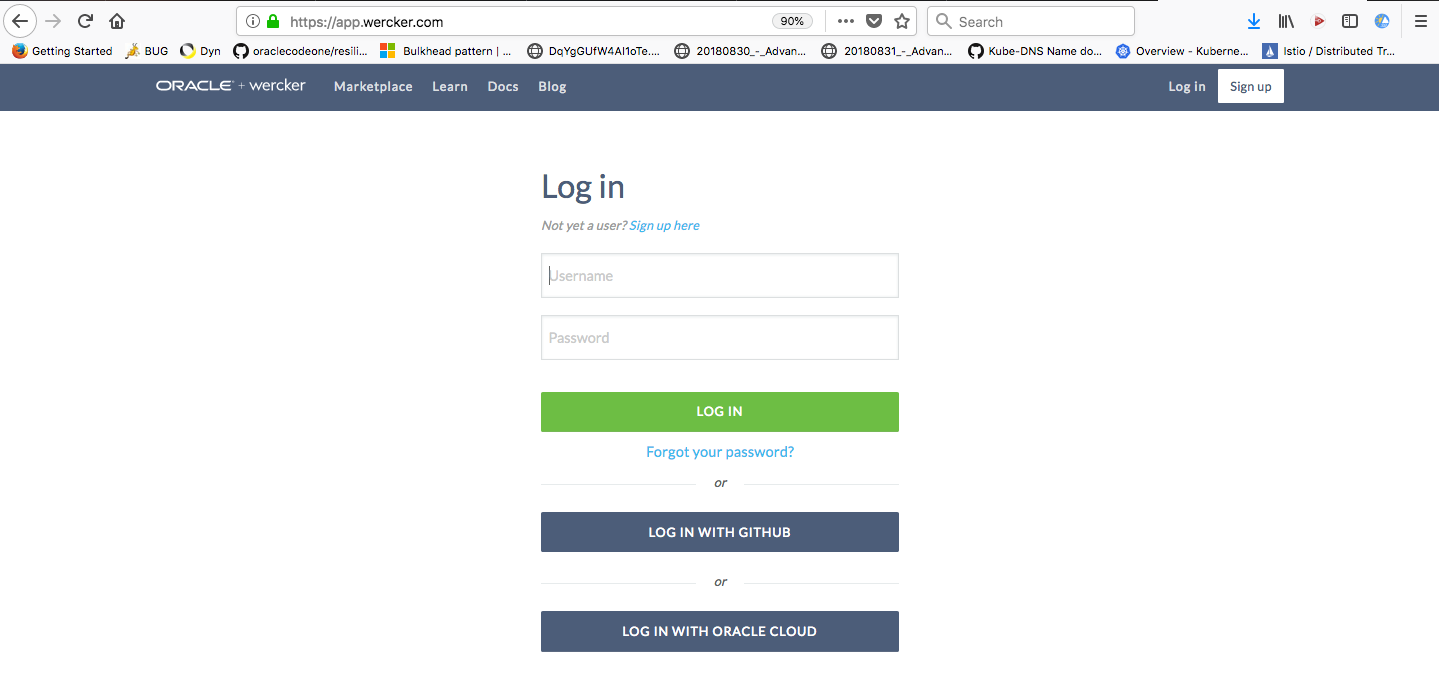
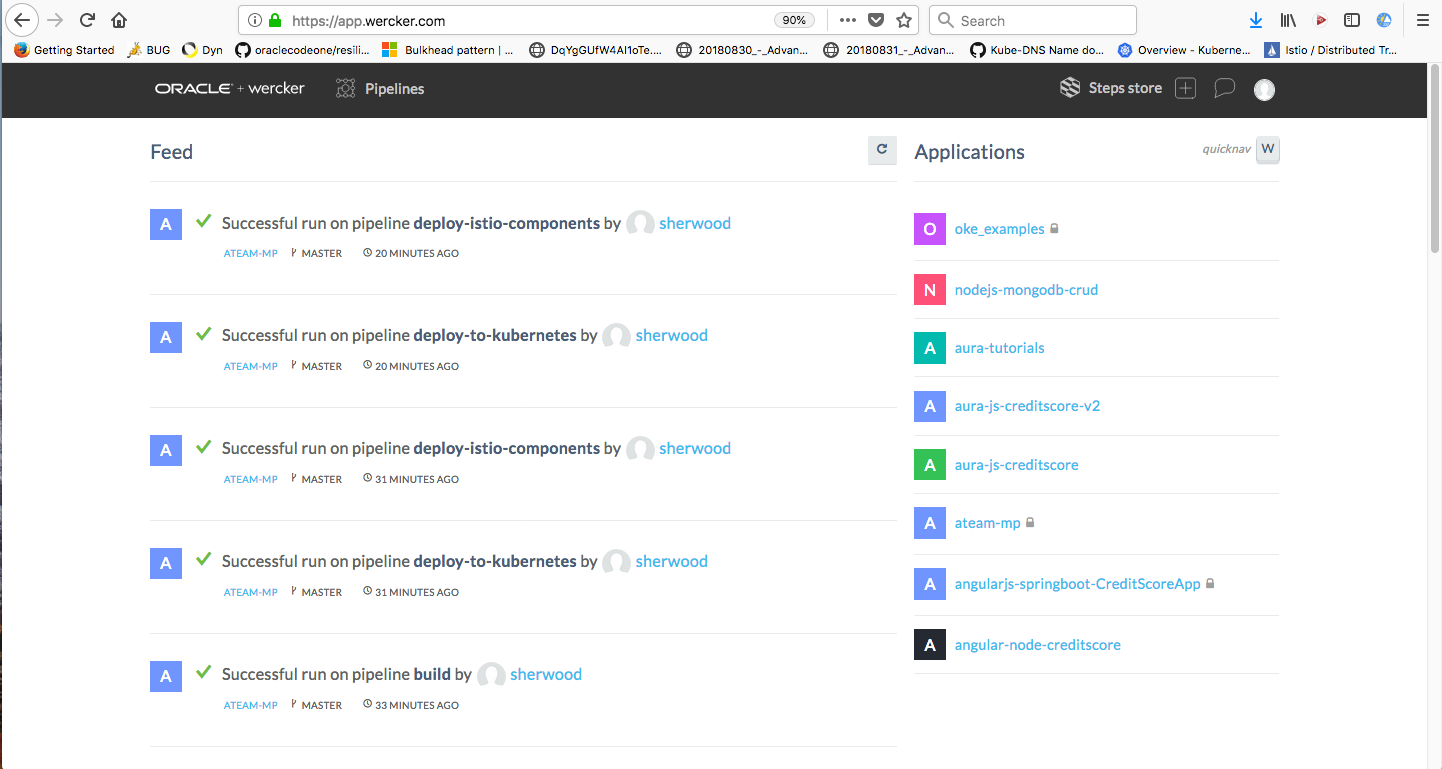
# CI/CD with Wercker

