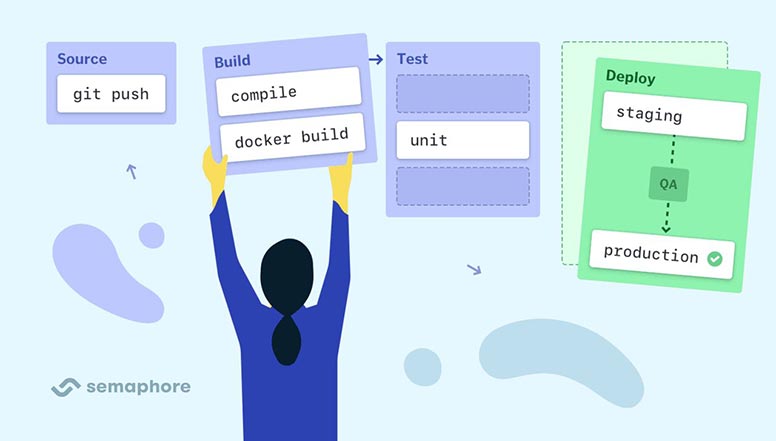
**CI/CD Pipeline: A Gentle Introduction**



**You can also download this article as a PDF and read it on the couch.**

[**Download**](https://semaphoreci.com/resources/download-cicd-pipeline)

Do you want your engineering team to deliver bug-free code at high velocity? A fast and reliable [CI/CD](https://semaphoreci.com/cicd) pipeline is crucial for doing that sustainably over time.

**What is a CI/CD pipeline?**

A CI/CD pipeline automates your software delivery process. The pipeline [builds code](https://semaphoreci.com/blog/build-stage), runs tests (CI), and safely deploys a new version of the application (CD).

Automated pipelines remove manual errors, provide standardized feedback loops to developers, and enable fast product iterations.

**What do CI and CD mean?**

CI, short for [*Continuous Integration*,](https://semaphoreci.com/continuous-integration) is a software development practice in which all developers merge code changes in a central repository multiple times a day. CD stands for *Continuous Delivery*, which on top of Continuous Integration adds the practice of automating the entire software release process.

With CI, each change in code triggers an automated build-and-test sequence for the given project, providing feedback to the developer(s) who made the change. The entire CI feedback loop should run in [less than 10 minutes](https://semaphoreci.com/blog/2017/03/02/what-is-proper-continuous-integration.html).

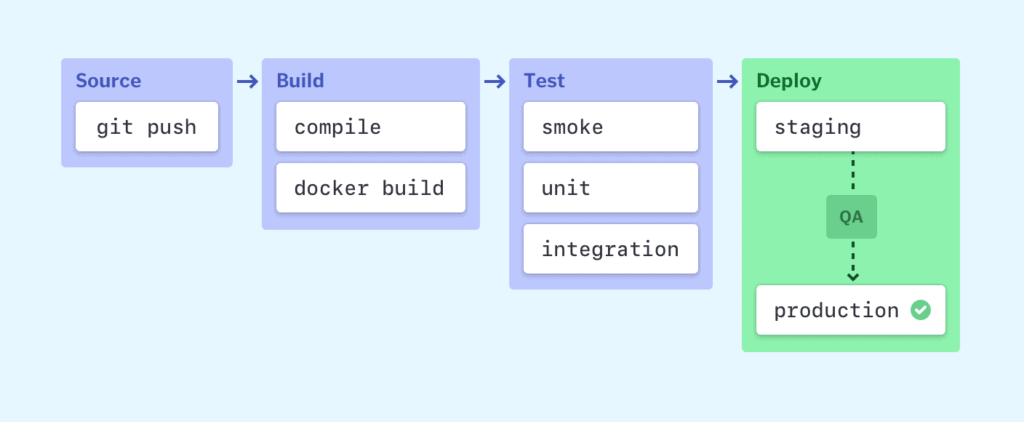
Continuous Delivery includes infrastructure provisioning and deployment, which may be manual and consist of multiple stages. What’s important is that all these processes are fully automated, with each run fully logged and visible to the entire team.

*Learn more here:*[*CI/CD: Continuous Integration & Delivery Explained*](https://semaphoreci.com/cicd)

**Elements of a CI/CD pipeline**

A CI/CD pipeline may sound like overhead, but it isn’t. It’s essentially a runnable specification of the steps that any developer needs to perform to deliver a new version of a software product. In the absence of an automated pipeline, engineers would still need to perform these steps manually, and hence far less productively.

