

MLops

Make life easy



Common problems with scaling ML

Getting started

Mundane steps with lots of boilerplate

Iterating quickly

Slow redundant iterative development

Hard to experiment and compare different models/algorithms

Productionizing

Incorporating ML models into production backend and handing off to engineers is very manual and bottlenecked

Using GitOps to manage software

laC:

GitOps uses a [Git repository](#) as the single source of truth for infrastructure definitions. Git is an open source version control system that tracks code management changes, and a Git repository is a `.git` folder in a project that tracks all changes made to files in a project over time. [Infrastructure as code \(laC\)](#) is the practice of keeping all infrastructure configuration stored as code. The actual desired state may or may not be stored as code (e.g., number of replicas or pods).

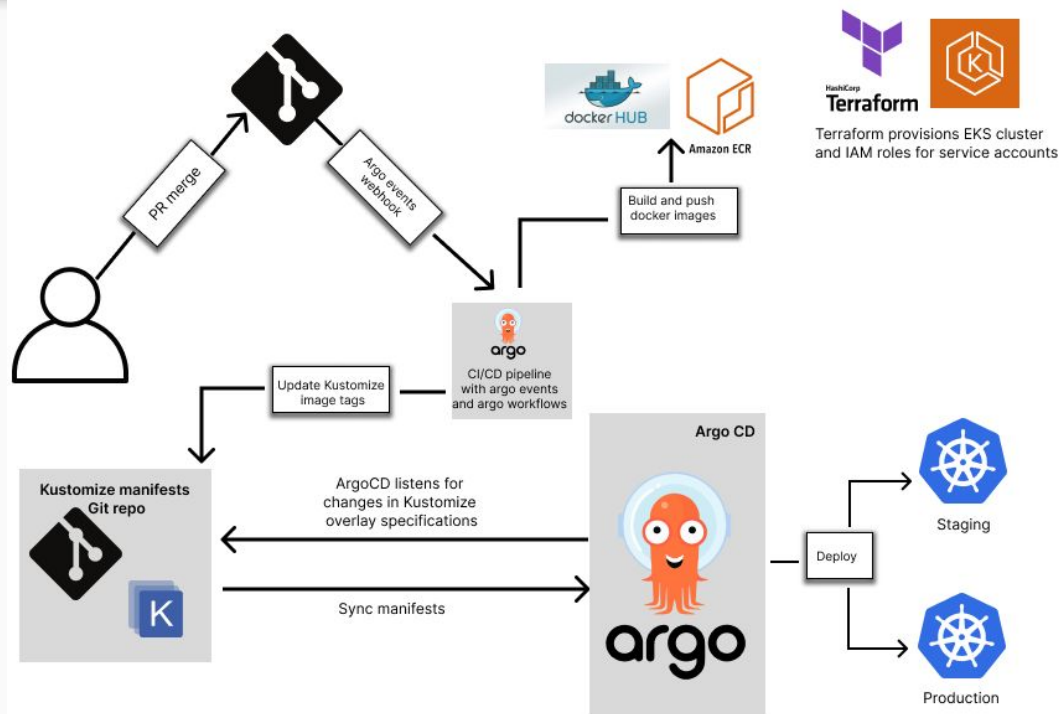
MRs:

GitOps uses merge requests (MRs) as the [change mechanism](#) for all infrastructure updates. The MR is where teams can collaborate via reviews and comments and where formal approvals take place. A merge commits to your main (or trunk) branch and serves as an audit log.