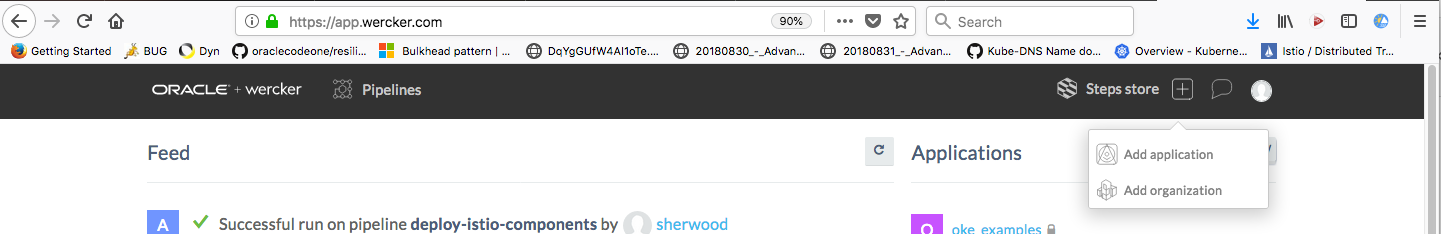
1. Access <http://app.wercker.com>



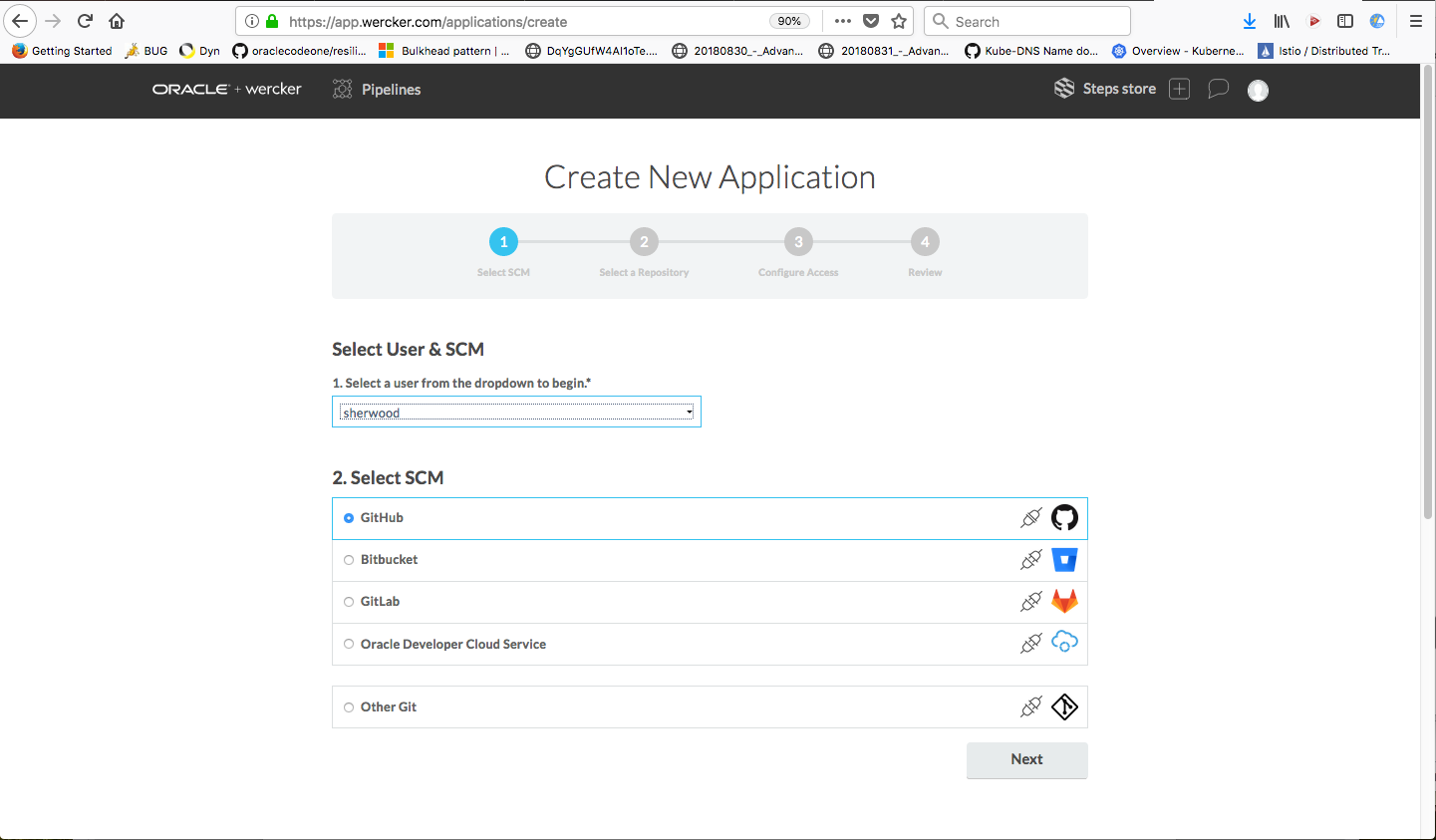
1. You are free to create a new account, login using GitHub, or Login with Oracle Cloud. Once logged in you will be presented the applications page. If you already an account and had applications created you will see those applications on the page.



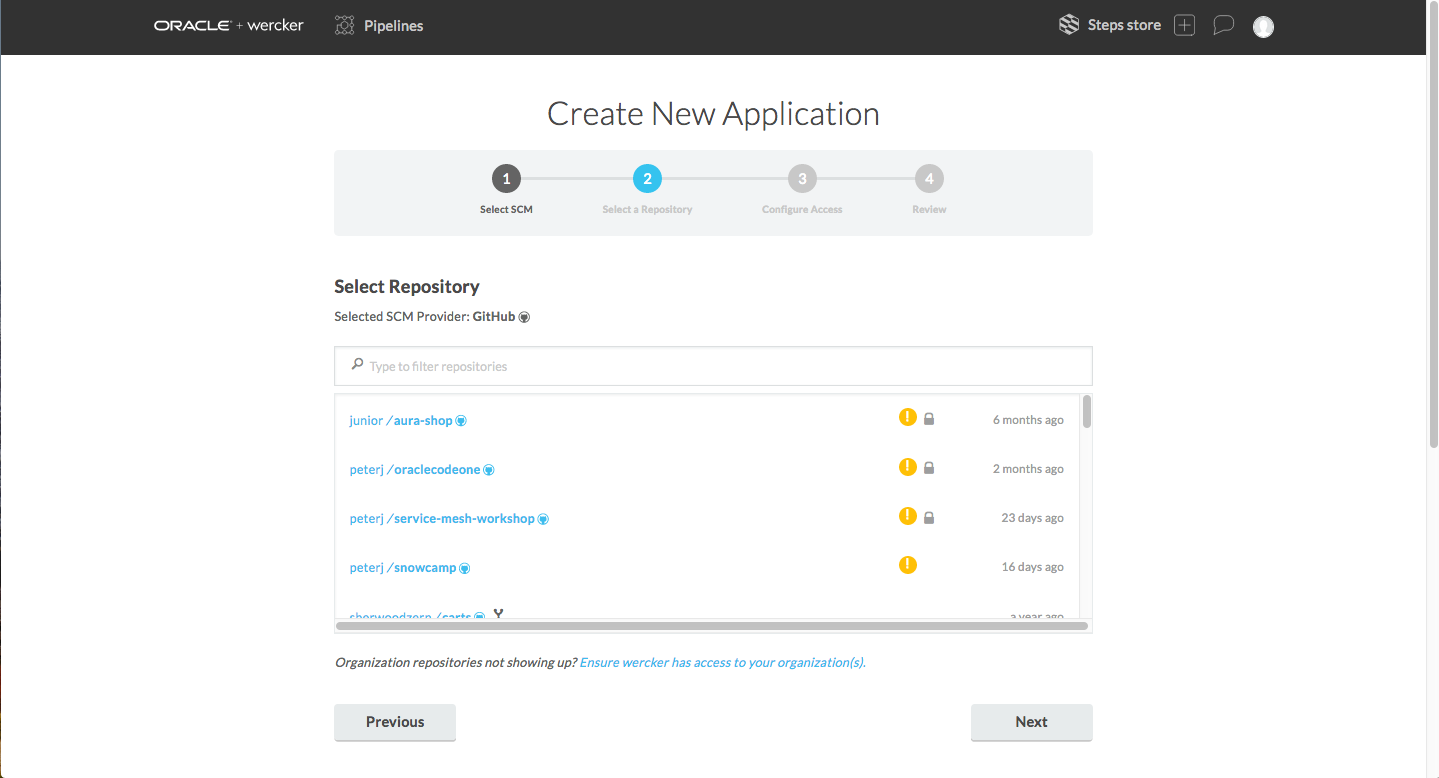
1. We are going to create a new application. The application will be our new Helidon application. Let’s create a new application. Click on the “+” icon in the upper right-hand corner of the applications page.



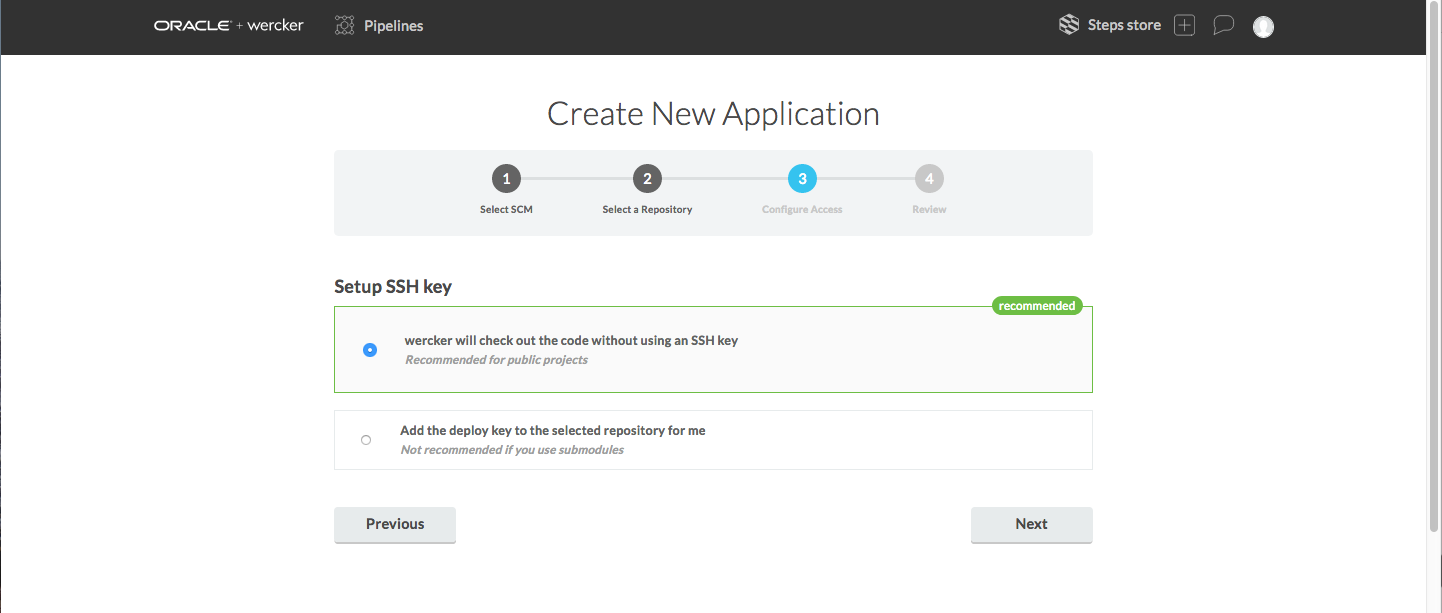
1. Click the “Add application”
2. When you click the add application you will be navigated to a wizard that walks you through the application creation process.



