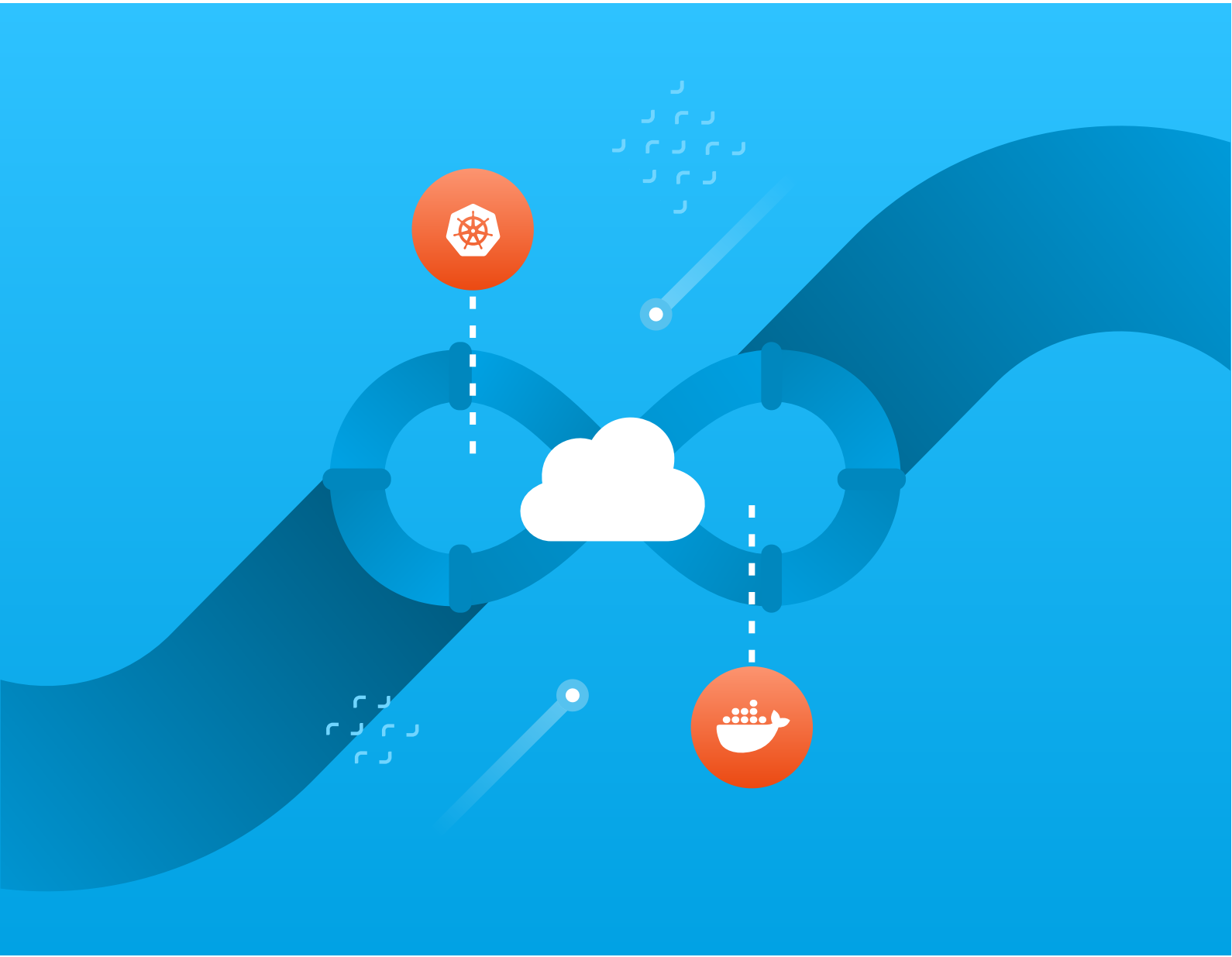
semaphore

CI/CD with Docker and Kubernetes

How to Deliver Cloud Native Applications at High Velocity



CI/CD with Docker and Kubernetes

Second Edition — How to Deliver Cloud Native Applications at High Velocity

Semaphore

Contents

This book is open source: https://github.com/semaphoreci/book-cicd-docker-kubernetes

Published on the Semaphore website: https://semaphoreci.com

Apr 2022: Second edition v2.0 (revision 53c52fc)

Share this book:

> *I've just started reading "CI/CD with Docker and Kubernetes", a free ebook by @semaphoreci: https://bit.ly/3bJELLQ (Tweet this!)*

Preface

To maximize the rate of learning, we must minimize the time to try things.

In software development, the cloud has been a critical factor in increasing the speed of building innovative products.

Today there's a massive change going on in the way we're using the cloud. To borrow the metaphor from Adrian Cockroft[1], who led cloud architecture at Netflix, we need to think of cloud resources not as long-lived and stable pets, but as transitory and disposable cattle.

Doing so successfully, however, requires our applications to adapt. They need to be disposable and horizontally scalable. They should have a minimal divergence between development and production so that we can continuously deploy them multiple times per day.

A new generation of tools has democratized the way of building such *cloud native* software. Docker containers are now the standard way of packaging software in a way that can be deployed, scaled, and dynamically distributed on any cloud. And Kubernetes is the leading platform to run containers in production. Over time new platforms with higher-order interfaces will emerge, but it's almost certain that they will be based on Kubernetes.

The great opportunity comes potentially at a high cost. Countless organizations have spent many engineering months learning how to deliver their apps with this new stack, making sense of disparate information from the web. Delaying new features by months is not exactly the outcome any business wants when engineers announce that they're moving to new tools that are supposed to make them more productive.

This is where this book comes into play, dear reader. Our goal is to help you transition to delivering cloud native apps quickly. The fundamentals don't change: we still need a rock-solid delivery pipeline, which automatically configures, builds, tests, and deploys code. This book shows you how to do that in a cloud native way — so you can focus on building great products and solutions.

[1] Currently VP Amazon Sustainability Architecture at Amazon *https://twitter.com/adrianco*

Who Is This Book For, and What Does It Cover?

The main goal of this book is to provide a practical roadmap for software development teams who want to:

- Use Docker containers to package their code,
- Run it on Kubernetes, and
- Continuously deliver all changes.

We don't spend much time explaining why you should, or should not use container technologies to ship your applications. We also don't provide a general reference to using Docker and Kubernetes. When you encounter a concept of Docker or Kubernetes that you're not familiar with, we recommend that you consult the official documentation.

We assume that you're fairly new to the container technology stack and that your goal is to establish a standardized and fully automated build, test, and release process.

We believe that both technology leaders and individual contributors will benefit from reading this book.

If you are a CTO or otherwise ultimately responsible for delivering working software to customers, this book will provide you with a clear vision of what a reliable CI/CD pipeline to Kubernetes looks like, and what it takes to build one.

If you are a developer or systems administrator, besides understanding the big picture, you will also find working code and configuration that you can reuse in your projects.

Chapter 1, "Using Docker for Development and CI/CD", outlines the key benefits of using Docker and provides a detailed roadmap to adopting it.

Chapter 2, "Deploying to Kubernetes", explains what you need to know about Kubernetes deployments to deliver your containers to production.

Chapter 3, "Best Practices for Cloud Native Applications", describes how both our culture and tools related to software delivery need to change to fully benefit from the agility that containers and cloud can offer.

Chapter 4, "A Complete CI/CD Pipeline", is a step-by-step guide to implementing a CI/CD pipeline with Semaphore that builds, tests, and deploys a Dockerized microservice to Kubernetes.

Changes in the Second Edition

A few changes were introduced in this second edition:

- Moved to Kubernetes version v1.20. All commands and actions were tested with this version.
- Added comments about accessing services in local development Kubernetes clusters.
- Added mention of new CI/CD features in Semaphore: parameterized pipelines, test results, code change detection.
- DigitalOcean deployment now uses their Private Container Registry service instead of Docker Hub.
- Updated setup steps for DigitalOcean, Google Cloud, and AWS.
- Updated UI screenshots using higher resolution.
- Modified deployment tutorial to use parametrized promotions.
- Other minor fixes.

How to Contact Us

We would very much love to hear your feedback after reading this book. What did you like and learn? What could be improved? Is there something we could explain further?

A benefit of publishing an ebook is that we can continuously improve it. And that's exactly what we intend to do based on your feedback.

You can send us feedback by sending an email to learn@semaphoreci.com.

Find us on Twitter: https://twitter.com/semaphoreci

Find us on Facebook: https://facebook.com/SemaphoreCI

Find us on LinkedIn: https://www.linkedin.com/company/rendered-text

About the Authors

Marko Anastasov is a software engineer, author and entrepreneur. Marko co-founded Rendered Text, a software company behind the Semaphore CI/CD service. He worked on building and scaling Semaphore from an idea to a cloud-based platform used by some of the world's best engineering teams. He writes about architectures, practices and tools that support continuous delivery on semaphoreci.com/blog. Follow Marko on Twitter at @markoa.

Jérôme Petazzoni was part of the team that built, scaled, and operated the dotCloud PAAS, before that company became Docker. He worked seven years at the container startup, where he wore countless hats and ran containers in production before it was cool. He loves to share what he knows, which led him to give hundreds of talks and demos on containers, Docker, and Kubernetes. He has trained thousands of people to deploy their apps in confidence on these platforms, and continues to do so as an independent consultant. He values diversity, and strives to be a good ally, or at least a decent social justice sidekick. He also collects musical instruments and can arguably play the theme of Zelda on a dozen of them. Follow Jérôme on Twitter at @jpetazzo.

Pablo Tomas Fernandez Zavalia is an electronic engineer and writer. He started his career in developing for the City of Buenos Aires City Hall (buenosaires.gob.ar). After graduating, he joined British Telecom as head of the Web Services department in Argentina. He then worked on IBM as a database administrator, where he also did tutoring, DevOps, and cloud migrations. In his free time he enjoys writing, sailing and board games. Follow Tomas on Twitter at @tomfernblog.

1 Using Docker for Development and CI/CD

In 2013, Solomon Hykes showed a demo of the first version of Docker during the PyCon conference in Santa Clara[2]. Since then, the benefits of Docker containers have spread to seemingly every corner of the software industry. While Docker (the project and the company) made containers so popular, they were not the first project to leverage containers out there; and they are definitely not the last either.

Several years later, we can hopefully see beyond the hype as some powerful, efficient patterns emerged to leverage containers to develop and ship better software, faster.