CI/CD:

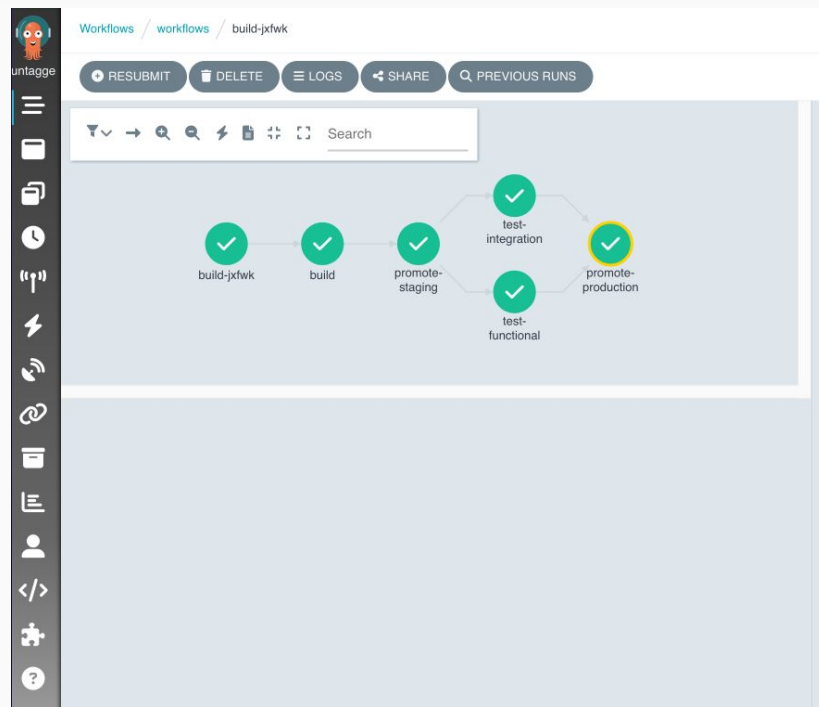
GitOps automates infrastructure updates using a Git workflow with [continuous integration \(CI\)](#) and continuous delivery (CI/CD). When new code is merged, the CI/CD pipeline enacts the change in the environment. Any configuration drift, such as manual changes or errors, is overwritten by GitOps automation so the environment converges on the desired state defined in Git. GitLab uses CI/CD [pipelines](#) to manage and implement GitOps automation, but other forms of automation, such as definitions operators, can be used as well.

Gitops CI/CD for any software product



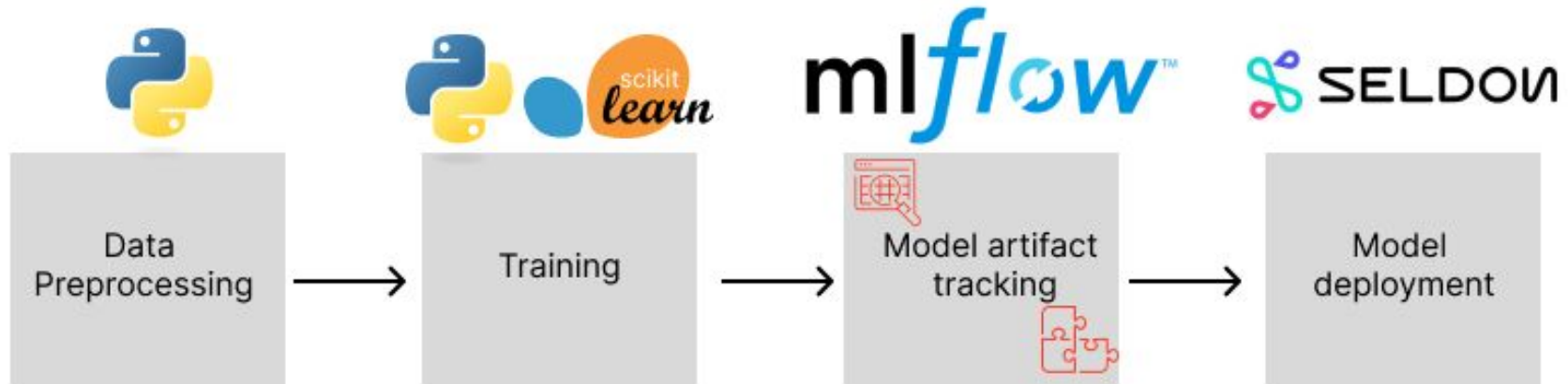
CI/CD using Argo workflows

1. Team makes change to MERN stack application and makes a Github PR
2. Github sends a webhook POST request to Argo event source
3. The CI/CD workflow trigger source is started
4. Docker image is built for frontend and backend and pushed to docker registry
5. New image tags are updated in Kustomization.yaml file
6. Argo CD picks this up and updates the deployment
7. Integration tests can be carried out in the staging environment to test that APIs endpoints can be reached and they return the expected results.
8. If all goes well, then we can update manifests in the production cluster