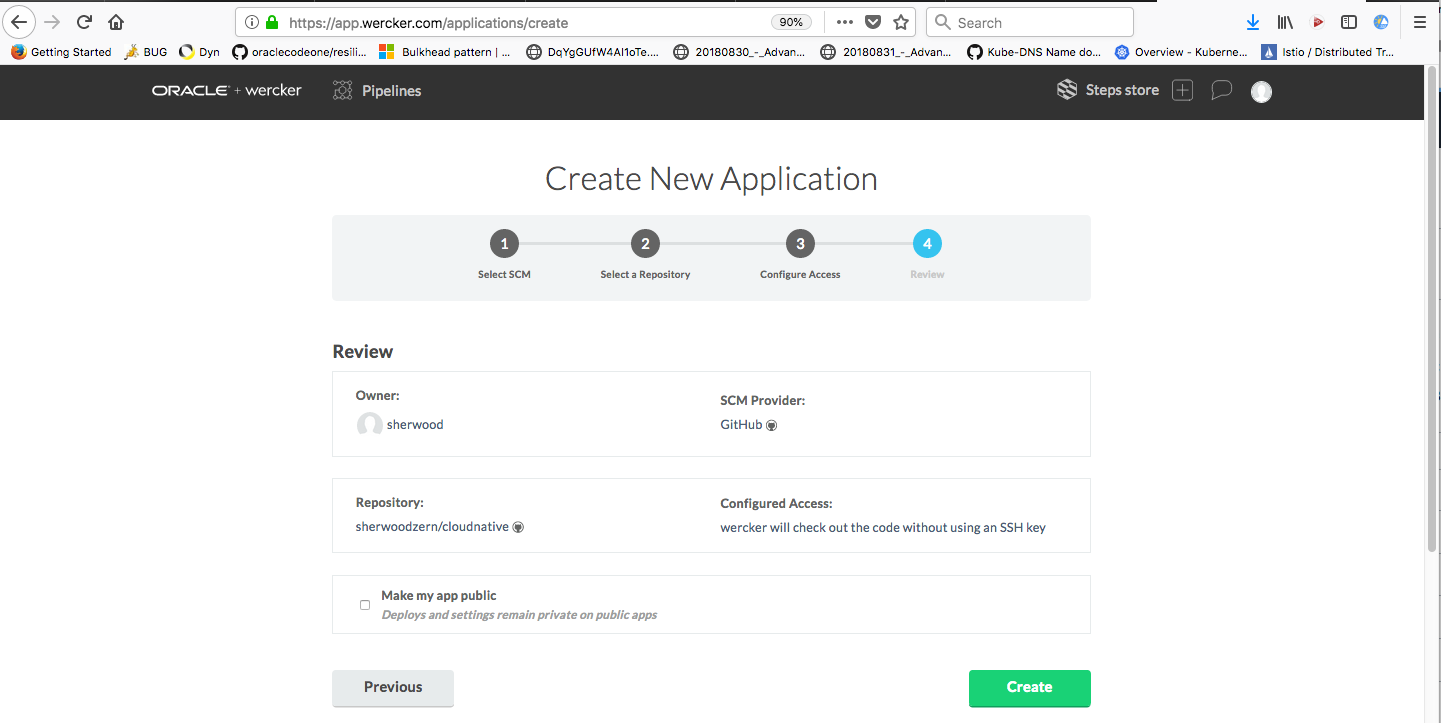
1. For “Select User & SCM” choose yourself.
2. For “Select SCM” choose one of the options. Everybody should be using Github; therefore, select “Github”
3. Select “Next”



1. The second step in the create application process is to select the application from your GitHub account. Select the Helidon application we recently uploaded. Once selected click the “Next” button.

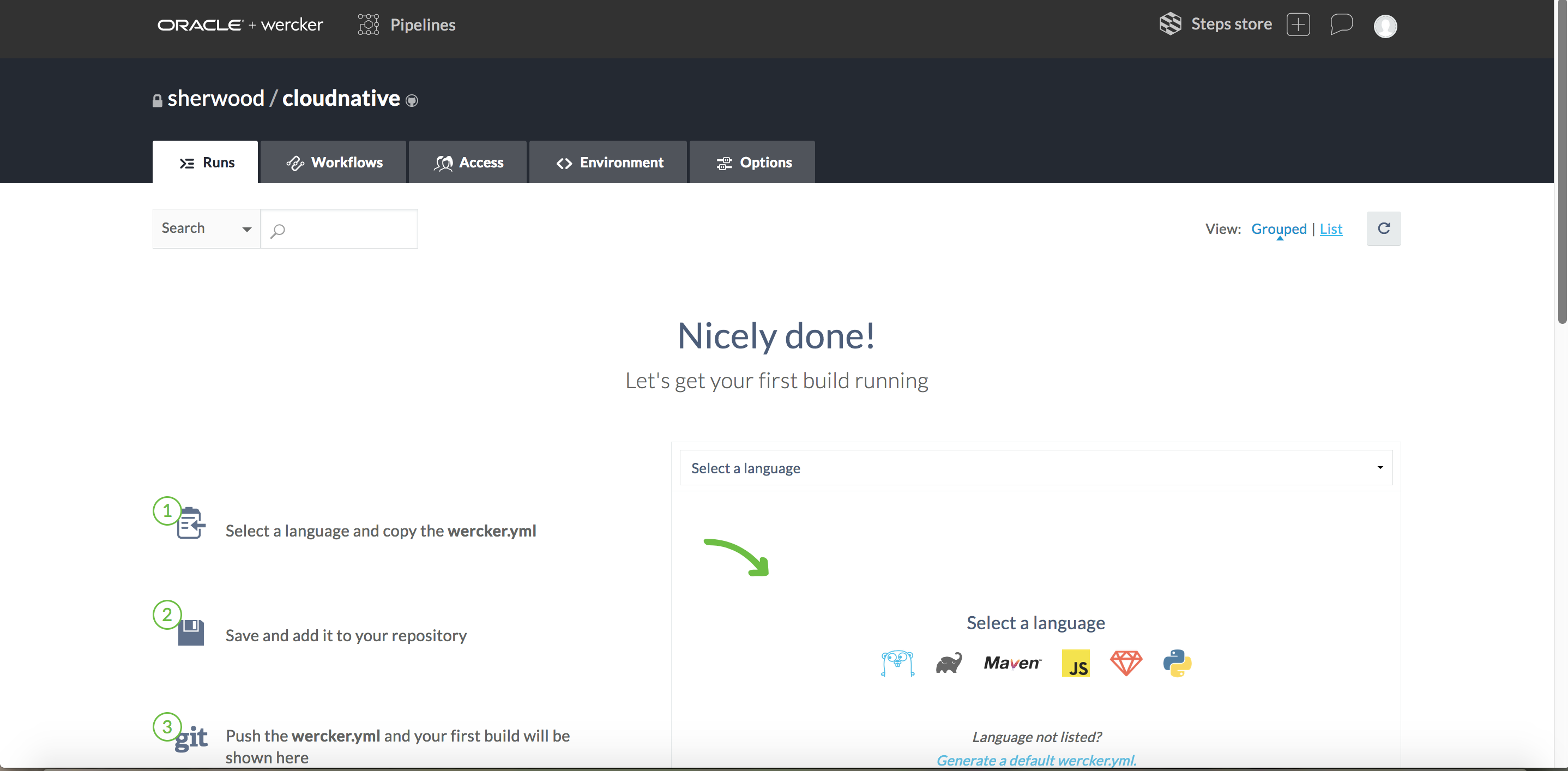


1. Allow Wercker to check out the code without using an SSH key. This is the default. Select the “Next” button.



* This is the summary page. If you’re happy with everything then select “Create”.

1. As soon as the application has been created in Wercker you will be presented with the following view. In this view, you will need to select the language of your application, create a default wercker.yml file, and add this wercker.yml file to your github repository.



