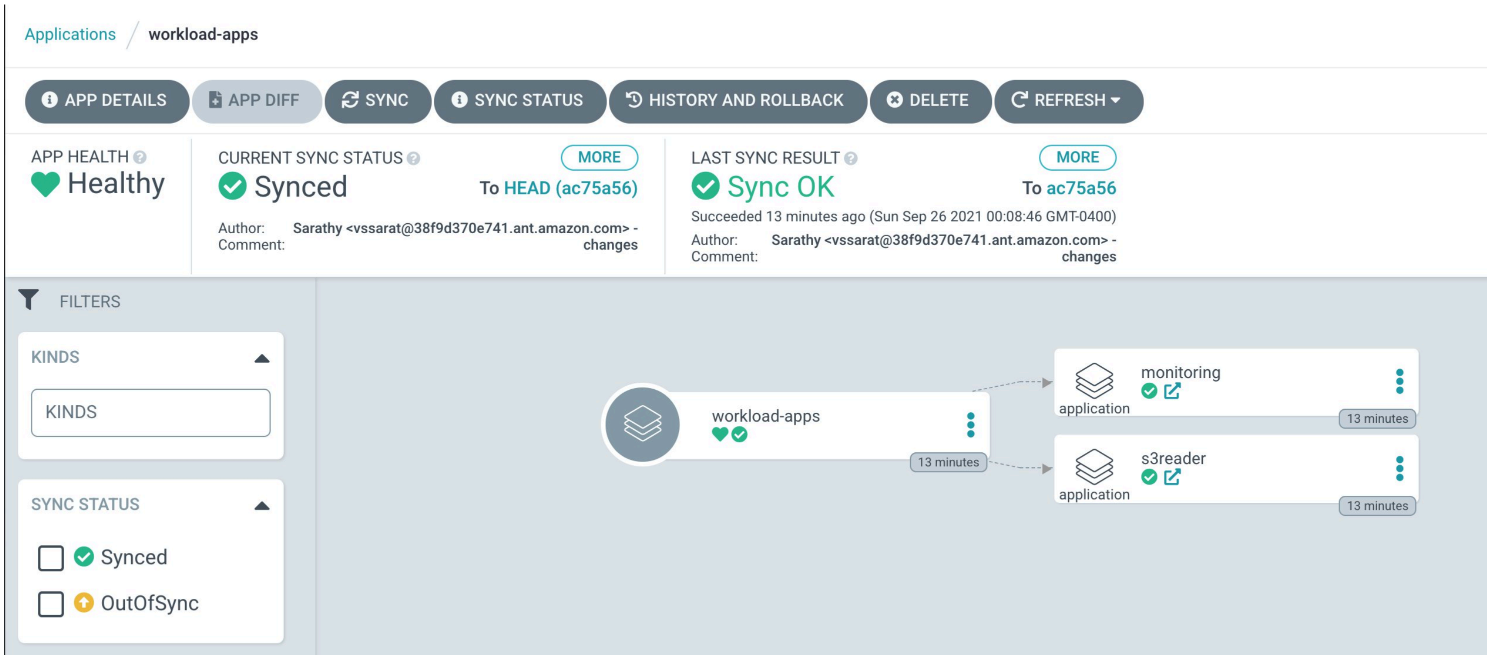

Applications / crossplane-aws-eks
APPLICATION DETAILS
APP DETAILS
APP DIFF
SYNC
SYNC STATUS
HISTORY AND ROLLBACK
DELETE
REFRESH
Log out
FILTERS
KINDS
SYNC STATUS
Synced
OutOfSync
HEALTH STATUS
Healthy
Progressing
Degraded
Suspended
Missing
Unknown
NAMESPACES
after-sealed-secretfcce3e6-syn... job 8 hours
after-sealed-secretfcce3e6-syn... pod 8 hours completed 0/1
aws-credentials sealedsecret 16 hours
aws-credentials secret 16 hours
eks-prod-cluster ekscluster 16 hours
eks-prod-cluster-2vjrg cluster 16 hours
eks-prod-cluster-slwfj nodegroup 16 hours
4d183680-07dc-4d7b-8393-6cf... providerconfigusage 16 hours
fba15e22-9c63-452b-ab31-0e0... providerconfigusage 16 hours
eksclusters.eks.sarathy.io compositeresourcedefinition 16 hours
amazon-eks-cluster composition 16 hours
crossplane-eks-composition configuration 16 hours
crossplane-eks-composition-4... configurationrevision 16 hours
lock lock 16 hours
activities.sfn.aws.crossplane.io crd 16 hours
apimappings.apigatewayv2.aw... crd 16 hours
apis.apigatewayv2.aws.crossp... crd 16 hours
authorizers.apigatewayv2.aws.... crd 16 hours
backups.dynamodb.aws.cross... crd 16 hours
clusters.eks.sarathy.io crd 16 hours
eksclusters.eks.sarathy.io crd 16 hours
crossplane:composite:eksclust... c-role 16 hours
crossplane:composite:eksclust... c-role 16 hours
crossplane:composite:eksclust... c-role 16 hours
crossplane:composite:eksclust... c-role 16 hours

APP HEALTH
Healthy

CURRENT SYNC STATUS MORE
OutOfSync From HEAD (ac75a56)
Author: Sarathy <vssarat@38f9d370e741.ant.amazon.com> -
Comment: changes

LAST SYNC RESULT MORE
Sync OK To ac75a56
Succeeded 9 minutes ago (Sun Sep 26 2021 00:07:35 GMT-0400)
Author: Sarathy <vssarat@38f9d370e741.ant.amazon.com> -
Comment: changes

Drilling down into the `workload-apps` App of Apps deployed to the workload cluster, we see the deployment status of the two Applications that it comprises.



APP HEALTH
Healthy

CURRENT SYNC STATUS MORE
Synced To HEAD (ac75a56)
Author: Sarathy <vssarat@38f9d370e741.ant.amazon.com> -
Comment: changes

LAST SYNC RESULT MORE
Sync OK To ac75a56
Succeeded 13 minutes ago (Sun Sep 26 2021 00:08:46 GMT-0400)
Author: Sarathy <vssarat@38f9d370e741.ant.amazon.com> -
Comment: changes

Conclusion

The GitOps declarative model has emerged as an efficient and scalable strategy to manage multiple Kubernetes clusters spread across geographical regions and public cloud providers. Cluster operators and developers are also looking for an efficient strategy to employ the same declarative model to provision cloud provider-specific managed resources, such as an Amazon S3 bucket or an Amazon RDS instance, that the application workloads depend on.

One of the key managed resources is the Kubernetes cluster itself which could be built from a manifest similar to the one used for deploying application workloads to a cluster.

This blog post discussed various tools emerging in the space of provisioning cloud-provider specific managed resources using the GitOps modes. I then delved into the details of an opinionated approach to provisioning and bootstrapping EKS clusters using two open-source tools, Crossplane and Argo CD. There are other implementation choices already available, and as tools evolve and mature, there will be many more.

The choice of tools aside, user adoption of the GitOps model to assemble cloud infrastructure, as well as managing it along with the application workloads that depend on them, is a paradigm shift. This approach is, in my opinion, gathering momentum and is likely to soon become mainstream. In a future blog post, we will explore other implementation choices in this space.

TAGS: Argo, crossplane, Gitops, Kubernetes