CI/CD for ML pipelines

Machine learning as composable units

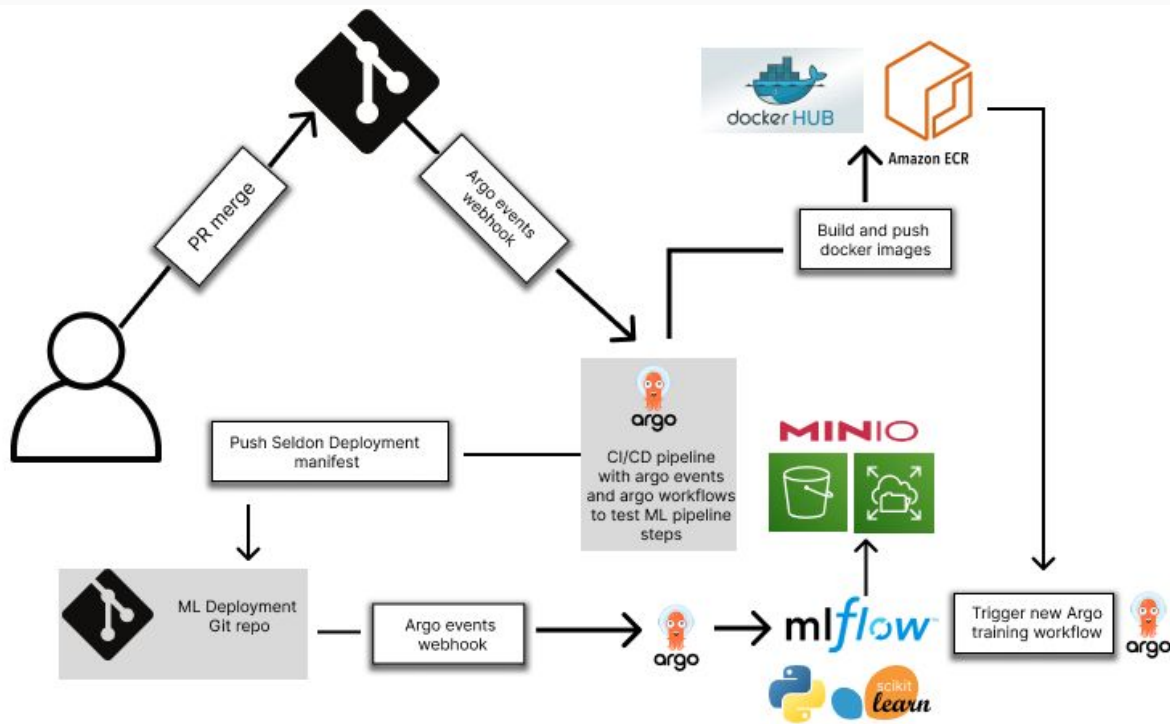


CI/CD for ML training pipeline

Training pipeline will store serialized model (.pkl file) to persistent NFS volume. It will also register the model with MLflow so we always know the properties of the model being used in production.