In this chapter, you will first learn about the kind of benefits that you can expect from implementing Docker containers.

Then, a realistic roadmap that any organization can follow realistically, to attain these benefits.

1.1 Benefits of Using Docker

Containers will not instantly turn our monolithic, legacy applications into distributed, scalable microservices.

Containers will not transform overnight all our software engineers into "DevOps engineers". Notably, because DevOps is not defined by our tools or skills, but rather by a set of practices and cultural changes.

So what can containers do for us?

1.1.1 Set up Development Environments in Minutes

Using Docker and its companion tool Compose, you can run a complex app locally, on any machine, in less than five minutes.

It sums up to:

```
$ git clone https://github.com/jpetazzo/dockercoins
$ cd dockercoins
$ docker-compose up
```

[2]The future of Linux Containers (2013), *https://www.youtube.com/watch?v=wW9CAH9nSLs*

You can run these three lines on any machine where Docker is installed (Linux, macOS, Windows), and in a few minutes, you will get the DockerCoins demo app up and running. DockerCoins was created in 2015; it has multiple components written in Python, Ruby, and Node.js, as well as a Redis store. Years later, without changing anything in the code, we can still bring it up with the same three commands.

This means that onboarding a new team member, or switching from a project to another, can now be quick and reliable. It doesn't matter if DockerCoins is using Python 2.7 and Node.js 8 while your other apps are using Python 3 and Node.js 10, or if your system is using even different versions of these languages; each container is perfectly isolated from the others and from the host system.

We will see how to get there.

1.1.2 Deploy Easily in the Cloud or on Premises

After we build container images, we can run them consistently on any server environment. Automating server installation would usually require steps (and domain knowledge) specific to our infrastructure. For instance, if we are using AWS EC2, we may use AMI (Amazon Machine Images), but these images are different (and built differently) from the ones used on Azure, Google Cloud, or a private OpenStack cluster.

Configuration management systems (like Ansible, Chef, Puppet, or Salt) help us by describing our servers and their configuration as manifests that live in version-controlled source repositories. This helps, but writing these manifests is no easy task, and they don't guarantee reproducible execution. These manifests have to be adapted when switching distributions, distribution versions, and sometimes even from a cloud provider to another, because they would use different network interfaces or disk naming, for instance.

Once we have installed the Docker Engine (the most popular option), it can run any container image and effectively abstract these environment discrepancies.

The ability to stage new environments easily and reliably gives us exactly what we need to set up CI/CD (continuous integration and continuous delivery). We will see how to get there. Ultimately, it means that advanced techniques, such as blue/green deployments, or immutable infrastructure, become accessible to us, instead of being a privilege of larger organizations able to spend a lot of time to build their perfect custom tooling.

1.1.3 Less Risky Releases

Containers can help us to reduce the risks associated with a new release.

When we start a new version of our app by running the corresponding container image, if something goes wrong, rolling back is very easy. All we have to do is stop the container, and restart the previous version. The image for the previous version will still be around and will start immediately.

This is way safer than attempting a code rollback, especially if the new version implied some dependency upgrades. Are we sure that we can downgrade to the previous version? Is it still available on the package repositories? If we are using containers, we don't have to worry about that, since our container image is available and ready.

This pattern is sometimes called **immutable infrastructure**, because instead of changing our services, we deploy new ones. Initially, immutable infrastructure happened with virtual machines: each new release would happen by starting a new fleet of virtual machines. Containers make this even easier to use.

As a result, we can deploy with more confidence, because we know that if something goes wrong, we can easily go back to the previous version.

1.2 A Roadmap to Adopting Docker

The following roadmap works for organizations and teams of all sizes, regardless of their existing knowledge of containers. Even better, this roadmap will give you tangible benefits at each step, so that the gains realized give you more confidence in the whole process.

Sounds too good to be true?

Here is the quick overview, before we dive into the details:

1. Write one Dockerfile. Pick a service where this will have the most impact.
2. Write more Dockerfiles. The goal is to get the whole application in containers.
3. Write a Compose file. Now anyone can get this app running on their machine in minutes.
4. Make sure that all developers are on board. They should all have a Docker setup in good condition.
5. Use this to facilitate quality assurance (QA) and end-to-end testing.
6. Automate this process: congratulations, you are now doing continuous deployment to staging.

7. The last logical step is continuous deployment to production.

Each step is a self-contained iteration. Some steps are easy, others are more work; but each of them will improve your workflow.

1.2.1 Choosing the First Project to Dockerize

A good candidate for our first Dockerfile is a service that is a pain in the neck to build, and moves quickly. For instance, that new Rails app that we're building, and where we're adding or updating dependencies every few days as we're adding features. Pure Ruby dependencies are fine, but as soon as we rely on a system library, we will hit the infamous "works on my machine (not on yours)" problem, between the developers who are on macOS, and those who are on Linux, for instance. Docker will help with that.

Another good candidate is an application that we are refactoring or updating, and where we want to make sure that we are using the latest version of the language or framework; without breaking the environment for everything else.

If we have a component that is tricky enough to require a tool like Vagrant to run on our developer's machines, it's also a good hint that Docker can help there. While Vagrant is an amazing product, there are many scenarios where maintaining a Dockerfile is easier than maintaining a Vagrantfile; and running Docker is also easier and lighter than running Vagrant boxes.

1.2.2 Writing the First Dockerfile

There are various ways to write your first Dockerfile, and none of them is inherently right or wrong. Some people prefer to follow the existing environment as closely as possible. For example, if you're currently using PHP 7.2 with Apache 2.4, and have some very specific Apache configuration and `.htaccess` files? Sure, it makes sense to put that in containers. But if you prefer to start anew from your `.php` files, serve them with PHP FPM, and host the static assets from a separate NGINX container, that's fine too. Either way, the official PHP images got us covered.

During this phase, we'll want to make sure that the team working on that service has Docker installed on their machine, but only a few people will have to meddle with Docker at this point. They will be leveling the field for everyone else.

Here's an example `Dockerfile`, for the `hasher` microservice that's part of DockerCoins demo, written in Ruby:

```
FROM ruby
RUN gem install sinatra
RUN gem install thin
ADD hasher.rb /
CMD ["ruby", "hasher.rb"]
EXPOSE 80
```

Once we have a working Dockerfile for an app, we can start using this container image as the official development environment for this specific service or component. If we pick a fast-moving one, we will see the benefits very quickly, since Docker makes library and other dependency upgrades completely seamless. Rebuilding the entire environment with a different language version now becomes effortless. And if we realize after a difficult upgrade that the new version doesn't work as well, rolling back is just as easy and instantaneous, because Docker keeps a cache of previous image builds around.

1.2.3 Writing More Dockerfiles

The next step is to get the entire application in containers.

Note that we're not talking about production yet, and even if your first experiments go so well that you want to roll out some containers to production, you can do so selectively, only for some components. In particular, it is advised to keep databases and other stateful services outside of containers until you gain more operational experience.

But in development, we want everything in containers, including the precious databases, because the ones sitting on our developers' machines don't, or shouldn't, contain any precious data anyway.

We will probably have to write a few more Dockerfiles, but for standard services like Redis, MySQL, PostgreSQL, MongoDB, and many more, we will be able to use standard images from the Docker Hub. These images often come with special provisions to make them easy to extend and customize; for instance the official PostgreSQL image will automatically run `.sql` files placed in the suitable directory to pre-load our database with table structure or sample data.

Once we have Dockerfiles (or images) for all the components of a given application, we're ready for the next step.

1.2.4 Writing a Docker Compose File

A `Dockerfile` makes it easy to build and run a single container; a Docker Compose file makes it easy to build and run a stack of multiple containers.

So once each component runs correctly in a container, we can describe the whole application with a Compose file.

Here's what `docker-compose.yml` for DockerCoins demo looks like:

```
rng:
    build: rng
    ports:
      - "8001:80"

hasher:
    build: hasher
    ports:
      - "8002:80"

webui:
    build: webui
    links:
      - redis
    ports:
      - "8000:80"
    volumes:
      - "./webui/files/:/files/"

redis:
    image: redis

worker:
    build: worker
    links:
      - rng
      - hasher
      - redis
```

This gives us the very simple workflow that we mentioned earlier:

```
$ git clone https://github.com/jpetazzo/dockercoins
$ cd dockercoins
$ docker-compose up
```

Compose will analyze the file `docker-compose.yml`, pull the required images, and build the ones that need to. Then it will create a private bridge network

for the application, and start all the containers in that network. Why use a private network for the application? Isn't that a bit overkill?

Since Compose will create a new network for each app that it starts, this lets us run multiple apps next to each other (or multiple versions of the same app) without any risk of interference.

This pairs with Docker's service discovery mechanism, which relies on DNS. When an application needs to connect to, say, a Redis server, it doesn't need to specify the IP address of the Redis server, or its FQDN. Instead, it can just use `redis` as the server host name. For instance, in PHP:

```
$redis = new Redis();
$redis->connect('redis', 6379);
```

Docker will make sure that the name `redis` resolves to the IP address of the Redis container in the current network. So multiple applications can each have a `redis` service, and the name `redis` will resolve to the right one in each network.

1.2.5 A Standardized Development Environment

Once we have that Compose file, it's a good time to make sure that everyone is on board; i.e. that all our developers have a working installation of Docker. Windows and Mac users will find this particularly easy thanks to Docker Desktop.

Our team will need to know a few Docker and Compose commands; but in many scenarios, they will be fine if they only know `docker-compose up --build`. This command will make sure that all images are up-to-date, and run the whole application, showing its log in the terminal. If we want to stop the app, all we have to do is hit `Ctrl-C`.

At this point, we are already benefiting immensely from Docker and containers: everyone gets a consistent development environment, up and running in minutes, independently of the host system.

For simple applications that don't need to span multiple servers, this would almost be good enough for production; but we don't have to go there yet, as there are other fields where we take advantage of Docker without the high stakes associated with production.

1.2.6 End-To-End Testing and QA

When we want to automate a task, it's a good idea to start by having it done by a human, and write down the necessary steps. In other words: do things manually first, but document them. Then, these instructions can be given to another person, who will execute them. That person will probably ask us some clarifying questions, which will allow us to refine our manual instructions.

Once these manual instructions are perfectly accurate, we can turn them into a program (a simple script will often suffice) that we can then execute automatically.

Follow these principles to deploy test environments, and execute CI (Continuous Integration) and end-to-end testing, depending on the kind of tests that you use in your organization. Even if you don't have automated testing, you surely have some kind of testing happening before you ship a feature, even if it's just someone messing around with the app in staging before your users see it.