You will be using Maven so select it. Once you select Maven a default wercker.yml file will be created. You need to add this wercker.yml file to your github repository. (I would recommend adding the file to your local file system, stage, commit, and then push the file to your github repository).

To be honest, for our use case this wercker.yml file created is pretty useless. We will be making some significant changes to this file. For our particular use case, it would be better if we had just created it from scratch, which you are free to do.

Once you have your wercker.yml file let’s start making it useful for our purposes.

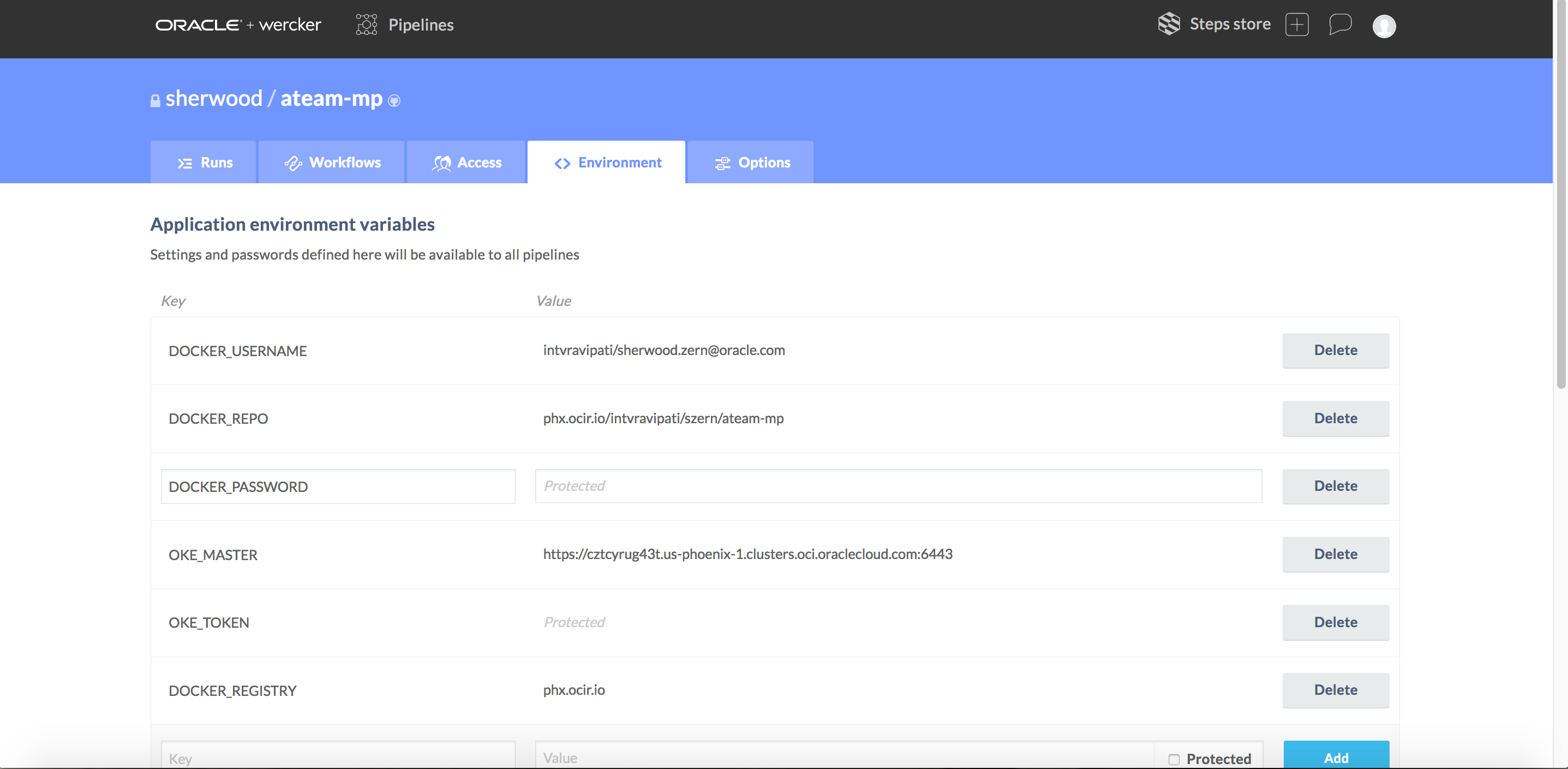
# Build the Image and push to OCIR

|  |  |
| --- | --- |
| **Current Code** | **New Code** |
| box: openjdk:8-jdk | box: docker |
| build: | build:  docker: true |
| steps: (remove all of the lines underneath) | steps:  - script:  name: Build the container  code: |  docker build -t ateam-mp:latest target  docker tag ateam-mp:latest $DOCKER\_REPO:1.0.0  - script:  name: Push the container to OCIR  code: |  docker login $DOCKER\_REGISTRY -u $DOCKER\_USERNAME -p $DOCKER\_PASSWORD  docker push $DOCKER\_REPO:1.0.0 |
|  |  |

Why are we doing this?

1. We are requesting a box that contains the docker daemon.
   1. Setting the value docker: true is requesting to start docker as soon as the image is started.
2. In the “steps:” portion we are requesting wercker to execute the following steps.
   1. Build the docker image (docker build -t …)
   2. Tag the image as the name of your repository with a version of 1.0.0.
3. In the final step we are having docker push the image to our OCIR repository.

Let’s talk about the environment variables. In order to use the environment variables shown in the YAML they need to be defined in the environment variables within wercker. Let’s do this now.

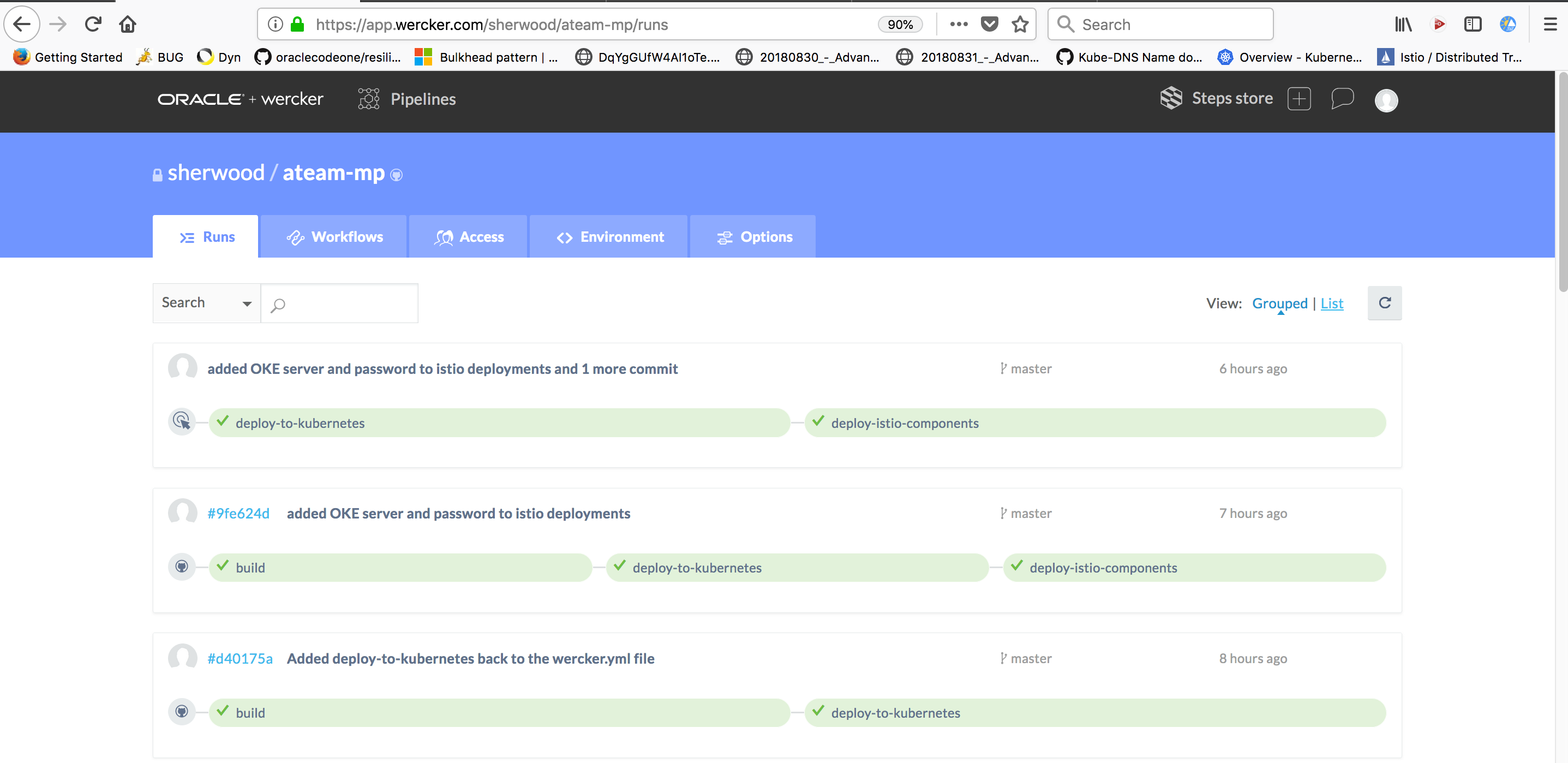


What you see above is a sample application with all of the environment variables that are referenced within the YAML file. Within the wercker UI go to the “Environment” tab. You need to specify each of the environment variables here. Below is a list of the variables that we will be using in the wercker.yml file.