Most software releases go through a couple of typical stages:

Stages of a CI/CD pipeline

Failure in each stage typically triggers a notification—via email, Slack, etc.—to let the responsible developers know about the cause. Otherwise, the whole team receives a notification after each successful deployment to production.

**Source stage**

In most cases, a pipeline run is triggered by a source code repository. A change in code triggers a notification to the CI/CD tool, which runs the corresponding pipeline. Other common triggers include automatically scheduled or user-initiated workflows, as well as results of other pipelines.

**Build stage**

We combine the source code and its dependencies to [build a runnable instance of our product](https://semaphoreci.com/blog/build-stage) that we can potentially ship to our end users. Programs written in languages such as Java, C/C++, or Go need to be compiled, whereas Ruby, Python and JavaScript programs work without this step.

Regardless of the language, cloud-native software is typically deployed with Docker, in which case this stage of the CI/CD pipeline builds the Docker containers.

Failure to pass the build stage is an indicator of a fundamental problem in our project’s configuration, and it’s best to address it immediately.

**Test stage**

In this phase, we run automated tests to validate our code’s correctness and the behavior of our product. The test stage acts as a safety net that prevents easily reproducible bugs from reaching the end-users.

The responsibility of writing tests falls on the developers. The best way to write automated tests is to do so as we write new code in [test- or behavior-driven development](https://semaphoreci.com/community/tutorials/behavior-driven-development).

Depending on the size and complexity of the project, this phase can last from seconds to hours. Many large-scale projects run tests in multiple stages, starting with [smoke tests](https://semaphoreci.com/community/tutorials/smoke-testing) that perform quick sanity checks to end-to-end integration tests that test the entire system from the user’s point of view. An extensive test suite is typically parallelized to reduce run time.

Failure during the test stage exposes problems in code that developers didn’t foresee when writing the code. It’s essential for this stage to produce feedback to developers quickly, while the problem space is still fresh in their minds and they can [maintain the state of flow](https://semaphoreci.com/blog/2016/11/03/how-bdd-and-continuous-delivery-help-developers-maintain-flow.html).

**Deploy stages**

Once we have a built a runnable instance of our code that has passed all predefined tests, we’re ready to deploy it. There are usually multiple deploy environments, for example, a “beta” or “staging” environment which is used internally by the product team, and a “production” environment for end-users.

Teams that have embraced the Agile model of development—guided by tests and real-time monitoring—usually deploy work-in-progress manually to a staging environment for additional manual testing and review, and automatically deploy approved changes from the master branch to production.

**Examples of CI/CD pipelines**

A pipeline can start very simple. Here’s [an example of a Go project pipeline](https://github.com/semaphoreci-demos/semaphore-demo-go) that compiles the code, checks code style and runs automated tests in two parallel jobs:

A simple CI pipeline for a Go project

Here’s a more complex example of a pipeline that [builds, tests and deploys a microservice to a Kubernetes cluster](https://semaphoreci.com/blog/cicd-microservices-digitalocean-kubernetes):

# CI/CD for Microservices on DigitalOcean Kubernetes

[Semaphore](https://semaphoreci.com/) gives you the power to easily create CI/CD pipelines that build, run and deploy Docker containers. DigitalOcean recently introduced [a managed Kubernetes service](https://blog.digitalocean.com/digitalocean-releases-k8s-as-a-service/) which simplifies running cloud-native applications. Together, they’re a great match for productive software development. In this article, we’ll show you how to connect these two services together in a fast continuous delivery pipeline.

## What we’re building

We’ll use a Ruby Sinatra microservice which exposes a few HTTP endpoints and includes a test suite. We’ll package it with Docker and deploy it to DigitalOcean Kubernetes. The [CI/CD pipeline](https://semaphoreci.com/blog/cicd-pipeline) will fully automate the following tasks:

* Install project dependencies, reusing them from cache most of the time;
* Run unit tests;
* Build and tag a Docker image;
* Push the Docker image to Docker Hub container registry;
* Provide a one-click deployment to DigitalOcean Kubernetes.

Graphical user interface

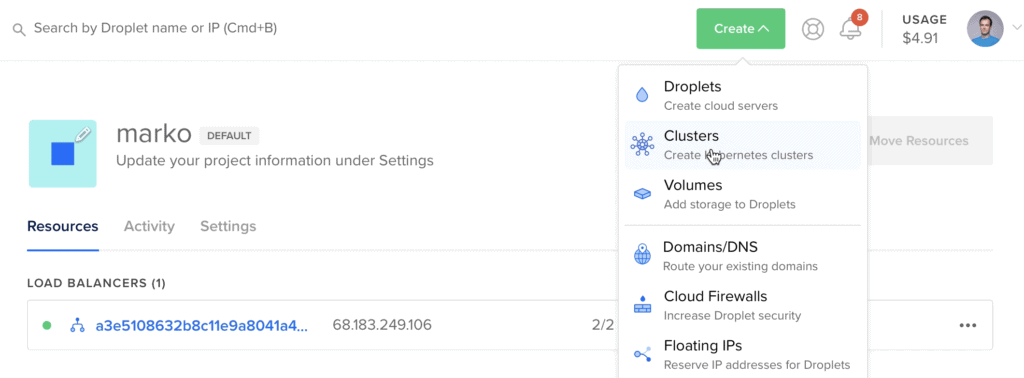
Description automatically generatedFirst CI/CD Pipeline

