GitOps With GitHub Actions and EKS

Guest post by Jeremy Adams - [@jpadamspdx](https://twitter.com/jpadamspdx) and John Bohannon - [@imjohnbo](https://twitter.com/imjohnbo)

Hi! We’re Jeremy Adams and John Bohannon from the GitHub Business Development team. Like the best haircuts, our team is a bit of business in the front (like Jeremy) and party, err engineering, in the back (like John). In reality our engineering teammates all enjoy working closely with our ecosystem partners on strategic projects. Many of us “business types” also like getting our hands on keyboard, which makes working in BD a blast.  
  
When we started our latest project, our goal was simple to state: A [Weave GitOps workflow](https://www.weave.works/technologies/gitops/) with [GitHub Actions](https://github.com/features/actions) and [Amazon EKS](https://aws.amazon.com/eks/).

A picture containing text, clipart, sign

Description automatically generated

Now where to start? When you want to deeply understand how various tools fit together into a more complex chain or system, you can just read about it, but we prefer to actually build it; starting with a simple working sketch of a system that can be extended into something more real-world. The goal is to ultimately create something that adds value to your organization or to the community at large, but friction and unnecessary complexity at this early stage are far from helpful.

In that spirit, we present this end-to-end technical walkthrough where you’ll get the hands-on experience of:

* setting up an EKS cluster with Weaveworks **eksctl**
* using a GitHub Actions workflow to build and push a new container image to [AWS ECR](https://aws.amazon.com/ecr/) upon code change
* deployment to EKS and commit of that new image tag back to GitHub with Weave Flux

Basically something like this:

Diagram

Description automatically generated

**Step 1: Setup AWS user, initial CLI tools, ECR**

Okay, we’ve got an outline and a diagram, but we’re going to need to get some tools that work together to make this happen. You’ll need [GitHub](https://www.github.com/) and [AWS accounts](https://aws.amazon.com/) for this exercise. We’re going to assume you have basic proficiency in **git**, Linux shell, basic container/Kubernetes concepts, and can get an AWS account, and IAM policy yourself or through your friendly neighborhood IT admin.

There are great resources for **ekstctl** from [Weaveworks](https://www.weave.works/blog/automate-eks-cluster-configuration-with-gitops-and-eksctl" \t "_blank) and [AWS](https://aws.amazon.com/blogs/opensource/eksctl-eks-cli/), but I’ll describe my exact process so you can follow along.

1. Add a user with the perms needed for **eksctl** and Flux to create an EKS cluster and push/pull the image in ECR. We created a new IAM Policy called “eksctl-policy” and used [this IAM Policy JSON](https://gist.github.com/jpadams/d1b076a6308346d581bc362360a858d0) with our AWS Account ID filled in.
2. Create a new IAM user with Programmatic Access. For permissions choose “Attach existing policies directly”. We then attached two permissions in order to create/destroy EKS clusters with eksctl and to create ECR repos and push/pull :  
     
   - Our custom “**eksctl-policy**”  
   - The AWS-managed “**AmazonEC2ContainerRegistryPowerUser**” policy
3. Check that the [aws CLI](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html" \t "_blank), [eksctl](https://eksctl.io/introduction/installation/" \t "_blank), and [fluxctl](https://docs.fluxcd.io/en/stable/references/fluxctl.html" \t "_blank) are installed locally to your workstation. Collect your user credential information and run aws configure to set your credentials and the AWS region. **NOTE:** Ensure that you have sufficient Elastic IPs and VPCs in the AWS region you select (five by default per region). I decided to simply change to a region that [had EKS available](https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/) and that wasn’t being used heavily by my organization.
4. Create an [Amazon Elastic Container Registry](https://docs.aws.amazon.com/AmazonECR/latest/userguide/Registries.html) (ECR) repository called “example-eks” in the same AWS region you’ll create the EKS cluster. I used the [AWS console to create the repo](https://docs.aws.amazon.com/AmazonECR/latest/userguide/repository-create.html). The ECR is where we’ll push our container images to and for Weave Flux (more about Flux in a moment) to deploy them from.

A picture containing graphical user interface

Description automatically generated

**Step 2: Create the EKS cluster**

Now that we have a user with the right permissions, our CLI tools, and an ECR repo. Let’s create the EKS cluster! We love to use **eksctl** for this, because it handles all of the tricky details of setup (and you can run eksctl delete cluster when you’re done). There are lots of eksctl options, but it can be as simple as:

$ eksctl create cluster  
  
It will take about 10-20 minutes to get your new cluster.