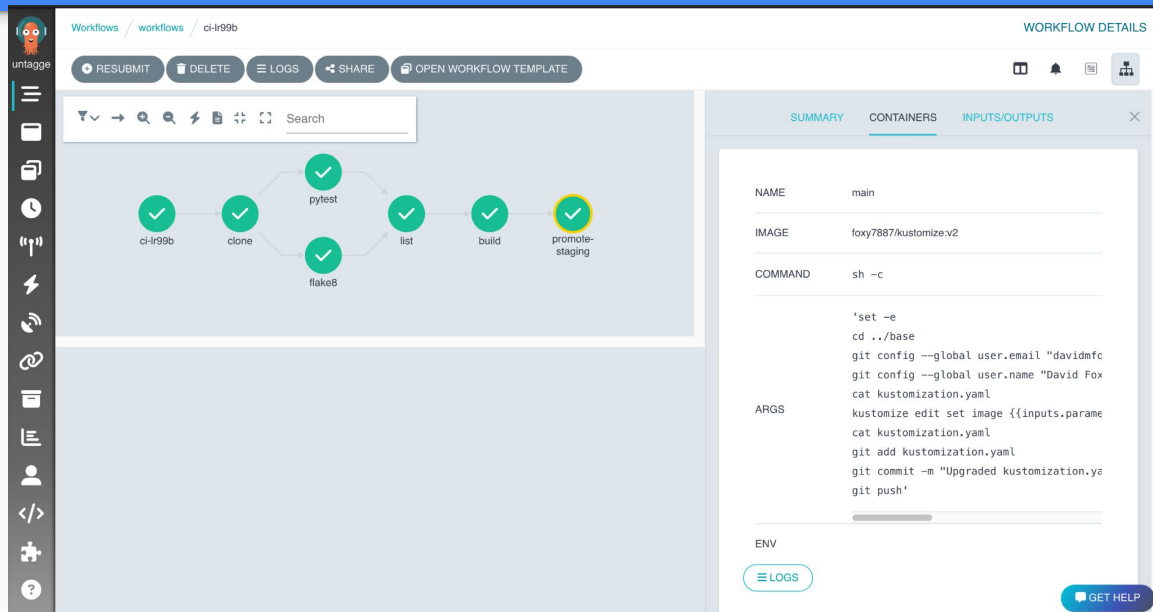
MLflow artifact storage will use minio locally and s3 in the cloud.

The training pipeline will be an Argo CronWorkflow, which can be run on some prespecified schedule.



CI/CD for Machine Learning

1. Once a model has been experimented and it has found to outperform the model in production, it is then modularized into ML pipeline steps, which are later containerized. Each pipeline step should be unit tested.
2. The next steps would be:
 - a. Build docker image with build-kit
 - b. Update image tags in staging
3. This ensures that the argo training workflow will use the most up-to-date image for the training pipeline steps