We’ll go step by step, but if you’d like to jump straight into the final version of source code, check out [semaphore-demo-ruby-kubernetes](https://github.com/semaphoreci-demos/semaphore-demo-ruby-kubernetes) repository on GitHub.

Let’s begin!

## Launch a Kubernetes cluster in 5 minutes on DigitalOcean

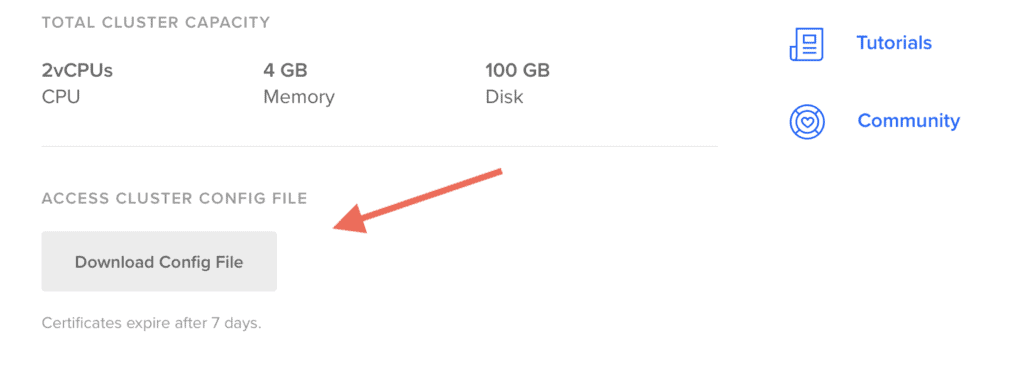
Launching a Kubernetes cluster on DigitalOcean is straightforward. From your dashboard, use the **Create** button on top to create it. The cluster will become available in 4-5 minutes.

[](https://wpblog.semaphoreci.com/wp-content/uploads/2019/03/create-digitalocean-kubernetes-cluster.png)Creating a Kubernetes cluster on DigitalOcean

The cluster page includes a number of tips and resources that you can use. If you haven’t done that yet, now is the time to [install kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/), the Kubernetes command-line tool.

## Connect to your DigitalOcean Kubernetes cluster

Scroll to the bottom of your cluster page to download the configuration file that you will use to authenticate and connect to the cluster.

[](https://wpblog.semaphoreci.com/wp-content/uploads/2019/03/download-k8s-config.png)Download your DigitalOcean Kubernetes configuration file

On your local machine, create a directory to contain the Kubernetes configuration file:

$ mkdir ~/.kube

Move the downloaded file to ~/.kube and instruct kubectl to use it. You can run the following command in your terminal session, or add it to your shell profile like .bashrc or .zshrc:

$ export KUBECONFIG=$HOME/.kube/dok8s.yaml

Make sure to change dok8s.yaml to match the name of your file.

Verify that you can communicate with your DigitalOcean Kubernetes cluster by running kubectl get nodes. When the command is successful, it returns information similar to the following:

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

nostalgic-heisenberg-8vi3 Ready <none> 4d v1.13.2

nostalgic-heisenberg-8vi8 Ready <none> 4d v1.13.2

The number of nodes will match the number you selected during the cluster creation process. Note that if you run get nodes while your cluster is still being provisioned, the number of nodes will be zero.

## Connect Semaphore to your DigitalOcean Kubernetes cluster

At this point, you have a Kubernetes cluster that you can control from your local machine. Let’s configure a basic CI/CD project in which Semaphore can also successfully execute kubectl get nodes.

If you’re new to Semaphore, start by [creating a free account](https://semaphoreci.com/). The free account provides you with $10 of credit every month, which is enough for up to 1,300 minutes of service.

### Create a project on Semaphore

Once you’re in Semaphore, click on the **+ (plus sign)** next to **Projects** and select your repository from the list:

Graphical user interface, application, email

Description automatically generatedAdd the repository to Semaphore

When prompted, choose:

* If you want to start with a minimal configuration and build it up while you follow this tutorial, select “**I want to configure this project from scratch**“.
* If you want to jump forward to the end and view the final configuration, select the “**I will use the existing configuration**” option.