... [✔] saved kubeconfig as "/Users//.kube/config"

... [✔] EKS cluster "improbable-name-9999999999" in "eu-west-1" region is ready  
  
Since there is a new **kubeconfig** in place, when you run kubectl from your desktop, you’ll be interacting with your new EKS cluster! The generated cluster name is in the last line of output, but if you forget, just run one of these commands.  
  
$ eksctl get clusters  
$ aws eks list-clusters

**Step 3: Setup our GitHub Actions Workflow repo**

We’re going to use GitHub Actions to drive our container build CI from a workflow file in the GitHub repo.  
To set up yours:

1. Fork the example repo into your GitHub user account. <https://github.com/github-developer/example-actions-flux-eks>
2. Navigate to the**.github** directory in the GitHub web interface Code tab. Click on the **Actions** tab and enable GitHub Actions workflow in this forked repo.

**Error! Filename not specified.**

Graphical user interface, text, application

Description automatically generated

  3. View the workflow in your repo under[**.github/workflows/build.yml.**](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml) We’ll walk through the whole workflow, but let’s make sure you have it set up first, starting with the environment variables on lines 16-20.

16 AWS\_DEFAULT\_REGION: eu-west-1  
17 AWS\_DEFAULT\_OUTPUT: json  
18 AWS\_ACCOUNT\_ID: ${{ secrets.AWS\_ACCOUNT\_ID }}  
19 AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}  
20 AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}  
  
  
Ensure that line 16 has the region where your EKS cluster resides. This should match the region in your ~/.aws/config or the region specified on the eksctl command line.

**Step 4: Configure secrets**

Lines 18-20 in the build.yml file refer to secrets. Secrets are set up in the Settings tab, which are found at the bottom of the left navigation menu in the Secrets section.  
  
Create three secrets called AWS\_ACCOUNT\_ID, AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY with your AWS account ID (obtainable from the AWS console or by running aws sts get-caller-identity)and the new user’s key ID and the secret key (found in your **~/.aws/credentials**file). Secrets associated with your GitHub repo allow you to keep sensitive data out of your source code, but still make them available to GitHub Actions workflows.

Graphical user interface, text, application

Description automatically generated

**Step 5: App development workflow**

Now is a good time to take a look around the repo. Notice that the simple containerized web app we’re going to deploy is in the **app** directory, whereas the Kubernetes YAML files for the Deployment and Service are in the **manifests** directory. The development workflow that we’re going to employ is as follows:

1. Developers push changes to the app directory.
2. GitHub Actions then builds and pushes container images t[o Elastic Container Repository (ECR)](https://aws.amazon.com/ecr/), as specified by a workflow, [.github/workflows/build.yml](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml).
3. Flux notices a new image in ECR and deploys it to EKS by modifying the deployment manifest, and memorializing that change with a git commit back to [manifests/deployment.yml](https://github.com/github-developer/example-actions-flux-eks/blob/master/manifests/deployment.yml).

Speaking of ECR, remember the ECR repo we created called “example-eks”? That repo name will match up with the name used for the image repo:tag when we [build it with the GitHub Actions workflow](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml#L21).

Now we’re going to get GitHub Actions to push our first container image there!

**Step 6: Priming the pump for our first deployment**

Before turning over the continuous deployment duties of new container images to Flux, we need to push at least one successful initial deployment. We’ll make a change to our app that triggers our GitHub Actions workflow to build a new container image, tag it with the current git SHA, and push it to ECR. That image will then be used in our deployment.yml as a starting point, and Flux can take it from there.

**Commit a change and copy the image URI**

To kick off that build let’s make a simple CSS change that switches the text color from white on a black background to green on black. It’s easy to do this from the GitHub web interface, and can also be done on the command line.