|  |  |
| --- | --- |
| **Key** | **Value** |
| DOCKER\_USERNAME | <tenancy>/<user id> |
| DOCKER\_REPO | <region>.ocir.io/<tenancy>/<repository name> |
| DOCKER\_PASSWORD | The generated OCIR authentication token (select Protected)  https://docs.cloud.oracle.com/iaas/Content/Registry/Tasks/registrygettingauthtoken.htm |
| OKE\_MASTER | The server URL from the kubeconfig  cat kubeconfig |
| OKE\_TOKEN | The token from the kubeconfig (select Protected)  cat kubeconfig |
| DOCKER\_REGISTRY | <region>.ocir.io |

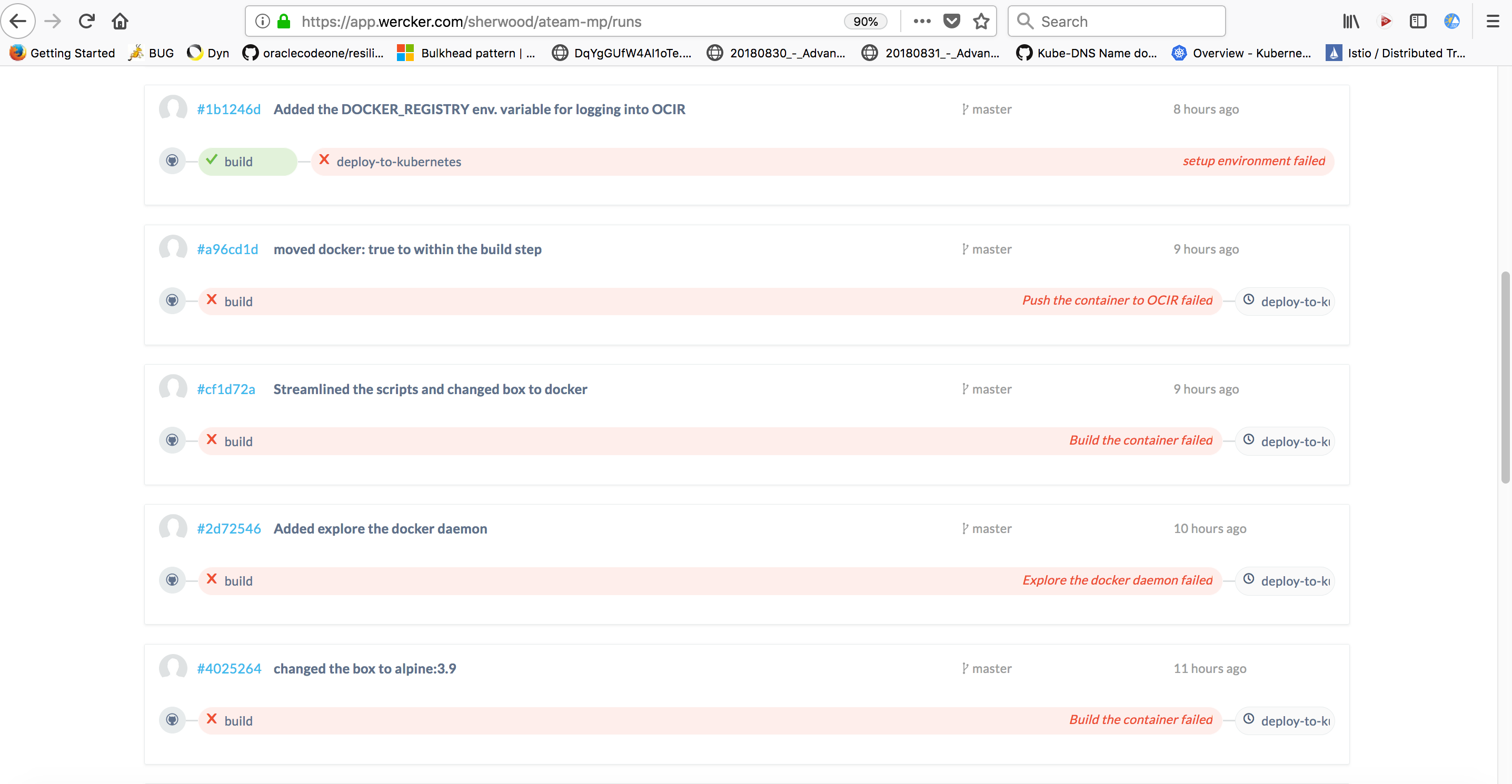
Now that you have the environment variables set and the wercker.yml file modified, update the github with your new wercker.yml file. As soon as you push the file to your github a build will automatically be kicked off.

Go to the “Runs” tab in the wercker UI.

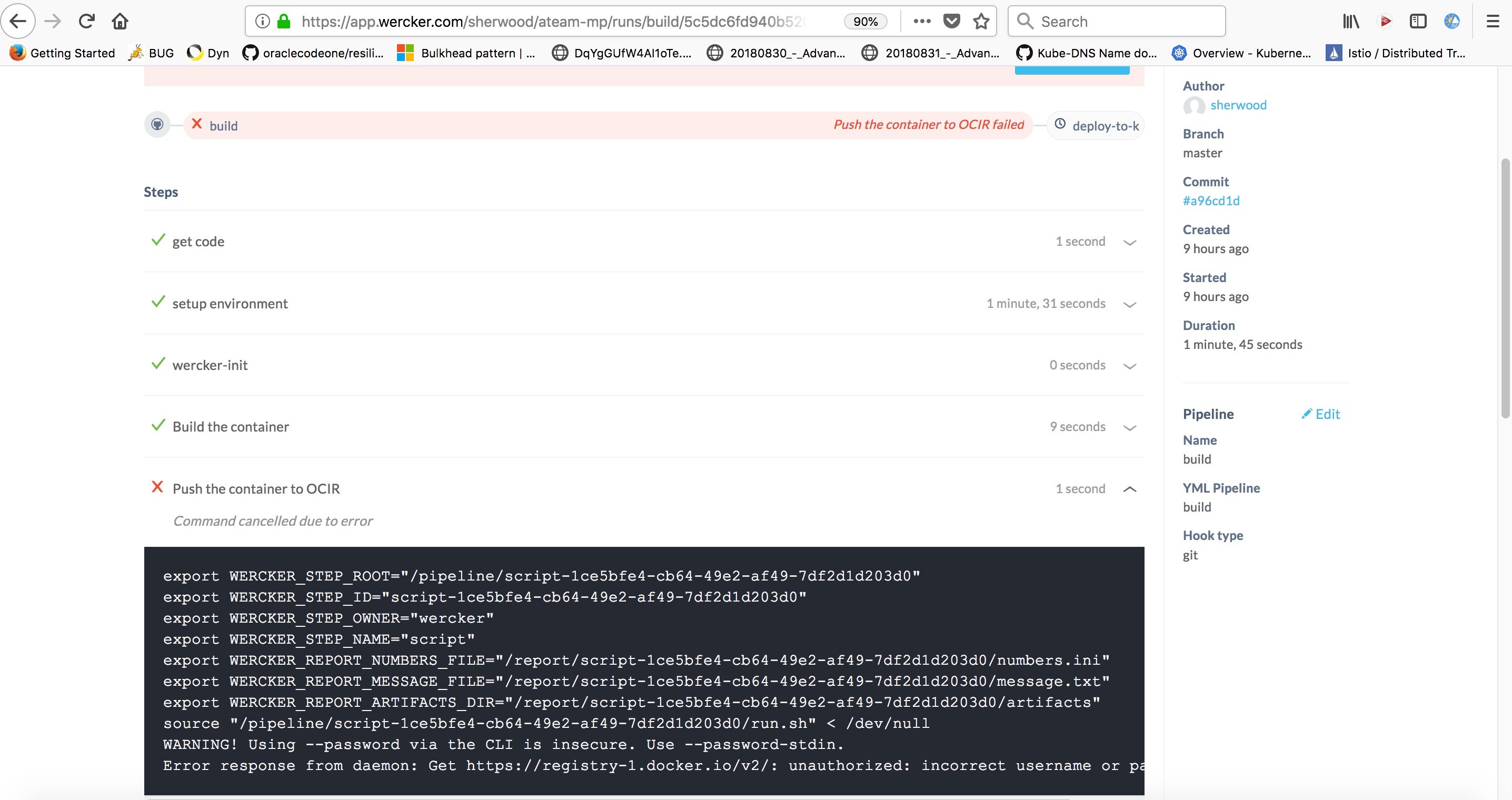


If your build was successful then it will show green, as demonstrated above. Make sure your build was successful before moving on to the next step. Make sure you find your image in OCIR. If you do not find your image in OCIR then you need to make sure your build process is correct within the wercker.yml.