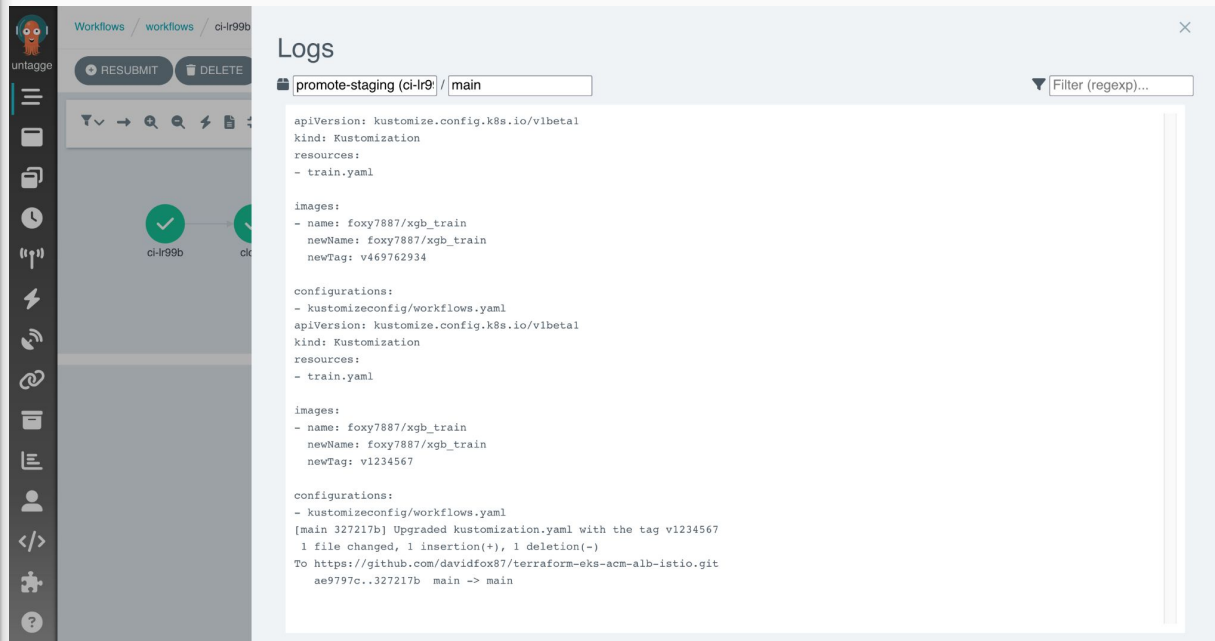
CI/CD for Machine Learning

We can observe the docker logs from the promote to staging step of the DAG.

This shows what the image tag read in the kustomization.yaml before and after the following command was run:

```
kustomize edit set image  
{{inputs.parameters.image}}=  
{{inputs.parameters.image}}:  
{{inputs.parameters.tag}}
```

Argo CD will see that the kustomize file has changed and apply our updated CronWorkflow training pipeline to the cluster.



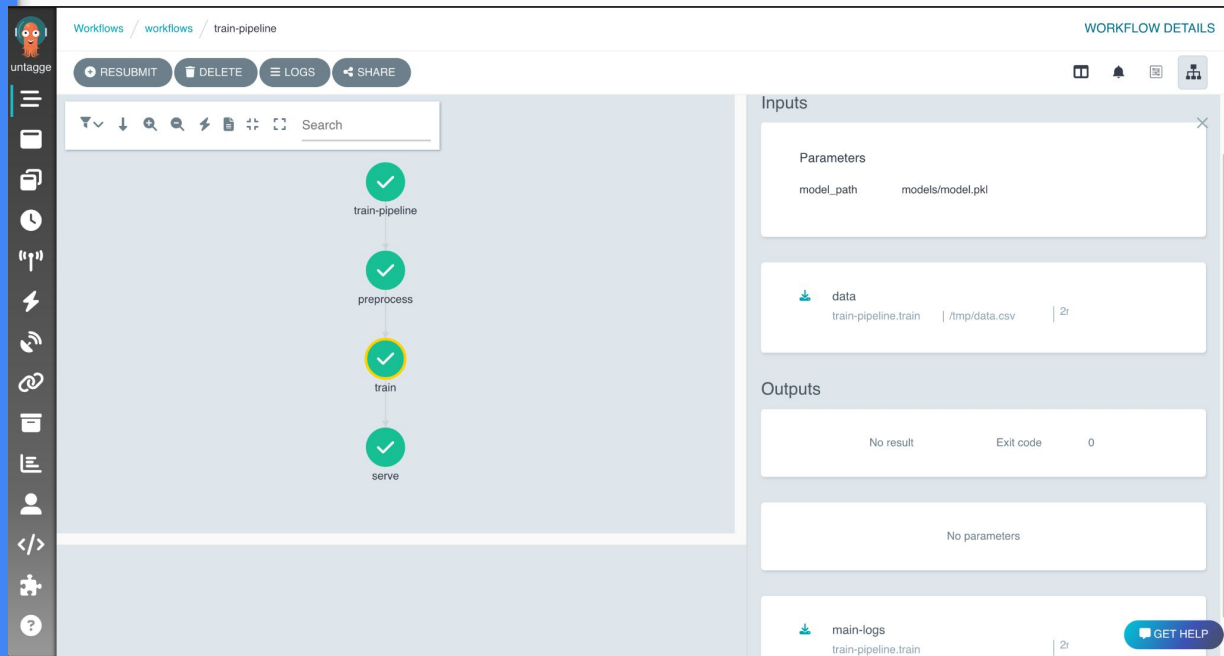
The screenshot displays the Argo CD web interface. On the left, a sidebar shows navigation icons and a workflow diagram with two steps, 'ci-lr99b' and 'ci-lr99c', both marked with green checkmarks. The main panel is titled 'Logs' and shows the logs for the 'promote-staging (ci-lr9)' workflow on the 'main' branch. The logs contain the following content:

```
apiVersion: kustomize.config.k8s.io/v1beta1  
kind: Kustomization  
resources:  
- train.yaml  
  
images:  
- name: foxy7887/xgb_train  
  newName: foxy7887/xgb_train  
  newTag: v469762934  
  
configurations:  
- kustomizeconfig/workflows.yaml  
apiVersion: kustomize.config.k8s.io/v1beta1  
kind: Kustomization  
resources:  
- train.yaml  
  
images:  
- name: foxy7887/xgb_train  
  newName: foxy7887/xgb_train  
  newTag: v1234567  
  
configurations:  
- kustomizeconfig/workflows.yaml  
[main 327217b] Upgraded kustomization.yaml with the tag v1234567  
1 file changed, 1 insertion(+), 1 deletion(-)  
To https://github.com/davidfox87/terraform-eks-acm-alb-istio.git  
ae9797c..327217b  main -> main
```