In practice, this means that we will document and then automate the deployment of our application, so that anyone can get it up and running by running a script.

Our final deployment scripts will be way simpler to write and to run than full-blown configuration management manifests, VM images, and so on.

If we have a QA team, they are now empowered to test new releases without relying on someone else to deploy the code for them.

If you're doing any kind of unit testing or end-to-end testing, you can now automate these tasks as well, by following the same principle as we did to automate the deployment process.

We now have a whole sequence of actions: building images, starting containers, executing initialization or migration hooks, and running tests. From now on, we will call this the *pipeline*, because all these actions have to happen in a specific order, and if one of them fails, we don't execute the subsequent stages.

1.2.7 Continuous Deployment to Staging

The next step is to run our pipeline automatically when we push changes to the code repository.

A CI/CD system like Semaphore can connect to GitHub, and run the pipeline each time someone opens, or updates, a pull request. The same or a modified

pipeline can also run on a specific branch, or a specific set of branches.

Each time there are relevant changes, our pipeline will automatically perform a sequence similar to the following:

- Build new container images;
- Run unit tests on these images (if applicable);
- Deploy them in a temporary environment;
- Run end-to-end tests on the application;
- Make the application available for human testing.

Further in this book we will see how to actually go and implement such a pipeline.

Note that we still don't require container orchestration for all of this to work. If our application in a staging environment can fit on a single machine, we don't need to worry about setting up a cluster, yet. In fact, thanks to Docker's layer system, running side-by-side images that share a common ancestry, which will be the case for images corresponding to successive versions of the same component, is very disk- and memory-efficient; so there is a good chance that we will be able to run many copies of our app on a single Docker Engine.

But this is also the right time to start looking into orchestration, and a platform like Kubernetes. Again, at this stage we don't need to roll that out straight to production; but we could use one of these orchestrators to deploy the staging versions of our application.

This will give us a low-risk environment where we can ramp up our skills on container orchestration and scheduling, while having the same level of complexity, minus the volume of requests and data, that our production environment.

1.2.8 Continuous Deployment to Production

It might be a while before we go from the previous stage to the next, because we need to build confidence and operational experience.

However, at this point, we already have a continuous deployment pipeline that takes every pull request (or every change in a specific branch or set of branches) and deploys the code on a staging cluster, in a fully automated way.

Of course, we need to learn how to collect logs, and metrics, and how to face minor incidents and major outages; but eventually, we will be ready to extend our pipeline all the way to the production environment.

1.3 Summary

Building a delivery pipeline with new tools from scratch is certainly a lot of work. But with the roadmap described above, we can get there one step at a time, while enjoying concrete benefits at each step.

In the next chapter, we will learn about deploying code to Kubernetes, including strategies that might not have been possible in your previous technology stack.

2 Deploying to Kubernetes

When getting started with Kubernetes, one of the first commands you learn and use is generally `kubectl run`. Folks who have experience with Docker tend to compare it to `docker run` and think: "Ah, this is how I can simply run a container!"

As it turns out, when you use Kubernetes, you don't simply run a container.

The way in which Kubernetes handles containers depends heavily on which version you are running [3]. You can check the server version with:

```
$ kubectl version
```

Kubernetes containers on versions 1.17 and lower

When using a version *lower* than 1.18, look at what happens after running a very basic `kubectl run` command:

```
$ kubectl run web --image=nginx
deployment.apps/web created
```

Alright! Then you check what was created on the cluster, and ...

```
$ kubectl get all
NAME                       READY STATUS  RESTARTS AGE
pod/web-65899c769f-dhtdx   1/1   Running 0        11s

NAME                 TYPE      CLUSTER-IP  EXTERNAL-IP PORT(S) AGE
service/kubernetes   ClusterIP 10.96.0.1   1.2.3.4     443/TCP 46s

NAME                  DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deployment.apps/web   1       1       1          1         11s

NAME                               DESIRED CURRENT READY AGE
replicaset.apps/web-65899c769f     1       1       1     11s
```

"I just wanted a container! Why do I get three different objects?"

Instead of getting a container, you got a whole zoo of unknown beasts:

- a *deployment* (called `web` in this example),
- a *replicaset* (`web-65899c769f`),

[3] At the time of writing, all major cloud vendors provide managed Kubernetes at versions 1.19 and 1.20. This book is based on and has been tested with those versions.

- a *pod* (`web-65899c769f-dhtdx`).

Note: you can ignore the *service* named `kubernetes` in the example above; that one already existed before the `kubectl run` command.

Kubernetes containers in versions 1.18 and higher

When you are running version 1.18 or *higher*, Kubernetes does indeed create a single pod. Look how different Kubernetes acts on newer versions:

```
$ kubectl run web --image=nginx
pod/web created
```

As you can see, more recent Kubernetes versions behave pretty much in line with what seasoned Docker users would expect. Notice that no deployments or replicasets are created:

```
$ kubectl get all
NAME      READY STATUS  RESTARTS AGE
pod/web   1/1   Running 0        3m14s

NAME                 TYPE      CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes   ClusterIP 10.96.0.1  1.2.3.4     443/TCP 4m16s
```

So, if we want to create a deployment we must be more explicit. This command works as expected on all Kubernetes versions:

```
$ kubectl create deployment web --image=nginx
deployment.apps/web created
```

The bottom line is that we should always use the most explicit command available to future-proof our deployments.

Next, you'll learn the roles of these different objects and how they are essential to zero-downtime deployments in Kubernetes.

Continuous integration gives you confidence that your code works. To extend that confidence to the release process, your deployment operations need to come with a safety belt too.

2.1 Containers and Pods

In Kubernetes, the smallest unit of deployment is not a container; it's a **pod**. A pod is just a group of containers (which can also be a group of *one* container) that runs on the same machine and shares a few things together.

For instance, the containers within a pod can communicate with each other over `localhost`. From a network perspective, all the processes in these containers are local.

But you can never create a standalone container: the closest you can do is create a pod with a single container in it.

That's what happens here: when you tell Kubernetes, "create me an NGINX!", you're really saying, *"I would like a pod, in which there should be a single container, using the* ***nginx*** *image."*

```
# pod-nginx.yml
# Create it with:
#    kubectl apply -f pod-nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          name: http
```

Alright, then, why doesn't it just have a pod? Why the replica set and deployment?

2.2 Declarative vs Imperative Systems

Kubernetes is a **declarative system** (which is the opposite of an imperative system). This means that you can't give it orders. You can't say, "Run this container." All you can do is describe what you want to have and wait for Kubernetes to take action to reconcile what you have, with what you want to have.

In other words, you can say, *"I would like a 40-feet long blue container with yellow doors"*, and Kubernetes will find such a container for you. If it doesn't exist, it will build it; if there is already one but it's green with red doors, it will paint it for you; if there is already a container of the right size and color, Kubernetes will do nothing, since *what you have* already matches *what you want*.

In software container terms, you can say, *"I would like a pod named* `web`*, in which there should be a single container, that will run the* `nginx` *image."*

If that pod doesn't exist yet, Kubernetes will create it. If that pod already exists and matches your spec, Kubernetes doesn't need to do anything.

With that in mind, how do you scale your `web` application, so that it runs in multiple containers or pods?

2.3 Replica Sets Make Scaling Pods Easy

If all you have is a pod, and you want more identical pods, all you can do is get back to Kubernetes and tell it, *"I would like a pod named* `web2`*, with the following specification: . . . "* and re-use the same specification as before. Then, repeat this as many times as you want to have pods.

This is rather inconvenient, because it is now your job to keep track of all these pods, and to make sure that they are all in sync, using the same specification.

To make things simpler, Kubernetes gives you a higher level construct: the **replica set**. The specification of a replica set looks very much like the specification of a pod, except that it carries a number indicating how many *replicas*—i.e. pods with that particular specification—you want.

So you tell Kubernetes, *"I would like a replica set named* `web`*, which should have 3 pods, all matching the following specification: . . . "* and Kubernetes will accordingly make sure that there are exactly three matching pods. If you start from scratch, the three pods will be created. If you already have three pods, nothing is done because *what you have* already matches *what you want*.

```
# pod-replicas.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: web-replicas
  labels:
    app: web
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
```

```
    labels:
      app: web
      tier: frontend
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Replica sets are particularly relevant for scaling and high availability.

Scaling is relevant because you can update an existing replica set to change the desired number of replicas. As a consequence, Kubernetes will create or delete pods so it will be the exact desired number in the end.

For high availability, it is relevant because Kubernetes will continuously monitor what's going on in the cluster. It will ensure that no matter what happens, you still have the desired number.

If a node goes down, taking one of the `web` pods with it, Kubernetes creates another pod to replace it. If it turns out that the node wasn't down, but merely unreachable or unresponsive for a while, you may have one extra pod when it comes back. Kubernetes will then terminate a pod to make sure that you still have the exact requested number.

What happens, however, if you want to change the definition of a pod within your replica set? For instance, what happens when you want to switch the image that you are using with a newer version?

Remember: the mission of the replica set is, *"Make sure that there are N pods matching this specification."* What happens if you change that definition? Suddenly, there are zero pods matching the new specification.

By now you know how a declarative system is supposed to work: Kubernetes should immediately create N pods matching your new specification. The old pods would just stay around until you clean them up manually.

It makes a lot of sense for these pods to be removed cleanly and automatically in a CI/CD pipeline, as well as for the creation of new pods to happen in a more gradual manner.

2.4 Deployments Drive Replica Sets

It would be nice if pods could be removed cleanly and automatically in a CI/CD pipeline and if the creation of new pods could happen in a more gradual manner.

This is the exact role of **deployments** in Kubernetes. At a first glance, the specification for a deployment looks very much like the one for a replica set: it features a pod specification, a number of replicas, and a few additional parameters that you'll read about later in this guide.

```yaml
# deployment-nginx.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Deployments, however, don't create or delete pods directly. They delegate that work to one or more replica sets.

When you create a deployment, it creates a replica set, using the exact pod specification that you gave it.

When you update a deployment and adjust the number of replicas, it passes that update down to the replica set.

2.4.1 What Happens When You Change Configuration

When you need to update the pod specification itself, things get interesting. For instance, you might want to change the image you're using (because you're

releasing a new version), or the application's parameters (through command-line arguments, environment variables, or configuration files).

When you update the pod specification, the deployment creates a new replica set with the updated pod specification. That replica set has an initial size of zero. Then, the size of that replica set is progressively increased, while decreasing the size of the other replica set.