For both cases, Semaphore creates a deploy key and webhook on GitHub so that it can access your code as it changes. It also creates a pipeline definition file .semaphore/semaphore.yml.

### Authenticating with Kubernetes using a Semaphore secret

Let’s edit the pipeline and instruct Semaphore on how to talk to Kubernetes.

Semaphore already [provides kubectl preinstalled](https://docs.semaphoreci.com/ci-cd-environment/ubuntu-18.04-image/). So what’s left is to securely upload the Kubernetes configuration file inside the Semaphore environment. We generally solve this by creating a [secret](https://docs.semaphoreci.com/guided-tour/concepts/). A secret can be a collection of environment variables and files. Once created, it’s available to all projects within an organization.

In our case, we need a secret based on a single file:

* Go to your Semaphore account.
* On the left navigatio menu, click on **Secrets** under **Configuration**.
* Press the **Create New Secret**button.

Graphical user interface, text, application

Description automatically generatedCreate a new secret for the Kubernetes config.

Create the secret for your Kubernetes configuration:

* Set the name of the secret to do-k8s.
* Under files, type the path /home/semaphore/.kube/dok8s.yaml. /home/semaphore/ is the default directory from which all CI/CD jobs run.
* Upload the DigitalOcean Kubernetes Config file.
* Click on **Save Changes**.

Graphical user interface, application

Description automatically generatedUpload Kubernetes Config file to Semaphore

To modify the pipeline configuration, click on the **Edit Workflow** button on the top right corner:

Graphical user interface, text, application, email

Description automatically generatedClick on**Edit Workflow** to modify the pipeline.

Let’s try connecting to Kubernetes. Modify the initial block so it looks like this:

Graphical user interface, application

Description automatically generatedConnecting to Kubernetes

If this is the first time you see a Semaphore pipeline, [a quick tour of concepts](https://docs.semaphoreci.com/guided-tour/concepts/) will help you understand it. Here’s the gist of how they apply in this example:

* Jobs have a name and a list of commands to execute.
* Jobs are grouped in blocks.
* We initialize environment variables for the jobs. In this case, we are setting KUBECONFIG = /home/semaphore/.kube/dok8s.yml
* We mount the Kubernetes config file with the do-k8s secret we’ve just created.
* Our pipeline has one block and one job, in which we download our code from GitHub and run kubectl get nodes

The **Run Workflow** button on the top right corner to push the changes to GitHub:

Graphical user interface, text, application, email

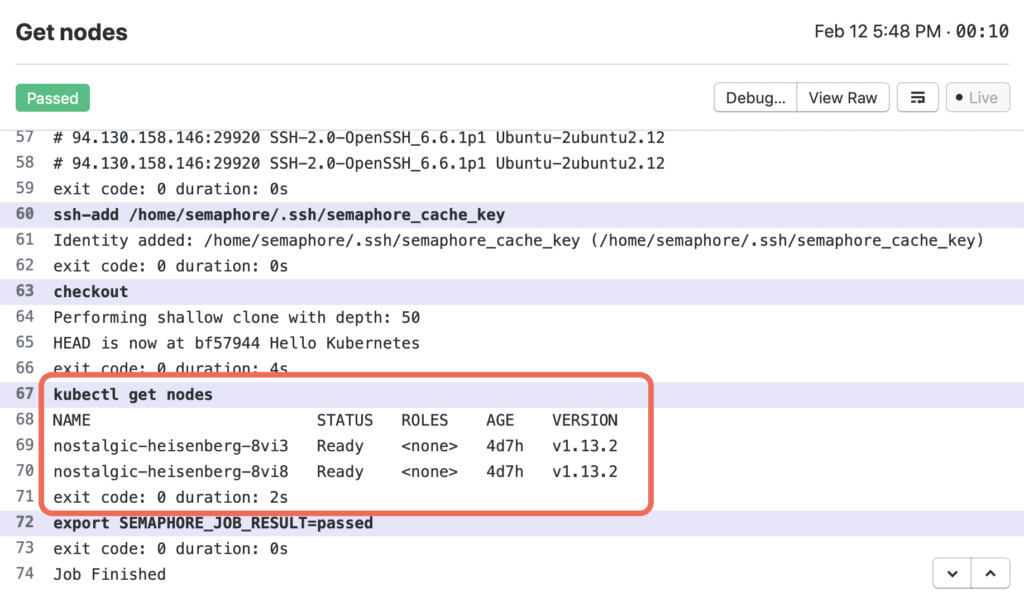
Description automatically generatedStart the pipeline

This will start the pipeline immediately:

Graphical user interface, text, application, email

Description automatically generatedSemaphore talking to the Kubernetes cluster

Clicking on the job will bring up its logs:



OK, we’re in business! Let’s proceed by setting up an actual project.