If your build failed then you will see your status as red. You can see an example of such failures below. Also, notice that the failure will list in which step the failure happened.



To understand why your build or deployment failed click on the red bar. You will be navigated to the status page regarding the build / deployment summary.



You can inspect each of the steps of the process up to the failure. Expand the where the failure happened and you will be able to see the cause of the error.

# Deploy to Kubernetes

With the build successful, let’s now deploy our image to Kubernetes. For this to work you will need to make sure you have the kubernetes yaml files ready to go. You will need to have the Service and Deployment yaml files. In the github repository I have provided these yaml files to help jumpstart you on this effort. You will need to fill out the remaining parts. Below you will see the yaml file. The pieces in the yaml file where you see {. . . . } are the areas you need to fill out.

apiVersion: v1

kind: Service

metadata:

name: {name-of-your-service}

namespace: default

labels:

app: {name-of-your-service}

version: v1

spec:

selector:

app: {name-of-your-service}

ports:

- port: 8080

targetPort: 8080

name: http

---

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: {name-of-your-service}-v1

namespace: default

labels:

app: {name-of-your-service}

version: v1

spec:

replicas: 1

template:

metadata:

labels:

app: {name-of-your-service}

version: v1

spec:

containers:

- image: {<ocir-registry>/image-name}:{tag}

imagePullPolicy: Always

name: welcome

ports:

- containerPort: 8080

**Don’t forget to add these components to your github repository.**

We’re now going to modify the wercker.yml file to do the deploy to kubernetes. The name of the pipeline will be “deploy-to-kubernetes”.

Since this is a new pipeline it will not fall within the build process, but is on the same level as the build process, i.e. it must start in column 1.

deploy-to-kubernetes:

box: openjdk:8-jre-slim

steps:

- kubectl:

name: deploy to kubernetes

server: $OKE\_MASTER

token: $OKE\_TOKEN

insecure-skip-tls-verify: true

command: apply -f k8s/<name of the svc and deployment>.yaml

We are instructing wercker to invoke the kubectl command to deploy our service and deployment to the OKE cluster defined by the $OKE\_MASTER. We are also supplying the token for security purposes.

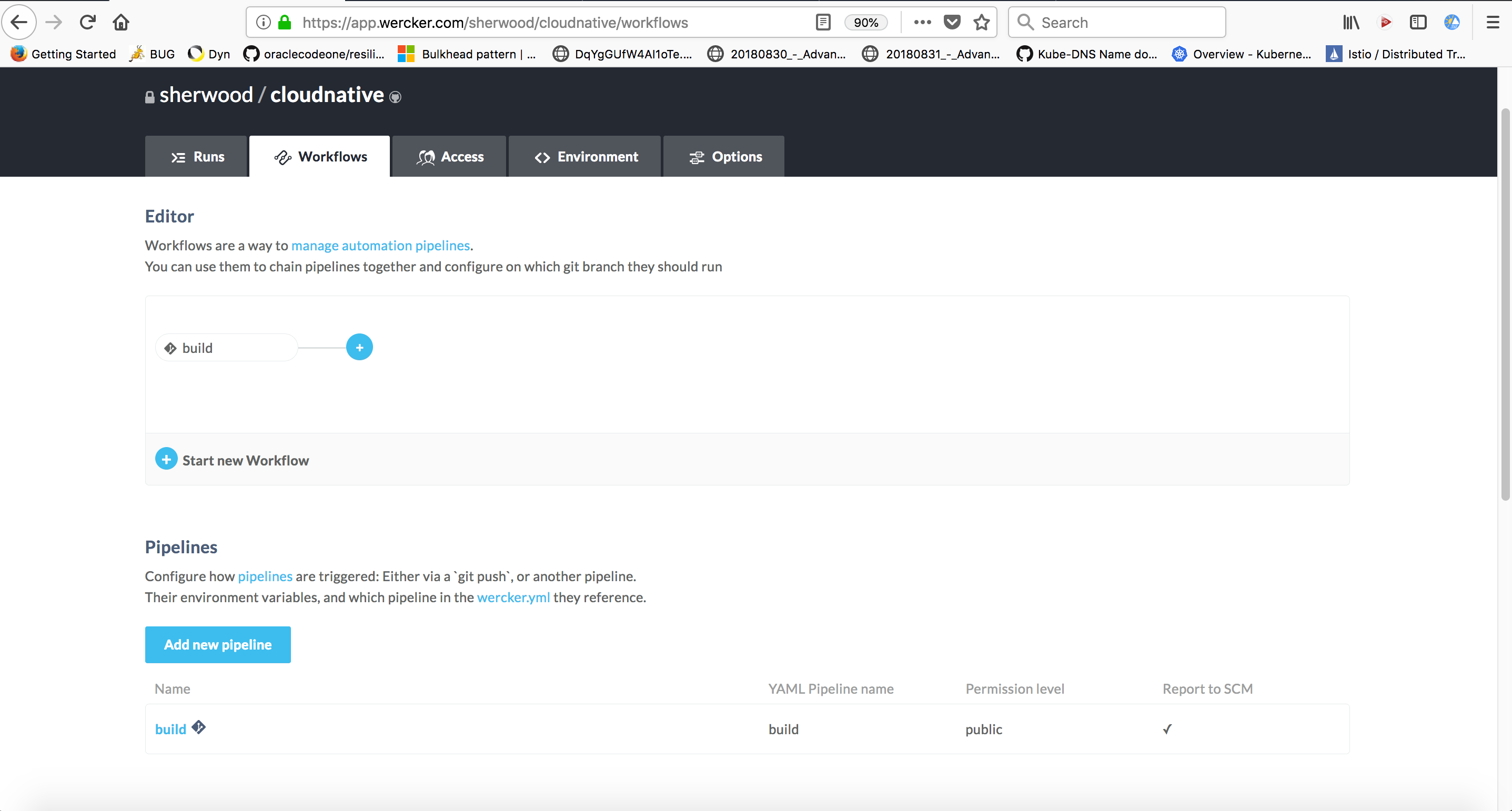
Save your wercker.yml file. Now stage, commit, and push these changes to your github application. Once the file has been uploaded, wercker will start another build process but now a successful build will do a deployment to kubernetes will happen.

Go to wercker to verify the build happened and the deployment happened.

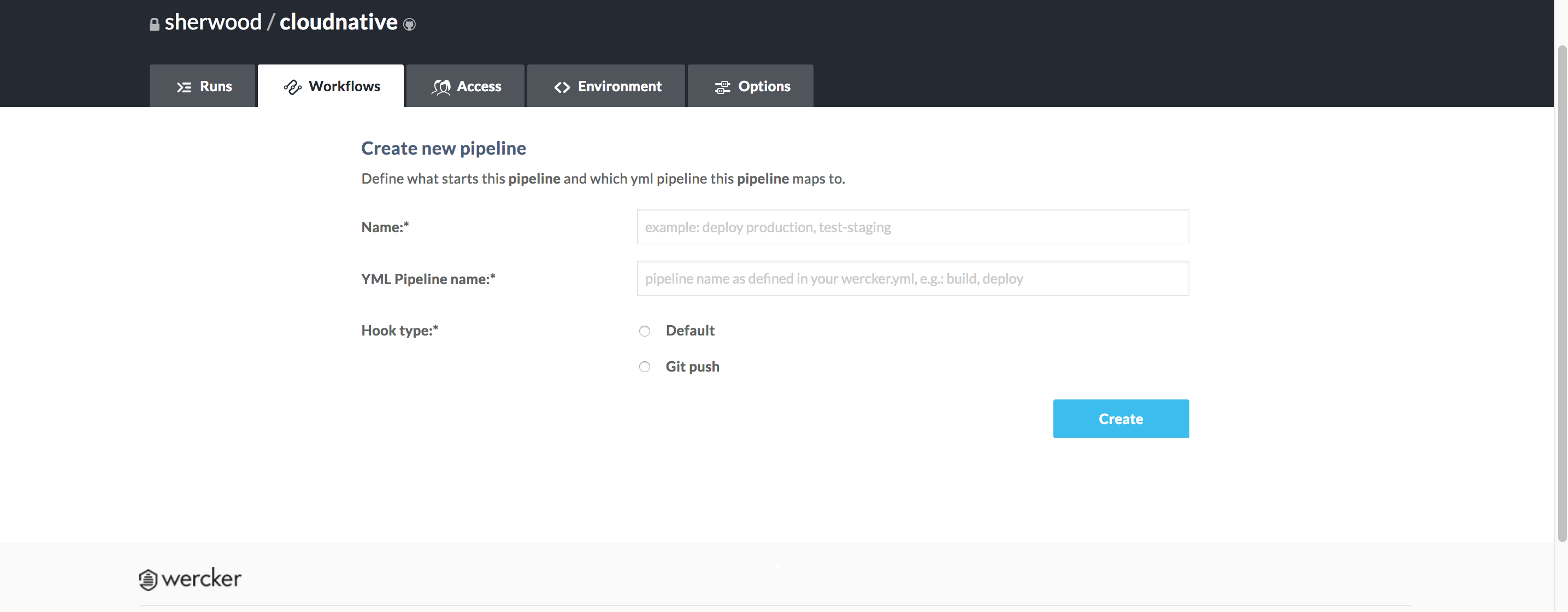
Did the deployment to Kubernetes occur? If not, why? Did the pipeline even run?