You could imagine that you have a sound mixing board in front of you, and you are going to fade in (turn up the volume) on the new replica set while fading out (turn down the volume) on the old one.

During the whole process, requests are sent to pods of both the old and new replica sets, without any downtime for your users.

That's the big picture, but there are many little details that make this process even more robust.

2.5 Detecting Broken Deployments with Readiness Probes

If you roll out a broken version, it could bring the entire application down, as Kubernetes will steadily replace your old pods with the new (broken) version, one at a time.

Unless you use **readiness probes**.

A readiness probe is a test that you can add to a container specification. It's a binary test that can only say "IT WORKS" or "IT DOESN'T," and will be executed at regular intervals. By default, it executes every 10 seconds.

Kubernetes supports three ways of implementing readiness probes:

1. Running a command inside a container;
2. Making an HTTP(S) request against a container; or
3. Opening a TCP socket against a container.

Kubernetes uses the result of that test to know if the container and the pod that it's a part of is ready to receive traffic. When you roll out a new version, Kubernetes will wait for the new pod to mark itself as "ready" before moving on to the next one.

If a pod never reaches the ready state because the readiness probe keeps failing, Kubernetes will never move on to the next. The deployment stops, and your

application keeps running with the old version until you address the issue.

Note: if there is no readiness probe, then the container is considered as ready, as long as it could be started. So make sure that you define a readiness probe if you want to leverage that feature!

2.6 Rollbacks for Quick Recovery from Bad Deploys

At any point in time, during the rolling update or even later, you can tell Kubernetes: *"Hey, I changed my mind; please go back to the previous version of that deployment."* It will immediately switch the roles of the "old" and "new" replica sets. From that point, it will increase the size of the old replica set (up to the nominal size of the deployment), while decreasing the size of the other one.

Generally speaking, this is not limited to two "old" and "new" replica sets. Under the hood, there is one replica set that is considered "up-to-date" and that you can think of as the "target" replica set. That's the one that you're trying to move to; that's the one that Kubernetes will progressively scale up. Simultaneously, there can be any number of other replica sets, corresponding to older versions.

As an example, you might run version 1 of an application over 10 replicas. Then you'd start rolling out version 2. At some point, you might have seven pods running version 1, and three pods running version 2. You might then decide to release version 3 without waiting for version 2 to be fully deployed (because it fixes an issue that wasn't noticed earlier). And while version 3 is being deployed, you might decide, after all, to go back to version 1. Kubernetes will merely adjust the sizes of the replica sets (corresponding to versions 1, 2, and 3 of the application) accordingly.

2.7 MaxSurge and MaxUnavailable

Kubernetes doesn't exactly update deployments one pod at a time. Earlier, you learned that deployments had "a few extra parameters": these parameters include `MaxSurge` and `MaxUnavailable`, and they indicate the pace at which the update should proceed.

You could imagine two strategies when rolling out new versions. You could be conservative about your application availability, and decide to start new pods before shutting down old ones. Only after a new pod is up, running, and ready, can you terminate an old one.

This, however, implies that you have some spare capacity available on our cluster. It might be the case that you can't afford to run any extra pod, because your cluster is full to the brim, and that you prefer to shutdown an old pod before starting a new one.

`MaxSurge` indicates how many extra pods you are willing to run during a rolling update, while `MaxUnavailable` indicates how many pods you can lose during the rolling update. Both parameters are specific to a deployment: each deployment can have different values for them. Both parameters can be expressed as an absolute number of pods, or as a percentage of the deployment size; and both parameters can be zero, but not at the same time.

Below, you'll find a few typical values for MaxSurge and MaxUnavailable and what they mean.

Setting MaxUnavailable to 0 means, *"do not shutdown any old pod before a new one is up and ready to serve traffic."*

Setting MaxSurge to 100% means, *"immediately start all the new pods"*, implying that you have enough spare capacity on your cluster and that you want to go as fast as possible.

The default values for both parameters are 25%, meaning that when updating a deployment of size 100, 25 new pods are immediately created, while 25 old pods are shutdown. Each time a new pod comes up and is marked ready, another old pod can be shutdown. Each time an old pod has completed its shut down and its resources have been freed, another new pod can be created.

2.8 Quick Demo

It's easy to see these parameters in action. You don't need to write custom YAML, define readiness probes, or anything like that.

All you have to do is to tell a deployment to use an invalid image; for instance an image that doesn't exist. The containers will never be able to come up, and Kubernetes will never mark them as "ready."

If you have a Kubernetes cluster (a one-node cluster like minikube or Docker Desktop is fine), you can run the following commands in different terminals to watch what is going to happen:

- `kubectl get pods -w`
- `kubectl get replicasets -w`
- `kubectl get deployments -w`

- `kubectl get events -w`

Then, create, scale, and update a deployment with the following commands:

```
$ kubectl create deployment web --image=nginx
$ kubectl scale deployment web --replicas=10
$ kubectl set image deployment web nginx=invalid-image
```

You can see that the deployment is stuck, but 80% of the application's capacity is still available.

If you run `kubectl rollout undo deployment web`, Kubernetes will go back to the initial version, running the `nginx` image.

2.9 Selectors and Labels

It turns out that the job of a replica set, as mentioned earlier, is to make sure that there are exactly N pods matching the right specification, that's not exactly what's going on. Actually, the replica set doesn't look at the pods' specifications, but only at their **labels**.

In other words, it doesn't matter if the pods are running `nginx` or `redis` or whatever; all that matters is that they have the right labels. In the examples in the beginning of the chapter, these labels would look like `run=web` and `pod-template-hash=xxxyyyzzz`.

A replica set contains a *selector*, which is a logical expression that "selects" a number of pods, just like a `SELECT` query in SQL. The replica set makes sure that there is the right number of pods, creating or deleting pods if necessary; but it doesn't change existing pods.

Just in case you're wondering: yes, it is absolutely possible to manually create pods with these labels, but running a different image or with different settings, and fool your replica set.

At first, this could sound like a big potential problem. In practice though, it is very unlikely that you would accidentally pick the "right" (or "wrong", depending on the perspective) labels, because they involve a hash function on the pod's specification that is all but random.

2.9.1 Services as Load Balancers

Selectors are also used by **services**, which act as load balancers of Kubernetes traffic, internal and external. You can create a service for the `web` deployment

with the following command:

```
$ kubectl expose deployment web --port=80
```

The service will have its own internal IP address (denoted by the name `ClusterIP`) and an optional external IP, and connections to these IP addresses on port 80 will be load-balanced across all the pods of this deployment.

In fact, these connections will be load-balanced across all the pods matching the service's selector. In that case, that selector will be `run=web`.

When you edit the deployment and trigger a rolling update, a new replica set is created. This replica set will create pods, whose labels will include, among others, `run=web`. As such, these pods will receive connections automatically.

This means that during a rollout, the deployment doesn't reconfigure or inform the load balancer that pods are started and stopped. It happens automatically through the selector of the service associated with the load balancer.

If you're wondering how probes and health checks play into this, a pod is added as a valid endpoint for a service only if all its containers pass their readiness check. In other words, a pod starts receiving traffic only once it's actually ready for it.

2.10 Advanced Kubernetes Deployment Strategies

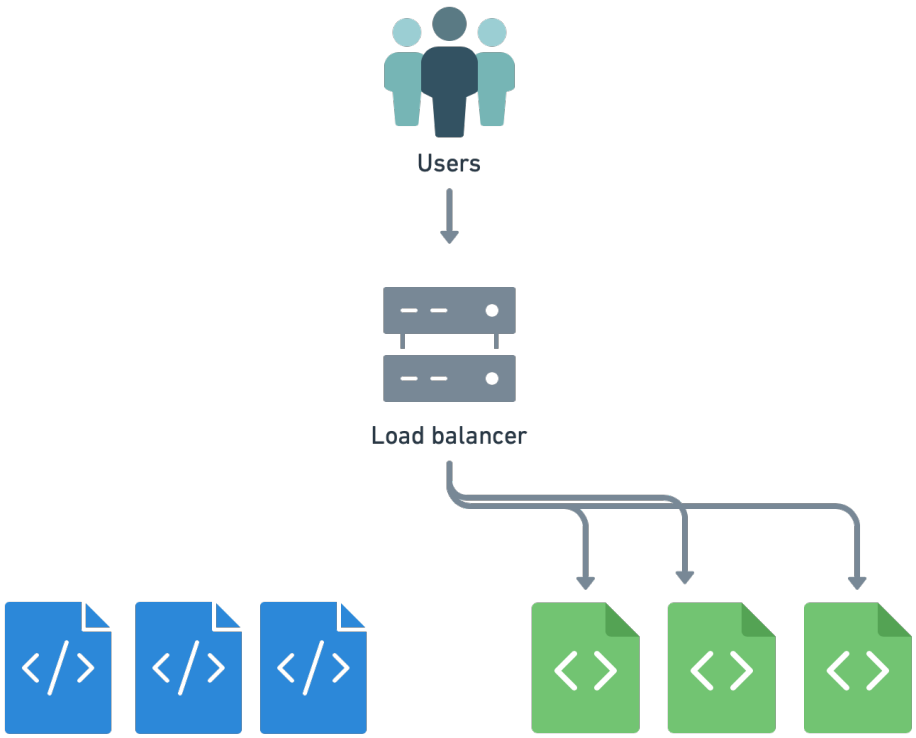
Sometimes, you might want even more control when you roll out a new version.

Two popular techniques are **blue/green deployment** and **canary deployment**.

2.10.1 Blue / Green Deployment

In blue/green deployment, you want to instantly switch over all the traffic from the old version to the new, instead of doing it progressively like explained previously. There could be a few reasons to do that, including:

- You don't want a mix of old and new requests, and you want the break from one version to the next to be as clean as possible.
- You are updating multiple components (say, web frontend and API backend) together, and you don't want the new version of the web frontend to talk to the old version of the API backend or vice versa.
- If something goes wrong, you want the ability to revert as fast as possible, without even waiting for the old set of containers to restart.



You can achieve blue/green deployment by creating multiple deployments (in the Kubernetes sense), and then switching from one to another by changing the selector of our service.

Let's see how this would work in a quick demo.

The following commands will create two deployments `blue` and `green`, respectively using the `nginx` and `httpd` container images:

```
$ kubectl create deployment blue --image=nginx
$ kubectl create deployment green --image=httpd
```

Then, you create a service called `web`, which initially won't send traffic anywhere:

```
$ kubectl create service clusterip web --tcp=80
```

Note: when running a local development Kubernetes cluster, such as MiniKube[4] or the one bundled with Docker Desktop, you'll wish to change the previous command to: `kubectl create service nodeport web --tcp=80`.