## Set up continuous integration for a Sinatra microservice

Our Sinatra app is a microservice with minimal configuration and an RSpec test suite:

.

├── Gemfile

├── Gemfile.lock

├── README.md

├── app.rb

├── config.ru

└── spec

├── app\_spec.rb

└── spec\_helper.rb

Let’s go back to setting up the Continuous Integration pipeline:

* **Edit the workflow**
* Delete the block we created above. We’ll set up a pipeline that looks like this:

Graphical user interface, text, application

Description automatically generatedThe Continuous Integration Pipeline

On Semaphore, blocks run sequentially, while jobs within a block run in parallel. The CI pipeline contains two blocks, one for installing dependencies and another for running tests. This is an example if it makes sense you could, of course, merge the blocks in one.

Click on the first block to edit the job. We’ll store our Ruby gems [in Semaphore cache](https://docs.semaphoreci.com/guided-tour/caching-dependencies/) to avoid running bundle install from scratch on every commit:

* Block name: Install dependencies
* Job name: bundle install
* Job commands:

checkout

cache restore

bundle install --deployment --path vendor/bundle

cache store

Graphical user interface, text, application, email

Description automatically generatedCreate the “Install dependencies” block

It’s necessary to run bundle install in the second block, even though cache restore will at that point always restore the gem bundle. It’s a limitation on Bundler’s side, but the good part is that the command will exit very quickly.

Graphical user interface, text, application, email

Description automatically generatedCreate the “Tests” block

When you click on **Run the Workflow** > **Start**, you’ll see a real CI pipeline shaping up on Semaphore:

Graphical user interface, text, application

Description automatically generatedContinuous integration pipeline for our microservice

### Dockerize Sinatra app

The next step is to package our Sinatra app in a Docker container. The following Dockerfile will do:

# Dockerfile

FROM ruby:2.5

RUN apt-get update -qq && apt-get install -y build-essential

ENV APP\_HOME /app

RUN mkdir $APP\_HOME

WORKDIR $APP\_HOME

ADD Gemfile\* $APP\_HOME/

RUN bundle install --without development test

ADD . $APP\_HOME

EXPOSE 4567

CMD ["bundle", "exec", "rackup", "--host", "0.0.0.0", "-p", "4567"]

To verify, you can build and run the container locally:

$ docker build -t demo .

$ docker run-p 80:4567 demo

$ curl localhost

> hello world

Add the Dockerfile to GitHub:

$ git add Dockerfile

$ git commit -m "add Dockerfile"

$ git push origin master

## Push Docker image to Docker Hub container registry

Deploying to Kubernetes requires us to push a Docker image to a container registry. In this example, we’ll use [Docker Hub](https://hub.docker.com/). The procedure is very similar to other registry providers.

Pushing to a container registry, public or private, requires authentication. For example, when you’re using Docker Desktop on Mac, you’re automatically authenticated and communication with Docker Hub just works. In the CI/CD environment, we need to make credentials available and authenticate before doing docker push.

### Authenticating with Docker Hub using a Semaphore secret

Following the [Docker Hub instructions available in Semaphore documentation](https://docs.semaphoreci.com/examples/publishing-docker-images-on-dockerhub/), we need to create a secret.

On the left menu, go to **Configuration**>**Secrets** and click on **Create New Secret**:

* Name of the Secret: “dockerhub”
* Variables:
  + DOCKER\_USERNAME: type your DockerHub username
  + DOCKER\_PASSWORD: type your DockerHub password.
* Click on **Save Changes**.

Graphical user interface, application

Description automatically generatedCreate dockerhub secret

We now have what it takes to push to Docker Hub from a Semaphore job.

### Configure a promotion to run docker build

With Docker build and push operations we are entering the delivery phase of our project. We’ll extend our Semaphore pipeline with [a promotion](https://docs.semaphoreci.com/guided-tour/deploying-with-promotions/) and use it to trigger the next stage.

To create a new promotion:

* **Edit the Workflow**
* Scroll right to the last block in the pipeline and press the **Add First Promotion** button.