Training and serving pipeline

1. Loads the data from Blob storage (minio, s3) but can also fetch from SQL tables
2. Preprocess can relabel columns, engineer features, impute missing data, scale features
3. Train will train an ML model
 - a. Logs and registers the model with MLflow
 - b. Save serialized pickle to NFS persistent storage
4. Serve will use seldon-core library to deploy the model in the NFS persistent storage as a prediction RESTful microservice.

Requests to the prediction microservice are routed through the istio- gateway and service mesh (next slide).



Making predictions

From “outside” the cluster (Note, that in the cloud, we will reach istio-ingressgateway through an ingress)

port-forward to istio-ingressgateway service:

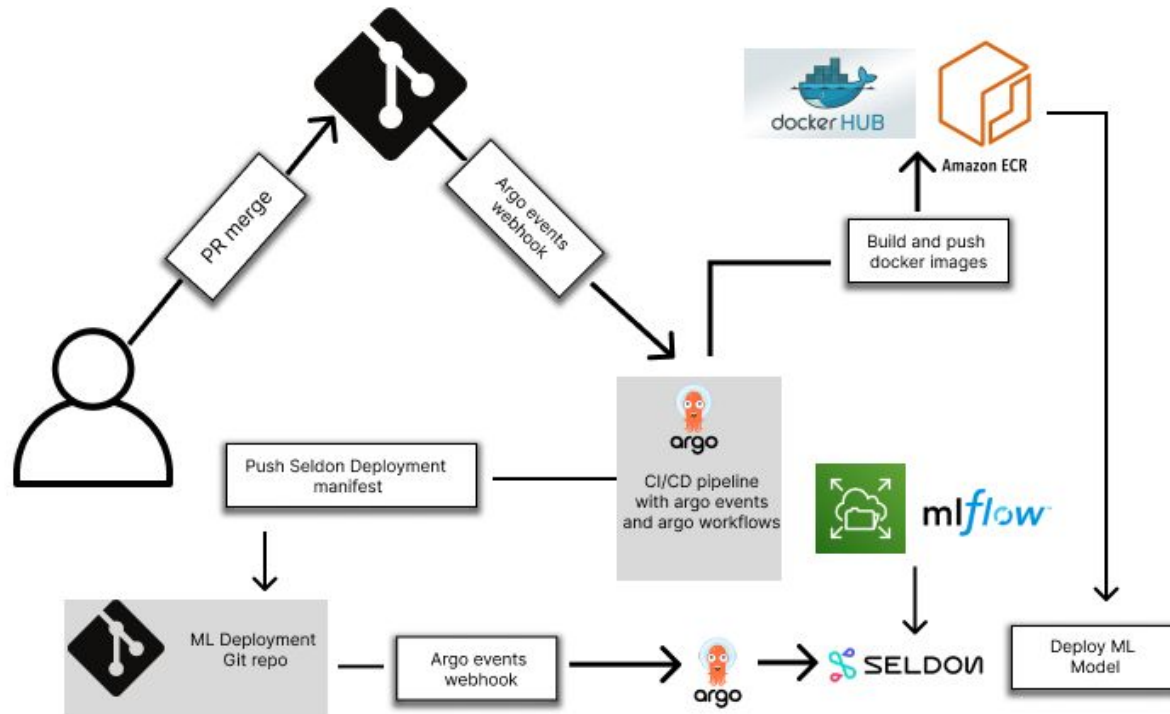
```
kubectl port-forward $(kubectl get pods -l istio=ingressgateway -n istio-system -o jsonpath='{.items[0].metadata.name}') -n istio-system 8004:8080
```

```
curl -X POST -H 'Content-Type: application/json' \
  -d '{"data": { "ndarray": [[1,2,3,4,5]]}}' \
http://localhost:8080/seldon/<NAMESPACE>/<SELDON-DEPLOYMENT-NAME>/api/v1.0/predictions
```