[4]The official local Kubernetes cluster for macOS, Linux, and Windows for testing and development. *https://minikube.sigs.k8s.io/docs/*

The NodePort type of service is easier to access locally as the service ports are forwarded to `localhost` automatically. To see this port mapping run `kubectl get services`.

Now, you can update the selector of the service `web` by running `kubectl edit service web`. This will retrieve the definition of service `web` from the Kubernetes API, and open it in a text editor. Look for the section that says:

```
selector:
  app: web
```

Replace `web` with `blue` or `green`, to your liking. Save and exit. `kubectl` will push your updated definition back to the Kubernetes API, and voilà! Service `web` is now sending traffic to the corresponding deployment.

You can verify for yourself by retrieving the IP address of that service with `kubectl get svc web` and connecting to that IP address with `curl`.

The modification that you did with a text editor can also be done entirely from the command line, using for instance `kubectl patch` as follows:

```
$ kubectl patch service web \
  -p '{"spec": {"selector": {"app": "green"}}}'
```

The advantage of blue/green deployment is that the traffic switch is almost instantaneous, and you can roll back to the previous version just as fast by updating the service definition again.

2.10.2 Canary Deployment

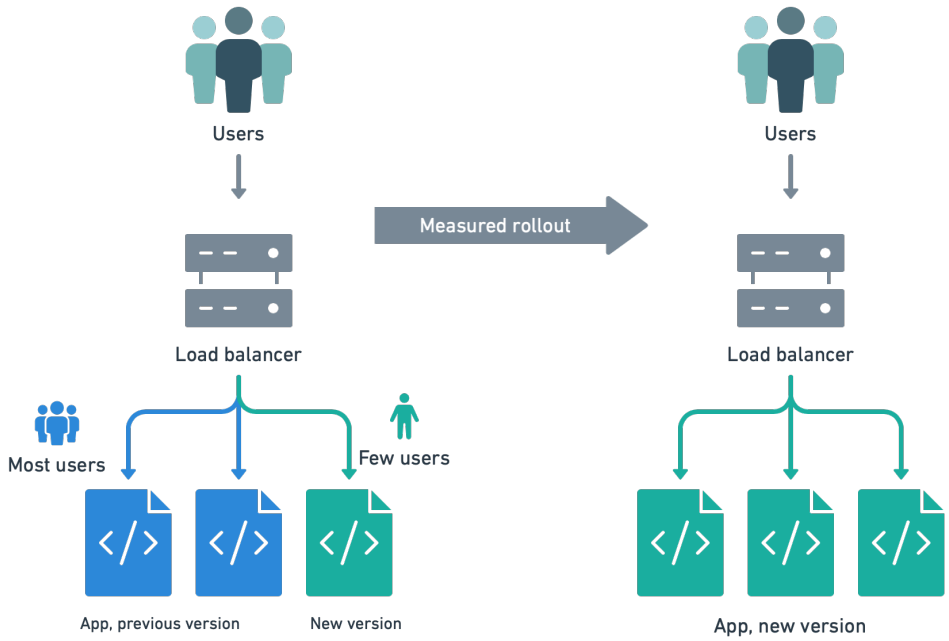
Canary deployment alludes to the canaries that were used in coal mines, to detect dangerous concentrations of toxic gas like carbon monoxide. Canaries are more sensitive to toxic gas than humans. The miners would carry a canary in a cage. If the canary passed out, it meant that the miners had reached a dangerous area and should head back before they would pass out too.

How does that map to software deployment?

Sometimes, you can't (or won't) afford to affect all your users with a flawed version, even for a brief period of time. So instead, you do a partial rollout of the new version. For instance, you could deploy a couple of replicas running the new version, or you send 1% of your users to that new version.

Then, you compare metrics between the current version and the canary that you just deployed. If the metrics are similar, you can proceed. If latency, error

rates, or anything else looks wrong, you roll back.



This technique, which would be fairly involved to set up, ends up being relatively straightforward thanks to Kubernetes' native mechanisms of labels and selectors.

It's worth noting that in the previous example, we changed the service's selector, but it is also possible to change the pods' labels.

For instance, if a service's selector is set to look for pods with the label `status=enabled`, you can apply such a label to a specific pod with:

```
$ kubectl label pod fronted-aabbccdd-xyz status=enabled
```

You can apply labels *en masse* as well, for instance:

```
$ kubectl label pods -l app=blue,version=v1.5 status=enabled
```

And you can remove them just as easily:

```
$ kubectl label pods -l app=blue,version=v1.4 status-
```

2.11 Summary

You now know a few techniques that can be used to deploy with more confidence. Some of these techniques simply reduce the downtime caused by the deployment itself, meaning that you can deploy more often, without being afraid of affecting your users.

Some of these techniques give you a safety belt, preventing a bad version from taking down your service. And some others give you an extra peace of mind, like hitting the "SAVE" button in a video game before trying a particularly difficult sequence, knowing that if something goes wrong, you can always go back where you were.

Kubernetes makes it possible for developers and operation teams to leverage these techniques, which leads to safer deployments. If the risk associated with deployments is lower, it means that you can deploy more often, incrementally, and see more easily the results of your changes as we implement them; instead of deploying once a week or month, for instance.

The end result is a higher development velocity, lower time-to-market for fixes and new features, as well as better availability of your applications. Which is the whole point of implementing containers in the first place.

3 CI/CD Best Practices for Cloud-Native Applications

Engineering leaders strive to deliver bug-free products to customers as productively as possible. Today's cloud-native technology empowers teams to iterate, at scale, faster than ever. But to experience the promised agility, we need to change how we deliver software.

"CI/CD" stands for the combined practices of Continuous Integration (CI) and Continuous Delivery (CD). It is a timeless way of developing software in which you're able to release updates at any time in a sustainable way. When changing code is routine, development cycles are faster. Work is more fulfilling. Companies can improve their products many times per day and delight their customers.

In this chapter, we'll review the principles of CI/CD and see how we can apply them to developing cloud-native applications.

3.1 What Makes a Good CI/CD Pipeline

A good CI/CD pipeline is fast, reliable, and comprehensive.

3.1.1 Speed

Pipeline velocity manifests itself in several ways:

How quickly do we get feedback on the correctness of our work? If it's longer than the time it takes to get a cup of coffee, pushing code to CI becomes too distracting. It's like asking a developer to join a meeting in the middle of solving a problem. Developers will work less effectively due to context switching.

How long does it take us to build, test and deploy a simple code commit? Take a project with a total time of one hour to run CI and deployment and a team of about a dozen engineers. Such CI/CD runtime means that the entire team has a hard limit of up to six or seven deploys in a workday. In other words, there is less than one deploy per developer per day available. The team will settle on a workflow with less frequent and thus more risky deployments. This workflow is in stark contrast to the rapid iterations that businesses today need.

How quickly can we set up a new pipeline? Difficulty with scaling CI/CD

infrastructure or reusing existing configuration creates friction. You make the best use of the cloud by writing software as a composition of small services. Developers need new CI/CD pipelines often, and they need them fast. The best way to solve this is to let developers create and own CI/CD pipelines for their projects.

For this to happen, the CI/CD tool of choice should fit into the existing development workflows. Such a CI/CD tool should support storing all pipeline configuration as code. The team can review, version, and reuse pipelines like any other code. But most importantly, CI/CD should be easy to use for every developer. That way, projects don't depend on individuals or teams who set up and maintain CI for others.

3.1.2 Reliability

A reliable pipeline always produces the same output for a given input. And with consistent runtime. Intermittent failures cause intense frustration among developers.

Engineers like to do things independently, and they often opt to maintain their CI/CD system. But operating CI/CD that provides on-demand, clean, stable, and fast resources is a complicated job. What seems to work well for one project or a few developers usually breaks down later. The team and the number of projects grow as the technology stack changes. Then someone from management realizes that by delegating that task, the team could spend more time on the actual product. At that point, if not earlier, the engineering team moves from a self-hosted to a cloud-based CI/CD solution.

3.1.3 Completeness

Any increase in automation is a positive change. However, a CI/CD pipeline needs to run and visualize everything that happens to a code change — from the moment it enters the repository until it runs in production. This requires the CI/CD tool to be able to model both simple and, when needed, complex workflows. That way, manual errors are all but impossible.

For example, it's not uncommon to have the pipeline run only the build and test steps. Deployment remains a manual operation, often performed by a single person. This is a relic of the past when CI tools were unable to model delivery workflows.

Today a service like Semaphore provides features like:

- Secret management
- Multi-stage, parametrized pipelines
- Change detection
- Container registry
- Connections to multiple environments (staging, production, etc.)
- Audit log
- Test results

There is no longer a reason not to automate the entire software delivery process.

3.2 General Principles

3.2.1 Architect the System in a Way That Supports Iterative Releases

The most common reason why a system is unable to sustain frequent iterative releases is tight coupling between components.

When building (micro)services, the critical decisions are in defining their: 1) boundaries, 2) communication with the rest of the system.

Changing one service shouldn't require changing another. If one service goes down, other services or, worse, the system as a whole should not go down. Services with well-defined boundaries allow us to change a behavior in one place and release that change as quickly as possible.

We don't want a system where making one change requires changing code in many different places. This process is slow and prevents clear code ownership. Deploying more than one service at a time is risky.

A loosely coupled service contains related behavior in one place. It knows as little as possible about the rest of the system with which it collaborates.

A loosely coupled system is conservative in the design of communication between services. Services usually communicate by making asynchronous remote procedure calls (RPC). They use a small number of endpoints. There is no shared database, and all changes to databases are run iteratively as part of the CI/CD pipeline.

Metrics and monitoring are also an essential enabler of iterative development. Being able to detect issues in real-time gives us the confidence to make changes, knowing that we can quickly recover from any error.

3.2.2 You Build It, You Run It

In the seminal 2006 interview to ACM[5], Werner Vogels, Amazon CTO, pioneered the mindset of *you build it, you run it*. The idea is that developers should be in direct contact with the operation of their software, which, in turn, puts them in close contact with customers.

The critical insight is that involving developers in the customer feedback loop is essential for improving the quality of the service. Which ultimately leads to better business results.

Back then, that view was radical. The tooling required was missing. So only the biggest companies could afford to invest in building software that way.