The deploy-to-kubernetes did not run, since we have not defined the pipeline in the wercker workflow. We did specify the pipeline name in the wercker.yml file, but have not made it known to the wercker workflow. Let’s do that now.

Select the “workflows” tab in the wercker UI.

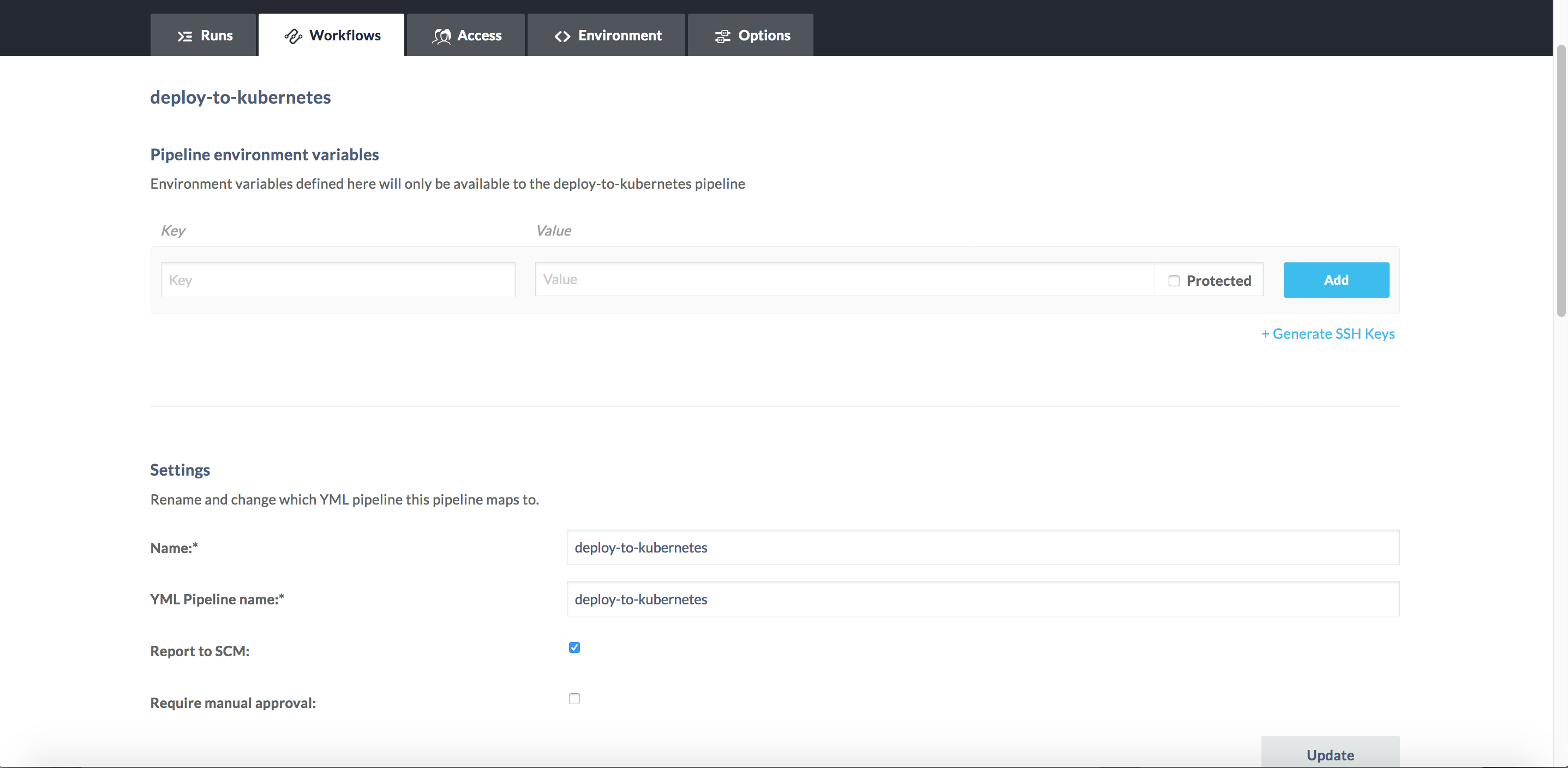


In this window, select “Add new pipeline”. You will now be presented with a new window. Populate all of the required fields.

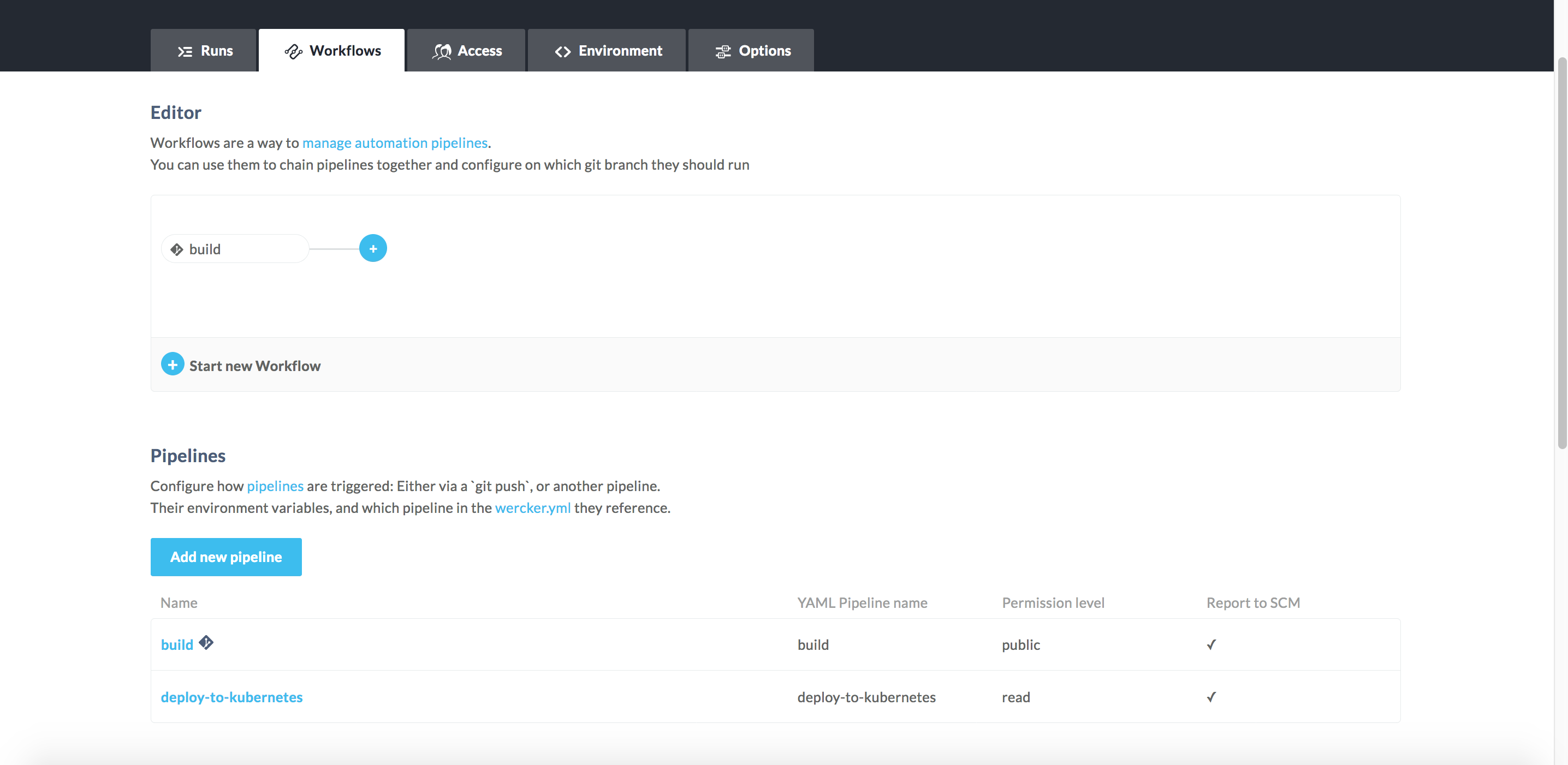


The “Name” field is required, but is a value for description type purposes. The “YML Pipeline name” must match the name you have specified in the wercker.yml file. The “Hook type”, select Default. In fact, it will probably get automatically selected.