Internal to the cluster from any other kubernetes deployment (Deployment, Job, Pod):

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -d '{"data": { "ndarray": [[1,2,3,4,5]]}}' \
http://<SELDON-DEPLOYMENT-NAME>.<NAMESPACE>.svc.cluster.local:8000/api/v1.0/predictions
```

CICD of the inference pipeline



Making this work in the cloud

Problem

Devops is hard...it takes time to master. AWS has 200+ services!

Not everybody can be an expert in everything but people should be able to consume cloud infra for testing and prototyping.

Can we enable everyone developing applications to serve their own infrastructure without having to learn all the complexities of AWS and Infrastructure as Code (IaC).

Goal

Create opinionated APIs that abstract away complexities of AWS service cloud primitives and enable application developers/data scientists to serve their own infra for prototyping solutions in the cloud.

A developer-friendly API

provide an API that developers can easily consume

provision a 20 GB Postgres instance that a platform team has defined, a developer would do a simple kubectl apply on

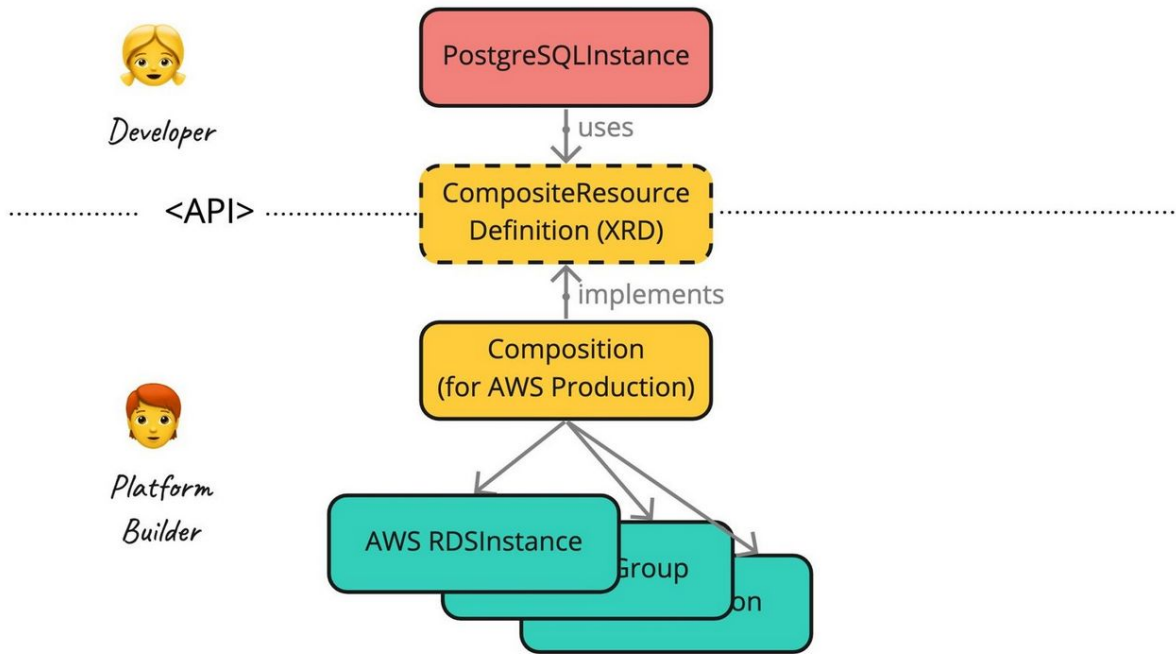
```
1  apiVersion: database.example.org/v1alpha1
2  kind: PostgreSQLInstance
3  metadata:
4    name: my-db