Graphical user interface, application

Description automatically generatedAdd a new promotion

Configure the new promotion:

* Check the **Enable automatic promotion** option
* Call the promotion: “Dockerize”

Graphical user interface, text, application, email

Description automatically generatedConfigure the promotion

Set up the new pipeline to build a Docker image:

* Click on the new pipeline. Set its name to “Docker build”. Here we can change the machine type that runs this pipeline, there is no need to change the default **e1-standard-2**.
* Edit the first block on the new pipeline. Let’s name it “Build”.
* Open the **Secrets** section and check the **dockerhub** entry.
* Type the following commands in the Job command box:

echo "${DOCKER\_PASSWORD}" | docker login -u "${DOCKER\_USERNAME}" --password-stdin

checkout

docker pull "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:latest || true

docker build --cache-from "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:latest -t "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:$SEMAPHORE\_WORKFLOW\_ID .

docker images

docker push "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:$SEMAPHORE\_WORKFLOW\_ID

Graphical user interface, text, application, email

Description automatically generatedSetting up the build job

In the first command in the job we authenticate with Docker Hub using the environment variables defined in the dockerhub secret.

We’re using [Docker layer caching](https://docs.semaphoreci.com/ci-cd-environment/docker-layer-caching/) to speed up the container build process. First, the docker pull command retrieves a previously built image from the registry. If this is the first time we run this operation, this step will be skipped and not fail.

By using the --cache-from flag with docker build, we’re reusing the layers from the pulled image to build the new one faster.

For the docker build and docker push commands we are using the SEMAPHORE\_WORKFLOW\_ID environment variable to produce a unique artifact after every build. It’s one of the environment variables available in every Semaphore job; [see documentation for a complete list](https://docs.semaphoreci.com/ci-cd-environment/environment-variables/).

Note that we’re not creating a new version of the latest tag. We’re going to do that only after a successful deploy.

Once you finished setting up the job, click on **Run workflow** > **Start**. You should have a pipeline in a state similar to this:

Graphical user interface, text

Description automatically generatedDocker Build pipeline

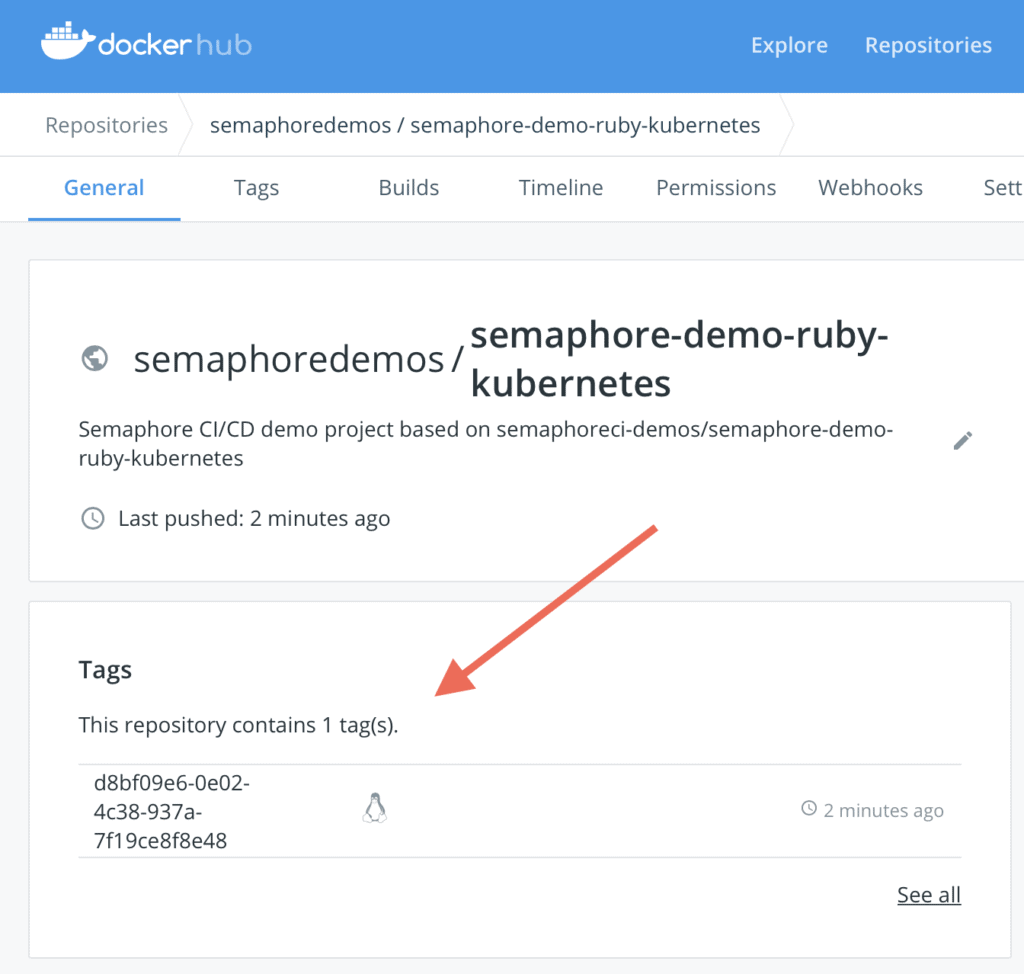
Depending on what branch you started the pipeline, the build pipeline may not start automatically. Automatic promotion is only triggered, by default, on the master branch. You may need to click **Promote** to start it.