Graphical user interface, text, application, email

Description automatically generated

Once you commit the change, switch over to the **Actions** tab to view the running workflow:

Graphical user interface, text, application, email

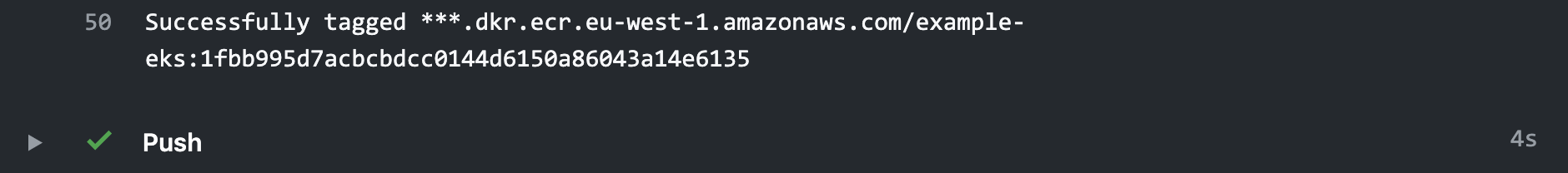
Description automatically generated

1.  Click on the workflow to view the steps.

Graphical user interface, text, application, website

Description automatically generated

2. Expand the Build and tag the image step, to view most of the new image repo and the tag on the last line.



3. Take a look at your “example-eks” repo in ECR and copy the full image URI.

Graphical user interface, text, application

Description automatically generated

4. Update manifests/deployment.yml with the [image URI](https://github.com/github-developer/example-actions-flux-eks/blob/master/manifests/deployment.yml#L22-L23) you just copied from the AWS console.

Now you’re ready to install Flux on the EKS cluster and have it perform its first sync!

**But first, a quick check-in. Almost there!**

Let’s take stock of where we are:

* We have our CLI tools, an EKS cluster, and an ECR repo.
* GitHub Actions workflow is pointed at the correct region and has the credentials it needs to push a new image to ECR in its secrets. The manifests/deployment.yml has the correct ECR repo noted as well (we just did that a moment ago).
* [Weave Flux](https://www.weave.works/oss/flux/) to watch for when GitHub Actions workflow builds and pushes a new container image to the ECR repo.

Okay, we’re ready. Let’s get Flux running on our cluster, and synchronize things for the first time. Then we’ll make a change to our app to get a new container image, and watch the magic happen.

**Step 7: Run Weave Flux on the EKS cluster**

We’re going to run Flux on our EKS cluster in its own “flux” namespace. We’ll use this short script (also in the README.md) to make it happen. Be sure to run it from the root of your **example-actions-flux-eks**repo. Notice that flux is watching the **manifests** folder for deployment YAML changes.

export GHUSER=

export GHREPO=example-actions-flux-eks

kubectl create ns flux

fluxctl install \

--git-user=${GHUSER} \

--git-email=${GHUSER}@users.noreply.github.com \

--git-url=git@github.com:${GHUSER}/${GHREPO} \

--git-path=manifests \

--namespace=flux | kubectl apply -f -

If all went well, you should have Flux running on your cluster.

serviceaccount/flux created

clusterrole.rbac.authorization.k8s.io/flux unchanged

clusterrolebinding.rbac.authorization.k8s.io/flux unchanged

deployment.apps/flux created

secret/flux-git-deploy created

deployment.apps/memcached created

service/memcached created

Check it by running:

  $ kubectl get all -n flux

Since we want Flux to be able to write back to our git repo the tag of the image it deploys, we need to give it [read/write access](https://docs.fluxcd.io/en/latest/tutorials/get-started.html#giving-write-access) to our repo with a Deploy Key. Run this command to get the Flux public key and add it.

$ fluxctl identity --k8s-fwd-ns flux

ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQC64WoWesnPneyDqq8ddTAAOKSaLHcu+0ALL8xxtGdnbK2WG99OZ7A9cq24Y9TmSL4gIuXb0HDvwhHsnbkTNsFmVWpO9xS/T3bqhLzhdQwLCGP21ckhRVF7RBv+pK6PnenY4ZjTRkW5h7SxYnunEarj/9E9NlL/JP8tDnb53liDXF4AB1y3Xi/nKwjlgwkGGrSBXGSRij7a6uq2iMlGF/H9MmHn8ct7w/dd/RF6VN4phbNpsVfnBVu1yDgRJTNKznXDOCEEAfflxAFrDWjbAsXwCxvWLNsbP5HtMTf5Ep/Eba7ZAjZ7XnWYLgoXRZHOf+0WYqn1EfsSot5pb01TFeYr

**NOTE**: Be sure to check the box to allow write access!

Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application, email

Description automatically generated

**Step 8: Sync the deployment and view the app!**

To get Flux and GitHub in sync for this first go-round, issue a manual Flux sync command. Flux immediately ensures that the state of objects in Kubernetes for which we have a YAML file match.