5    namespace: default
6  spec:
7    parameters:
8      storageGB: 20
9    compositionSelector:
10     matchLabels:
11       provider: aws
12       environment: production
13   writeConnectionSecretToRef:
14     name: db-conn
```

Compositions

The implementation behind this API can be rather complex and may involve setting up the right cloud provider resources like permissions, networking, VPCs, and database instances

The basic building block in Crossplane is a Managed Resource from a cloud provider.

These basic building blocks (the Managed Resources in green) are stitched together as Crossplane Compositions (in yellow) and provide the implementation to the API provided to the developers.



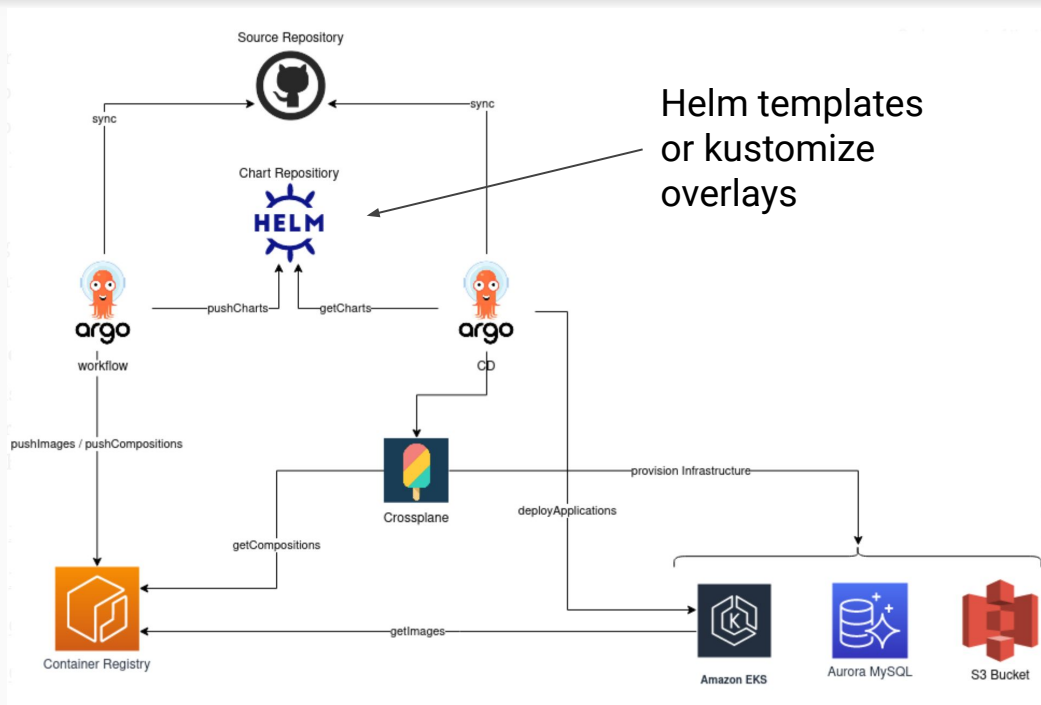
Crossplane and Argo CD

Deploy infrastructure using the Kubernetes API. No need to switch to completely different environment (Terraform/Pulumi).

If infrastructure is defined declaratively with K8s manifests, then we can use Kustomize and Argo CD to sync any changes to the cluster and crossplane will make sure that the state as defined in the manifests is reflected in our AWS environment.



Crossplane and Argo CD



Demo

Provision an AWS RDS postgres database by consuming our app developer API