The job log shows that the container image has been created and pushed:

[Application

Description automatically generated with low confidence](https://wpblog.semaphoreci.com/wp-content/uploads/2019/03/docker-build-job.png)Docker build job log

And your Docker registry should contain the latest images:

Pushed container image on Docker Hub

## Deploy to DigitalOcean Kubernetes

Back on the DigitalOcean’s Kubernetes cluster page, the “Getting Started” section includes examples to “Deploy a workload”. We can use the example provided for Nginx and modify it for our app.

In the example configuration, you’ll notice a reference to a source container image:

# ...

spec:

containers:

- name: nginx

image: library/nginx

If your Docker image is private, you’ll need to enable the Kubernetes cluster to authenticate with the Docker registry. The way to do that is, once again, by creating a secret, only this time on the Kubernetes cluster’s end. For demonstration purposes, I will show you how to do this, even though the image that we’re using in this tutorial is public.

### Create a Kubernetes secret to pull a private Docker image from the registry

Run the following command on your local machine to create a docker-registry-type secret on your Kubernetes cluster:

$ kubectl create secret docker-registry dockerhub-user \

--docker-server=https://index.docker.io/v1/ \

--docker-username=YOUR\_DOCKER\_HUB\_USERNAME \

--docker-password=YOUR\_DOCKER\_HUB\_PASSWORD \

--docker-email=YOUR\_EMAIL

You can verify the secret by running:

$ kubectl get secret dockerhub-user --output=yaml

Kubernetes secrets are base64-encoded and the output will look similar to:

apiVersion: v1

data:

.dockerconfigjson: eyJhdXRocyI6eyJodHR...

kind: Secret

metadata:

creationTimestamp: 2019-02-08T10:18:52Z

name: dockerhub-user

namespace: default

resourceVersion: "7431"

selfLink: /api/v1/namespaces/default/secrets/dockerhub-user

uid: eec7c39e-2b8a-11e9-a804-1a46bc991881

type: kubernetes.io/dockerconfigjson

### Write a deployment manifest

Create a new file in your repository, for example, called deployment.yml:

# deployment.yml

apiVersion: apps/v1

kind: Deployment

metadata:

name: semaphore-demo-ruby-kubernetes

spec:

replicas: 1

selector:

matchLabels:

app: semaphore-demo-ruby-kubernetes

template:

metadata:

labels:

app: semaphore-demo-ruby-kubernetes

spec:

containers:

- name: semaphore-demo-ruby-kubernetes

image: semaphoredemos/semaphore-demo-ruby-kubernetes:$SEMAPHORE\_WORKFLOW\_ID

imagePullSecrets: # if using a private image

- name: dockerhub-user

Comparing to the Nginx example provided by DigitalOcean, we’ve pretty much only substituted the application and image name. Since Semaphore is tagging images using SEMAPHORE\_WORKFLOW\_ID environment variable, we’re using it here in line 18 as well.

The deployment configuration file as it appears now is **not** valid YML. The plan is to use a Linux utility called **‌**[envsubst](https://www.gnu.org/software/gettext/manual/html_node/envsubst-Invocation.html) (also available on Mac via Homebrew) to replace $SEMAPHORE\_WORKFLOW\_ID with its value within a Semaphore CI/CD job.

Our deployment manifest, however, is not yet complete without **a Kubernetes load balancer** which will expose the deployed service on a public IP address. Append the following content to the same file:

# deployment.yml

# ...

---

apiVersion: v1

kind: Service

metadata:

name: semaphore-demo-lb

spec:

selector:

app: semaphore-demo-ruby-kubernetes

type: LoadBalancer

ports:

- port: 80

targetPort: 4567

You can verify that the Kubernetes configuration works as intended from your local machine by replacing $SEMAPHORE\_WORKFLOW\_ID with latest and running:

$ kubectl apply -f deployment.yml

$ kubectl get services

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

semaphore-demo-ruby-kubernetes LoadBalancer 10.245.117.152 68.183.249.106 80:30569/TCP 5d

...