$ fluxctl sync --k8s-fwd-ns flux

Synchronizing with git@github.com:jpadams/example-actions-flux-eks

Revision of master to apply is 1fbb995

Waiting for 1fbb995 to be applied ...

Done.

Let’s find out where we can view our app running in the default Kubernetes namespace and browse to the DNS name of the ELB.

$ kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

aws-example-octodex LoadBalancer 10.100.64.193 80:31273/TCP 20h

You should be seeing an Octocat in green, spouting wisdom. Hurray!

Text

Description automatically generated

After this, we’re going on a sort of auto-pilot. Whenever we change our app code, GH Actions will build a new container image and push it to ECR. Flux will notice the new image, deploy it to the cluster, and then sync up the actual Kubernetes Deployment with the manifests/deployment.yml. From here on out, the state of the cluster will win. Let’s try that out.

**Step 9: Make a change to the app and push to master**

We’ll make a change to our app code again and push it to master, but this time we won’t do anything else but wait (for about five or six minutes).

Let’s change the CSS again to go from green to yellow this time.

Graphical user interface, text, application, email

Description automatically generated

Feel free to peek at the Actions tab and the AWS ECR console to see how things are going with the build and push of the new image. While we wait a few minutes for Flux to deploy the change, let’s read a little more about the workflow.

**More details on the GitHub Actions workflow**

You might have noticed that in the[.github/workflows/build.yml](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml#L2-L10) we configured the workflow to only run when there’s either a PR or a push to master that involves files under the app directory. Changes anywhere else in the repo’s file tree don’t trigger a workflow run. That’s a good thing because we don’t want Flux’s writes to the **manifests** directory to trigger another run post deployment (we could have also [ignored that directory path](https://help.github.com/en/github/automating-your-workflow-with-github-actions/workflow-syntax-for-github-actions#onpushpull_requestpaths)).

on:

pull\_request:

paths:

- app/\*\*

push:

paths:

- app/\*\*

branches: # array of glob patterns matching against refs/heads. Optional; defaults to all

- master # triggers on pushes that contain changes in master

**Building and tagging images**

The [middle of the workflow](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml#L12-L47)involves building and tagging (but not pushing quite yet) the container image to ECR. This will run for every PR or push to master. There are comments suggesting where you might want to insert linting, unit tests, or security scans. All good ideas, but beyond the scope of our basic example.

name: Build and Push to ECR

env:

AWS\_DEFAULT\_REGION: eu-west-1

AWS\_DEFAULT\_OUTPUT: json

AWS\_ACCOUNT\_ID: ${{ secrets.AWS\_ACCOUNT\_ID }}

AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

CONTAINER\_IMAGE: example-eks:${{ github.sha }}

jobs:

build-and-push:

name: Build and deploy

runs-on: ubuntu-latest

steps:

- name: Checkout

uses: actions/checkout@master

# Add steps here like linting, testing, minification, etc.

- name: Setup ECR

run: |

# Login to AWS ECR

$( aws ecr get-login --no-include-email )

- name: Build and tag the image

run: |

# Build and tag the image

docker build \

-t $CONTAINER\_IMAGE \

-t $GITHUB\_REPOSITORY:$GITHUB\_SHA \

-t $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_DEFAULT\_REGION.amazonaws.com/$CONTAINER\_IMAGE ./app

# Add additional steps here like scanning of image

**Pushing images to the container repo**

[The last step in the workflow](https://github.com/github-developer/example-actions-flux-eks/blob/master/.github/workflows/build.yml#L49-L54) is where the push to ECR occurs, but we’ve added an if condition to ensure that we only run that step if it’s a push to master, and not just a PR.

# Only push to registry on master

- name: Push

if: github.ref == 'refs/heads/master'

run: |

# Push image to AWS ECR

docker push $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_DEFAULT\_REGION.amazonaws.com/$CONTAINER\_IMAGE

After five minutes are up, refresh the app web page or check if a new pod has started in Kubernetes.

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

aws-example-octodex-5b87d5f56f-wdk8d 1/1 Running 0 62s

Diagram, text

Description automatically generated

**Flux, the GitOps operator**

Great that worked, but clearly I’m running a new container image with the “yellow” version. The last time I touched my deployment.yml, I had specified the old “green” version of my app’s container image. Better take a look!

Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application, Teams

Description automatically generated Graphical user interface, application

Description automatically generated