Select create and a new pipeline will get created. Upon creation a new window displays. In this window, you have the ability to add environment variables that are only for this pipeline.

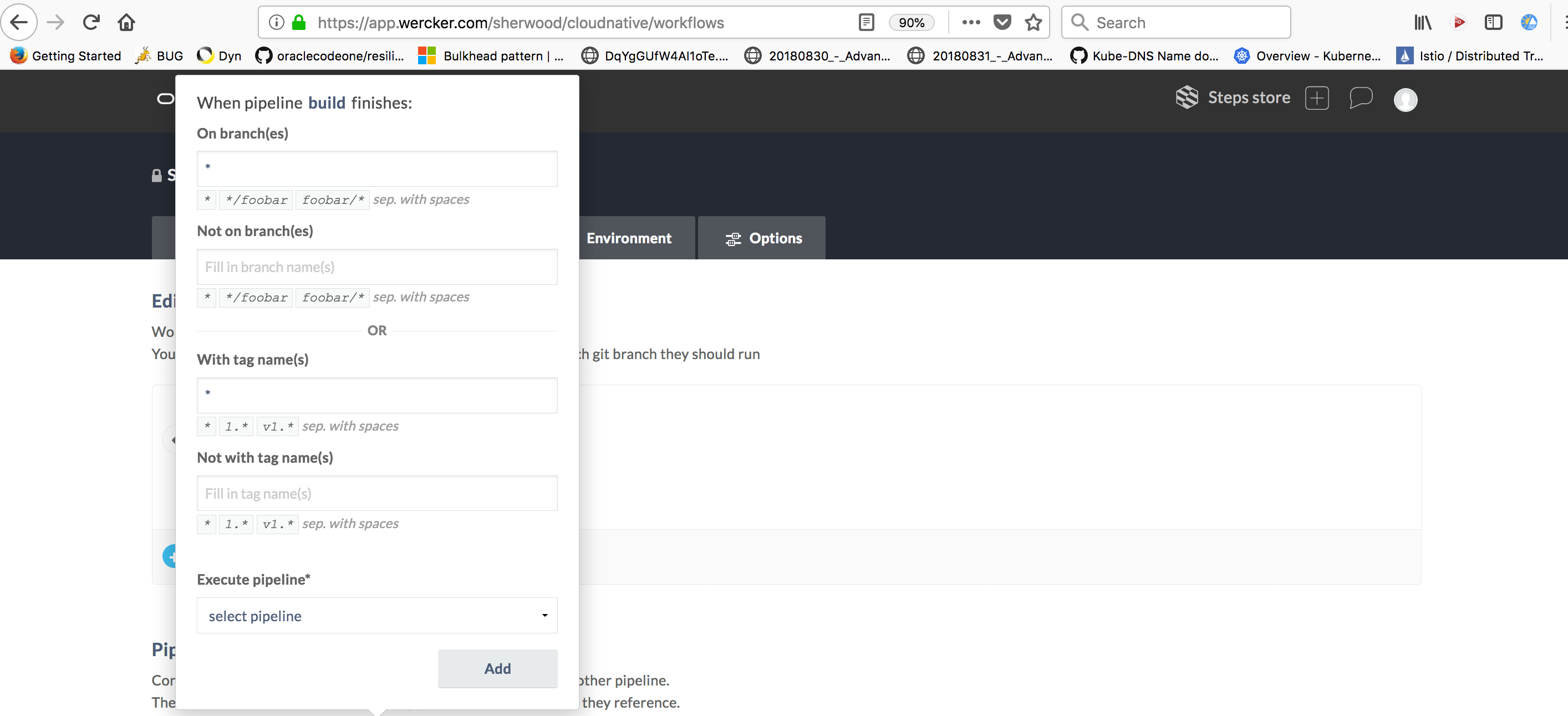


I usually enable “Report to SCM” and select “Update”. After completing any updates then select the “Workflows” tab. Now in the window, under the Pipelines heading you will see the new pipeline “deploy-to-kubernetes”.



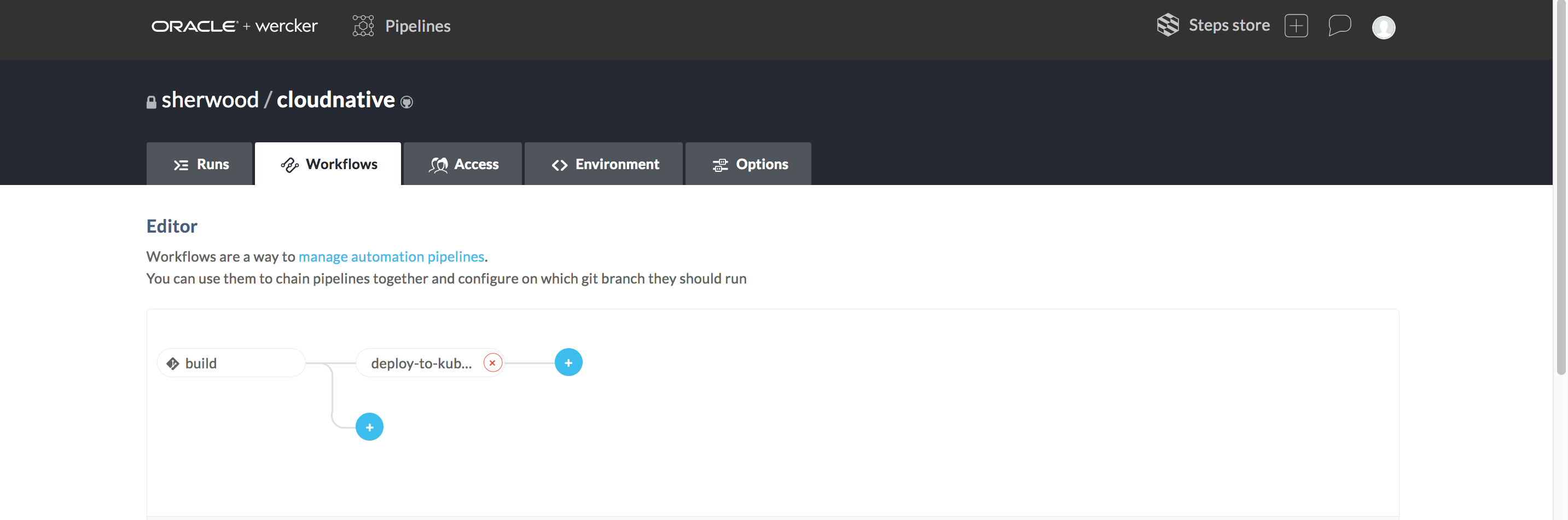
Add the “deploy-to-kubernetes” pipeline to our workflow.

Next to the “build” pipeline select the blue, round “+” icon.



We are going to make this simple. For the “Execute pipeline” drop down the “select pipeline” drop-down and select “deploy-to-kubernetes”. After choosing “deploy-to-kubernetes” hit the “Add” button.

The workflow now has an additional pipeline to execute – deploy-to-kubernetes.



Rerun the workflow. Verify that the docker image gets built, is pushed to OCIR, and the service is deployed to kubernetes.

To verify your service is deployed to kubernetes.

$ kubectl get pod

$ kubectl get deployment

Your service should now show up in the pod list and the deployment list.

The very last step is to deploy the Istio components. The reason for the Istio components is so we can have an ingress gateway and manage and monitor our services. Edit the wercker.yml file to add another pipeline – deploy-istio-components.

deploy-istio-components:

box: alpine:3.9

steps:

- kubectl:

name: deploy the virtualservice

server: $OKE\_MASTER

token: $OKE\_TOKEN

insecure-skip-tls-verify: true

command: apply -f istio/ateam-mp-virtualservice.yaml

- kubectl:

name: deploy the destination rule

server: $OKE\_MASTER

token: $OKE\_TOKEN

insecure-skip-tls-verify: true

command: apply -f istio/ateam-mp-destinationrule.yaml

We’ve covered at a high-level what is happening so there is no need to discuss each of the kubectl commands.

* Update the wercker.yml file
* Push the wercker.yml file to github
* Create the virtualservice yaml file
* Create the destinationrule yaml file
  1. Since we have covered Istio and you have a number of samples on how to create these files I will not provide the yaml files for you. If you are stuck then review some of the yaml files that we covered on Day 2. If you are unsure the instructor will assist.
* Create the pipeline in wercker
* Add the pipeline to the workflow
* Trigger the workflow
* Verify all of the pipelines in the workflow succeeded
* Make sure the pod, deployment, virtualservice, and destination rule are found in your k8s cluster.
* Test your service. curl -v http://<external-ip>/greet