Add the deployment manifest to GitHub:

$ git add deployment.yml

$ git commit -m "add Kubernetes manifest"

$ git push origin master

### Define a Semaphore deployment pipeline

We’re entering the last stage of CI/CD configuration. At this point, we have a CI pipeline and a Docker build pipeline. We’re going to define a third pipeline to trigger manually from Docker build which will deploy to Kubernetes.

* **Edit the Workflow**
* Scroll right to the last block in the workflow and click on the **Add First Promotion** button.
* Call the promotion “Deploy to Kubernetes”
* Click on the new pipeline. Set its name to “Deploy to Kubernetes”.
* Click on the first block on the new pipeline. Let’s call it “Deploy”.
* Under the **Secrets** section: check the **dockerhub** and **do-k8s** entries.
* Open the **Environment Variables** section. Use the **Add env\_vars** link to create a new variable:
  + KUBECONFIG = /home/semaphore/.kube/dok8s.yaml
* Set the job name to “Deploy” and type the following commands:

checkout

kubectl get nodes

kubectl get pods

envsubst < deployment.yml | tee deployment.yml

kubectl apply -f deployment.yml

Graphical user interface, application

Description automatically generatedSetting up the Deploy to Kubernetes block.

Add a second block to push the latest Docker image to the Docker Hub:

* Click on **Add Block**. Set its name to “Tag latest release”.
* Set the job name to “docker tag latest”.
* Under **Secrets**, check the **dockerhub** and **do-k8s** entries.
* Type the following commands in the job box:

echo "${DOCKER\_PASSWORD}" | docker login -u "${DOCKER\_USERNAME}" --password-stdin

docker pull "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:$SEMAPHORE\_WORKFLOW\_ID

docker tag "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:$SEMAPHORE\_WORKFLOW\_ID "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:latest

docker push "${DOCKER\_USERNAME}"/semaphore-demo-ruby-kubernetes:latest

Graphical user interface, text, application, chat or text message

Description automatically generatedSetting up the Tag latest release block

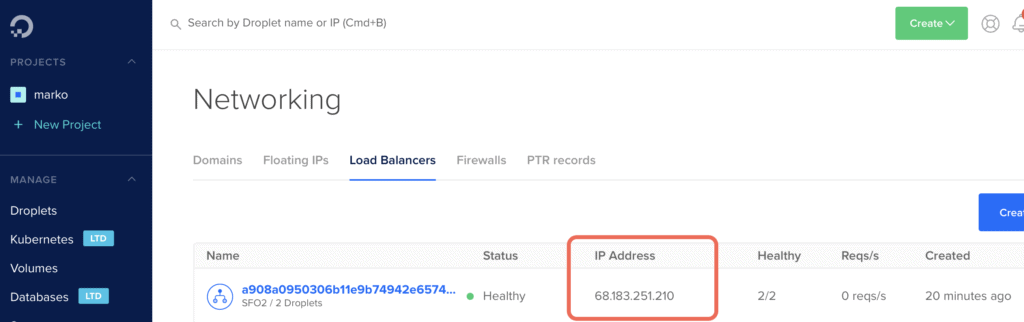
The deployment pipeline has two blocks: to apply a new Kubernetes configuration and to create a new version of our latest container image, which we’re treating like the master branch in Git (your practice may vary).

Once you click on **Run workflow** > **Start,** Semaphore runs the pipeline. Once the Docker build pipeline is completed successfully, click on the “Promote” button to trigger the deployment:

Graphical user interface, application

Description automatically generatedLaunched deployment to Kubernetes

You can now run kubectl get services or open the Load Balancers list on DigitalOcean > Networking tab section to find the public IP address of your microservice:

Public IP address from the load balancer

And test it out:

$ curl 68.183.251.210

hello world :))

Congratulations! You now have a fully automated continuous delivery pipeline to Kubernetes.

## Deploy a demo app yourself

Feel free to fork the [semaphore-demo-ruby-kubernetes](https://github.com/semaphoreci-demos/semaphore-demo-ruby-kubernetes) repository and create a Semaphore project to deploy it on your Kubernetes instance.

[semaphoreci-demos](https://github.com/semaphoreci-demos)/[semaphore-demo-ruby-kubernetes](https://github.com/semaphoreci-demos/semaphore-demo-ruby-kubernetes)

Here are some ideas for potential changes you can make:

* Introduce a staging environment
* Build a Docker image first, and run tests inside it (requires a development version of Dockerfile since it’s best to avoid installing development and test dependencies when producing an image for production).
* Extend the project with more microservices.