Since then, the philosophy has passed the test of time. Today the best product organizations are made of small autonomous teams. They own the full lifecycle of their services. They have more freedom to react to feedback from users and make the right decisions quickly.

Being responsible for the quality of software requires being responsible for releasing it. This breaks down the silos between traditional developers and operations groups. Everyone must work together to achieve high-level goals.

It's not rare that in newly formed teams there is no dedicated operations person. Instead, the approach is to do "NoOps". Developers who write code also own the delivery pipeline. The cloud providers take care of hosting and monitoring production services.

3.2.3 Use Ephemeral Resources

There are three main reasons for using ephemeral resources to run your CI/CD pipeline.

The speed imperative demands CI/CD to not act as a bottleneck. It needs to scale to meet the growth of your team, applications, and test suites. A simple solution is to rely on a cloud service that automatically scales CI/CD pipelines on demand. Ideally, this would come at a pay-as-you-go pricing model, so that you only pay for what you use.

Ephemeral resources help ensure that your tests run consistently. Cloud-based CI/CD solutions run your code in clean and isolated environments. They are created on-demand and deleted as soon as the job has finished.

[5] A Conversation with Werner Vogels, ACMQueue *https://queue.acm.org/detail.cfm?id=1142065*

As we've seen in chapter 1, containers allow us to use one environment in development, CI/CD, and production. There's no need to set up and maintain infrastructure or sacrifice environmental fidelity.

3.2.4 Automate Everything

It's worth repeating: automate everything you can.

There are cases when complete automation is not possible. You may have customers who simply don't want continuous updates to their systems. There may be regulations restricting how software can be updated. This is the case, for example, in the aerospace, telecom, and medical industries.

But if these conditions do not apply and you still think that your pipeline can't be fully automated — you're almost certainly wrong.

Take a good look at your end-to-end process and uncover where you're doing things manually out of habit. Make a plan to make any changes that may be needed, and automate it.

3.3 Continuous Integration Best Practices

Getting the continuous integration process right is a prerequisite for successful continuous delivery. Usually, when the CI process is fast and reliable, the leap to full CI/CD is not hard to make.

3.3.1 Treat Master Build as If You're Going to Make a Release at Any Time

Small, simple, frequent changes are a less risky way of building software in a team than making big, complex, rare changes. This implies that the team will make fewer mistakes by always being ready for release, not more.

Your team's goal should be to get new code to production as soon as it's ready. And if something goes wrong — own it and handle it accordingly. Let the team grow through the sense of ownership of what they do.

Being always ready for a release requires a highly developed testing culture. A pull request with new code should always include automated tests. If it doesn't, then there's no point in moving fast to oblivion.

If you're starting a new project, invest time to bring everyone on the same page, and commit to writing automated tests for all code. Set up the entire CI/CD

pipeline, even while the application has no real functionality. The pipeline will discourage any manual or risky processes from creeping in and slowing you down in the future.

If you have an existing project with some technical debt, you can start by committing to a "no broken windows" policy on the CI pipeline. When someone breaks master, they should drop what they're doing and fix it.

Every test failure is a bug. It needs to be logged, investigated, and fixed. Assume that the defect is in application code unless tests can prove otherwise. However, sometimes the test itself is the problem. Then the solution is to rewrite it to be more reliable.

The process of cleaning up the master build usually starts as being frustrating. But if you're committed and stick to the process, over time, the pain goes away. One day you reach a stage when a failed test means there is a real bug. You don't have to re-run the CI build to move on with your work. No one has to impose a code freeze. Days become productive again.

3.3.2 Keep the Build Fast: Up to 10 Minutes

Let's take two development teams, both writing tests, as an example. Team A has a CI build that runs for about 3 minutes. Team B has a build that clocks at 45 minutes. They both use a CI service that runs tests on all branches. They both release reliable software in predictable cycles. But team A has the potential to build and release over 100 times in a day, while team B can do that up to 7 times. Are they both doing *continuous* integration?

The short answer is no.

If a CI build takes a long time, we approach our work defensively. We tend to keep branches on the local computer longer. Every developer's code is in a different state. Merges are rarer, and they become big and risky events. Refactoring becomes hard to do on the scale that the system needs to stay healthy.

With a slow build, every "git push" leads to a huge distraction. We either wait or look for something else to do to avoid being completely idle. And if we context-switch to something else, we know that we'll need to switch back again when the build is finished. The catch is that every task switch in programming is hard, and it sucks up our energy.

The point of *continuous* in continuous integration is speed. Speed drives high

productivity: we want feedback as soon as possible. Fast feedback loops keep us in a state of flow, which is the source of our happiness at work.

So, it's helpful to establish criteria for how fast should a CI process be:

Proper continuous integration is when it takes you less than 10 minutes from pushing new code to getting results.

The 10-minute mark is about how much a developer can wait without getting too distracted. It's also adopted by one of the pioneers of continuous delivery, Jez Humble. He performs the following informal poll at conferences[6].

First, he asks his audience to raise their hands if they do continuous integration. Usually, most of the audience raise their hands.

He then asks them to keep their hands up if everyone on their team commits and pushes to the master branch at least daily.

Over half the hands go down. He then asks them to keep their hands up if each such commit causes an automated build and test. Half the remaining hands are lowered.

Finally, he asks if, when the build fails, it's usually back to green within ten minutes.

With that last question, only a few hands remain. Those are the people who pass the informal CI certification test.

There are a couple of tactics which you can employ to reduce CI build time:

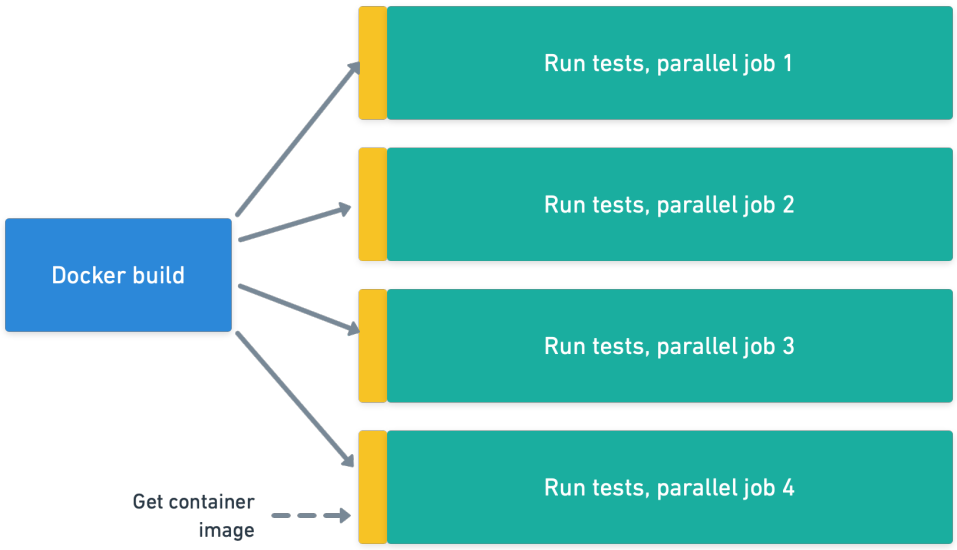
- **Caching**: Project dependencies should be independently reused across builds. When building Docker containers, use the layer caching feature to reuse known layers from the registry.
- **Built-in Docker registry**: A container-native CI solution should include a built-in registry. This saves a lot of money compared to using the registry provided by your cloud provider. It also speeds up CI, often by several minutes.
- **Test parallelization**: A large test suite is the most common reason why CI is slow. The solution is to distribute tests across as many parallel jobs as needed.
- **Change detection**: Large test suites can be dramatically sped up by only testing code that has changed since the last commit.

[6] What is Proper Continuous Integration, Semaphore https://semaphoreci.com/blog/2017/03/02/what-is-proper-continuous-integration.html

3.3.3 Build Only Once and Promote the Result Through the Pipeline

In the context of container-based services, this principle means building containers only once and then reusing the images throughout the pipeline.

For example, consider a case where you need to run tests in parallel and then deploy a container. The desired pipeline should build the container image in the first stage. The later stages of testing and deployment reuse the container from the registry. Ideally, the registry would be part of the CI service to save costs and avoid network overhead.



Building container once

The same principle applies to any other assets that you need to create from source code and use later. The most common are binary packages and website assets.

Besides speed, there is the aspect of reliability. The goal is to be sure that every automated test ran against the artifact that will go to production.

To support such workflows, your CI system should be able to:

- Execute pipelines in multiple stages.
- Run each stage in an identical, clean, and isolated environment.
- Version and upload the resulting artifact to an artifact or container storage system.

- Reuse the artifacts in later stages of the pipeline.

These steps ensure that the build doesn't change as it progresses through the system.

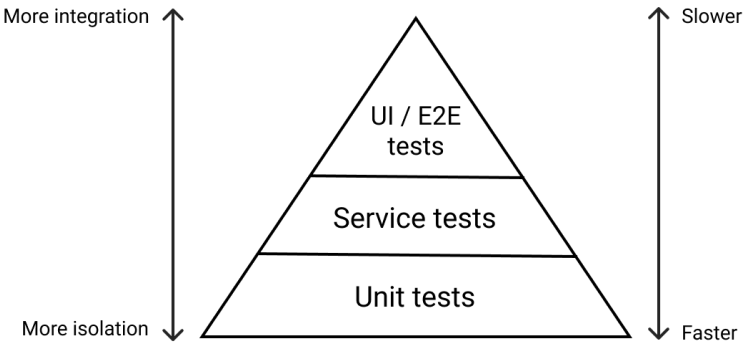
3.3.4 Run Fast and Fundamental Tests First

On many occasions, you can get all the feedback from CI that you need without running the entire test suite.

Unit tests run the fastest because they:

- Test small units of code in isolation from the rest of the system.
- Verify the core business logic, not behavior from the end-user perspective.
- Usually don't touch the database.

The **test pyramid** diagram is a common representation of the distribution of tests in a system:

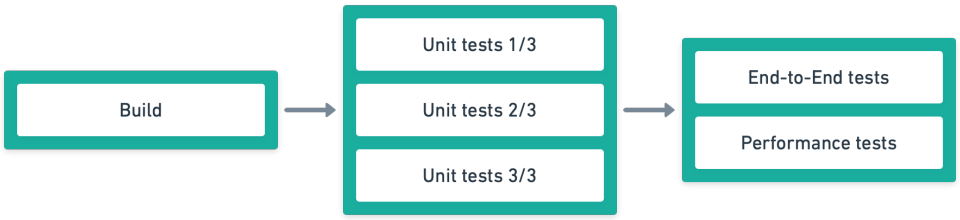


According to this strategy, a test suite has:

- The most unit tests.
- Somewhat less service-level tests, which include calls to the database and any other core external resource.
- Few user interfaces, or end-to-end tests. These serve to verify the behavior of the system as a whole, usually from the user's perspective.

If a team follows this strategy, a failing unit test is a signal of a fundamental problem. The remaining high-level and long-running tests are irrelevant until we resolve the problem.

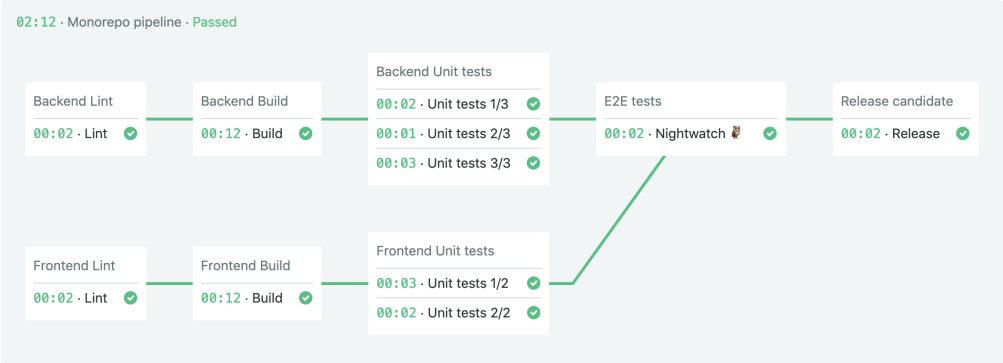
For these reasons, projects with test suites that run for anything longer than a minute should prioritize unit tests in the CI pipeline. For example, such a pipeline may look like this:



This strategy allows developers to get feedback on trivial errors in seconds. It also encourages all team members to understand the performance impact of individual tests as the code base grows.

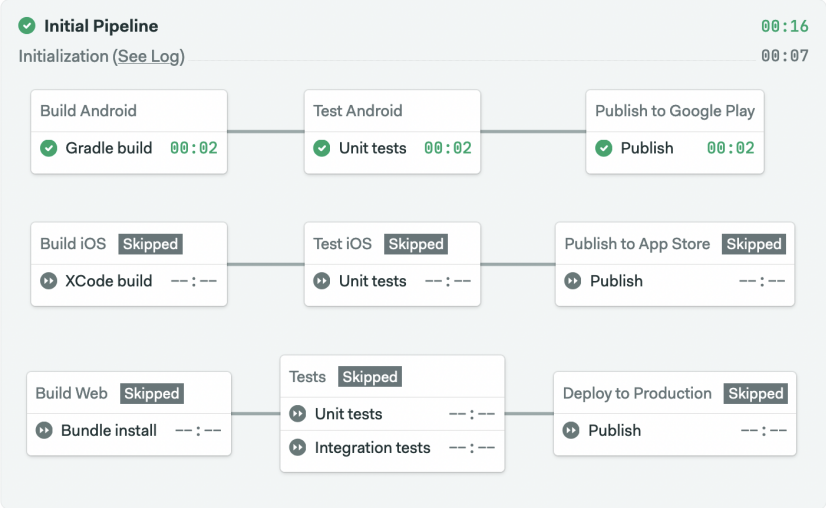
There are additional tactics that you can use with your CI system to get fast feedback:

Conditional stage execution lets you defer running certain parts of your build for the right moment. For example, you can configure your CI to run a subset of end-to-end tests only if a related component has changed.



In the pipeline above, backend and frontend tests run if code changed in the corresponding directories. End-to-end tests run if any of the two has passed and none has failed.

Change detection lets you skip steps in the pipeline when the underlying code has not changed. By running only the relevant tests for a given commit, you can speed up the pipeline and cut down costs.



A fail-fast strategy gives you instant feedback when a job fails. CI stops all currently running jobs in the pipeline as soon as one of the jobs has failed. This approach is particularly useful when running parallel jobs with variable duration.

Automatic cancelation of queued builds can help in situations when you push some changes, only to realize that you have made a mistake. So you push a new revision immediately but would then need to wait for twice as long for feedback. Using automatic cancelations, you can get feedback on revisions that matter while skipping the intermediate ones.

3.3.5 Minimize Feature Branches, Embrace Feature Flags

One of the reasons why Git overshadowed earlier version control systems like Subversion is that it made branching easy. This motivated developers to create and merge branches many times per day.

The point of making such short-lived branches is to work in isolation from the master, which we agreed to keep in a releasable state at all times. In a Git branch, developers can commit and save their work at any time. When they're done, they can squash all commits to form a nicely formatted changeset. Then they submit the changeset for feedback and eventually merge it. It's also common to call such branches "feature branches". In that context, the term is misleading.

That is not what this best practice is about.

By feature branches, we refer to branches that live for as long as a new product

feature is in development. Such branches do not live for hours, but months.

Working in a branch for so long opens the door to all the problems that come up with infrequent integration. Dependencies and internal APIs are likely to change. The amount of work and coordination needed to merge skyrockets. The difficulty is not just to merge code on a line-by-line level. It's also to make sure it doesn't introduce unforeseen bugs at runtime.

The solution is to use feature flags. Feature flags boil down to:

```
if current_user.can_use_feature?("new-feature")
  render_new_feature_widget
end
```

So you don't even load the related code unless the user is a developer working on it, or a small group of beta testers. No matter how unfinished the code is, nobody will be affected. So you can work on it in short iterations and make sure each iteration is well integrated with the system as a whole. Such integrations are much easier to deal with than a big-bang merge.

3.3.6 Use CI to Maintain Your Code

If you're used to working on monolithic applications, building microservices leads to an unfamiliar situation. Services often reach a stage of being done, as in no further work is necessary for the time being.

No one may touch the service's repository for months. And then, one day, there's an urgent need to deploy a change. The CI build unexpectedly fails: there are security vulnerabilities in several dependencies, some of which have introduced breaking changes. What seemed like a minor update becomes a high-risk operation that may drag into days of work.

To prevent this from happening, you can **schedule a daily CI build**. A scheduled build is an excellent way of detecting any issues with dependencies early, regardless of how often your code changes (or doesn't).

You can further support the quality of your code by incorporating in your CI pipeline:

- Code style checkers
- Code smell detectors
- Security scanners

And running them first, before unit tests.

3.4 Continuous Delivery Best Practices

3.4.1 The CI/CD Pipeline is the Only Way to Deploy to Production

A CI/CD pipeline is a codified standard of quality and procedure for making a release. By rejecting any change that breaks any of the rules, the pipeline acts as a gatekeeper of quality. It protects the production environment from unverified code. It pushes the team to work in the spirit of continuous improvement.

It's crucial to maintain the discipline of having every single change go through the pipeline before reaching production. The CI/CD pipeline should be the only way code can reach production.

It can be tempting to break this rule in cases of seemingly exceptional circumstances and revert to manual procedures that circumvent the pipeline. On the contrary, the times of crisis are exactly when the pipeline delivers value by making sure that the system doesn't degrade even further. When timing is critical, the pipeline should roll back to the previous release.

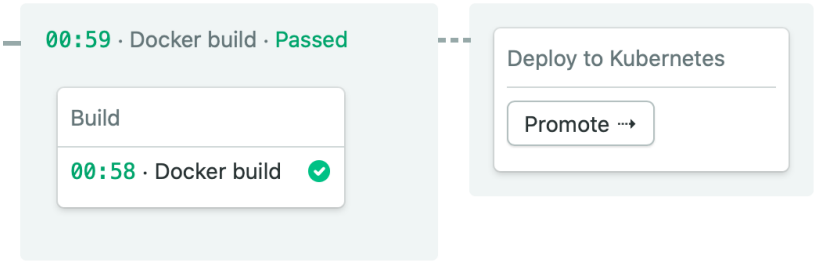
Once it happens that the configuration and history of the CI/CD pipeline diverge from what teams do in reality, it's difficult to re-establish automation and the culture of quality. For this reason, it's important to invest time in making the pipeline fast so that no one feels encouraged to skip it.

3.4.2 Developers Can Deploy to Production-Like Staging Environments at a Push of a Button

An ideal CI/CD pipeline is almost invisible. Developers get feedback from tests without losing focus and deploy with a single command or button press. There's no delay between intent and actualization. Anything that gets in the way of that ideal state is undesirable.

Developers should be the ones who deploy their code. This is in line with the general principle of "You build it, you run it". Delegating that task to anyone else simply makes the process an order of magnitude slower and more complicated.

Developers who build containerized microservices need to have a staging Kubernetes cluster where they can deploy at will. Alternatively, they need a way to deploy a canary build, which we describe later in the book.



The deployment operation needs to be streamlined to a single command that is trivial to run and very unlikely to fail. A more complicated deployment sequence invites human and infrastructure errors that slow down the progress.

3.4.3 Always Use the Same Environment

Before containers, the realistic advice would be to make the pipeline, staging, and production as similar as possible. The goal is to ensure that the automated tests which we run in the CI/CD pipeline accurately reflect how the change would behave in production. The bigger the differences between staging and production, the higher is the chance of introducing bugs.

Today containers guarantee that your code always runs in the same environment. You can run your entire CI/CD pipeline in your custom Docker containers. And you can be sure that the containers that you build early in the pipeline are bit-exact in further pipeline tests, staging, and production.

Other environments are still not the same as production, since reproducing the same infrastructure and load is expensive. However, the differences are manageable, and we get to avoid most of the errors that would have occurred with non-identical environments.

Chapter 1 includes a roadmap for adopting Docker for this purpose. Chapter 2 described some of the advanced deployment strategies that you can use with Kubernetes. Strategies like blue-green and canary deployment reduce the risk of bad deployments. Now that we know what a proper CI/CD pipeline should look like, it's time to start implementing it.

4 Implementing a CI/CD Pipeline

Going to a restaurant and looking at the menu with all those delicious dishes is undoubtedly fun. But in the end, we have to pick something and eat it—the whole point of going out is to have a nice meal. So far, this book has been like a menu, showing you all the possibilities and their ingredients. In this chapter, you are ready to order. *Bon appétit*.

Our goal is to get an application running on Kubernetes using CI/CD best practices.

Our process will include the following steps:

- **Build**: Package the application into a Docker image.
- **Run end-to-end tests**: Run end-to-end tests inside the image.
- **Canary deploy**: Deploy the image as a canary to a fraction of the users.
- **Run functional tests**: Verify the canary in production to decide if we should proceed.
- **Deploy**: If the canary passes the test, deploy the image to all users.
- **Rollback**: If it fails, undo all changes, so we can fix a problem and try again later.

4.1 Docker and Kubernetes Commands

In previous chapters, we've learned most of the Docker and Kubernetes commands that we'll need in this chapter. Here are a few that we haven't seen yet.

4.1.1 Docker Commands

A Docker *registry* stores Docker images. Docker CLI provides the following commands for managing images:

- `push` and `pull`: these commands work like in Git. We can use them to transfer images to and from the registry.

- `login`: we need to log in before we can push images. Takes a username, password, and an optional registry URL.
- `build`: creates a custom image from a `Dockerfile`.
- `tag`: renames an image or changes its tag.
- `exec`: starts a process in an already-running container. Compare it with `docker run`, which creates a new container instead.

4.1.2 Kubectl Commands

Kubectl is the primary admin CLI for Kubernetes. We'll use the following commands during deployments:

- `get service`: in chapter 2, we learned about services in Kubernetes; this shows what services are running in a cluster. For instance, we can check the status and external IP of a load balancer.
- `get events`: shows recent cluster events.
- `describe`: shows detailed information about services, deployments, nodes, and pods.
- `logs`: dumps a container's stdout messages.
- `apply`: starts a declarative deployment. Kubernetes compares the current and target states and takes the necessary steps to reconcile them.
- `rollout status`: shows the deployment progress and waits until the deployment finishes.
- `exec`: works like `docker exec`, executes a command in an already-running pod.
- `delete`: stops and removes pods, deployments, and services.

4.2 Setting Up The Demo Project

It's time to put the book down and get our hands busy for a few minutes. In this section, you'll fork a demo repository and install some tools.

4.2.1 Install Prerequisites

You'll need to have the following tools installed on your computer:

- **git** (*https://git-scm.com*) to manage the code.
- **docker** (*https://www.docker.com*) to run containers.
- **kubectl** (*https://kubernetes.io/docs/tasks/tools/install-kubectl/*) to control the Kubernetes cluster.
- **curl** (*https://curl.haxx.se*) to test the application.

4.2.2 Download The Git Repository

We have prepared a demo project on GitHub with everything that you'll need to set up a CI/CD pipeline:

- Visit *https://github.com/semaphoreci-demos/semaphore-demo-cicd-kubernetes*
- Click on the *Fork* button.
- Click on the *Clone or download* button and copy the URL.
- Clone the Git repository to your computer: `git clone YOUR_REPOSITORY_URL`.

The repository contains a microservice called "addressbook" that exposes a few API endpoints. It runs on Node.js and PostgreSQL.

You will see the following directories and files:

- `.semaphore`: a directory with the CI/CD pipeline.
- `docker-compose.yml`: Docker Compose file for the development environment.
- `Dockerfile`: build file for Docker.
- `manifests`: Kubernetes manifests.
- `package.json`: the Node.js project file.
- `src`: the microservice code and tests.

4.2.3 Running The Microservice Locally

Use `docker-compose` to start a development environment:

```
$ docker-compose up --build
```

Docker Compose builds and runs the container image as required. It also downloads and starts a PostgreSQL database for you.

The included `Dockerfile` builds a container image from an official Node.js image:

```
FROM node:12.16.1-alpine3.10

ENV APP_USER node
ENV APP_HOME /app

RUN mkdir -p $APP_HOME
RUN chown -R $APP_USER $APP_HOME

USER $APP_USER
WORKDIR $APP_HOME

COPY package*.json .jshintrc $APP_HOME/
RUN npm install

COPY src $APP_HOME/src/

EXPOSE 3000
CMD ["node", "src/app.js"]
```

Based on this configuration, Docker performs the following steps:

- Pull the Node.js image.
- Copy the application files.
- Run `npm` inside the container to install the libraries.
- Set the starting command to serve on port 3000.

To verify that the microservice is running correctly, run the following command to create a new record:

```
$ curl -w "\n" -X PUT \
  -d "firstName=al&lastName=pacino" \
  localhost:3000/person

{
  "id":1,
  "firstName":"al",
  "lastName":"pacino",
  "updatedAt":"2020-03-27T10:59:09.987Z",
  "createdAt":"2020-03-27T10:59:09.987Z"
}
```

To list all records:

```
$ curl -w "\n" localhost:3000/all

[
 {
  "id":1,
  "firstName":"al",
  "lastName":"pacino",
  "createdAt":"2020-03-27T10:59:09.987Z",
  "updatedAt":"2020-03-27T10:59:09.987Z"
 }
]
```

4.2.4 Reviewing Kubernetes Manifests

In chapter 3, we learned that Kubernetes is a declarative system: instead of telling it what to do, we state what we want and trust it knows how to get there.

The `manifests` directory contains all the Kubernetes manifest files.

`service.yml` describes the LoadBalancer service. Forwards traffic from port 80 (HTTP) to port 3000.

```yaml
# service.yml
apiVersion: v1
kind: Service
metadata:
  name: addressbook-lb
spec:
  selector:
    app: addressbook
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 3000
```

`deployment.yml` describes deployment. The directory also contains some AWS-specific manifests.

```yaml
# deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: $deployment
spec:
  replicas: $replicas
```

```
  selector:
    matchLabels:
      app: addressbook
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: addressbook
        deployment: $deployment
    spec:
      containers:
        - name: addressbook
          image: $img
          readinessProbe:
            httpGet:
              path: /ready
              port: 3000
          env:
            - name: NODE_ENV
              value: "production"
            - name: PORT
              value: "$PORT"
            - name: DB_SCHEMA
              value: "$DB_SCHEMA"
            - name: DB_USER
              value: "$DB_USER"
            - name: DB_PASSWORD
              value: "$DB_PASSWORD"
            - name: DB_HOST
              value: "$DB_HOST"
            - name: DB_PORT
              value: "$DB_PORT"
            - name: DB_SSL
              value: "$DB_SSL"
```

The deployment manifest combines several Kubernetes concepts we've discussed in chapter 3:

1. A deployment called "addressbook" with rolling updates.
2. Labels for the pods manage traffic and identify release channels.
3. Environment variables for the containers in the pod.
4. A readiness probe to detect when the pod is ready to accept connections.

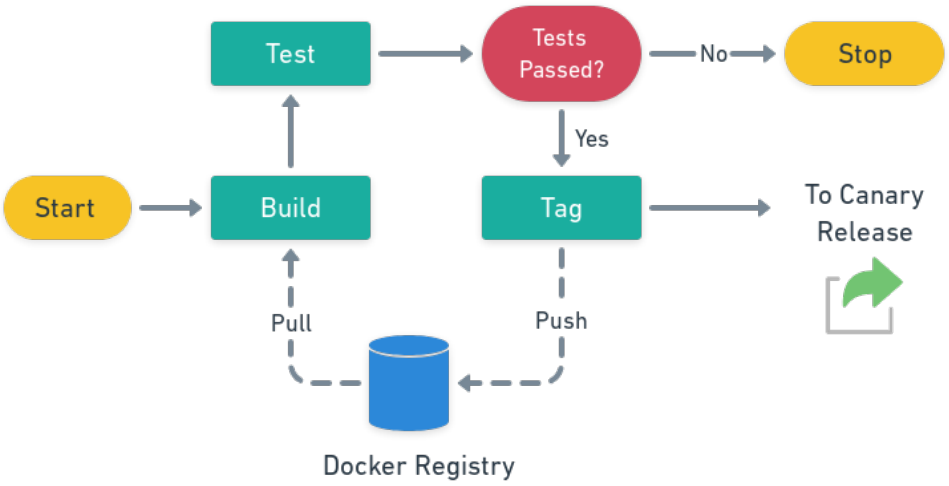
Note that we're using dollar ($) variables in the file. This gives us some flexibility to reuse the same manifest for deploying to multiple environments.

4.3 Overview of the CI/CD Workflow

A good CI/CD workflow takes planning as there are many moving parts: building, testing, and safely deploying code.

4.3.1 CI Pipeline: Building a Docker Image and Running Tests

Our CI/CD workflow begins with the mandatory continuous integration pipeline:



The CI pipeline performs the following steps:

- **Git checkout**: Get the latest source code.
- **Docker pull**: Get the latest available application image, if it exists, from the CI Docker registry. This optional step decreases the build time in the following step.
- **Docker build**: Create a Docker image.
- **Test**: Start the container and run tests inside.
- **Docker push**: If all tests pass, push the accepted image to the production registry.

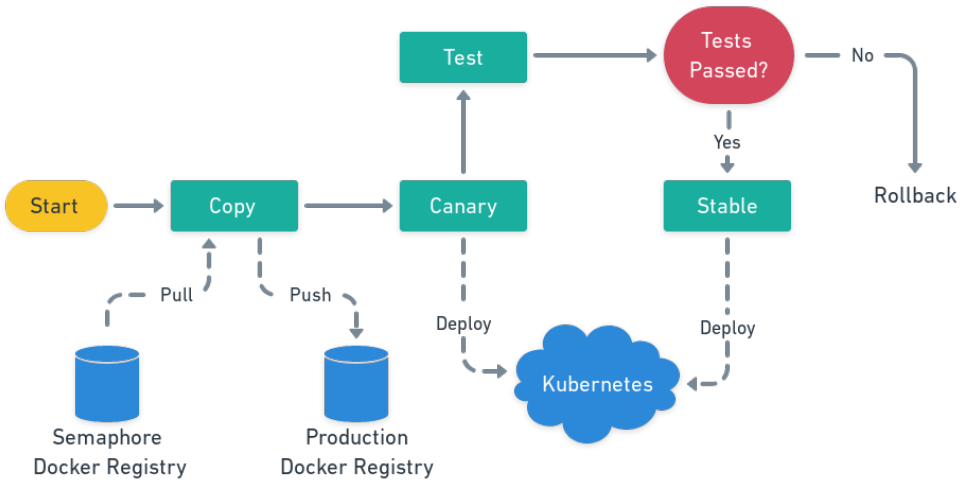
In this process, we'll use Semaphore's built-in Docker registry. This is faster and cheaper than using a registry from a cloud vendor to work with containers in the CI/CD context.

4.3.2 CD Pipelines: Canary and Stable Deployments

In chapter 2, we learned about canaries and rolling deployments. In chapter 3, we have talked about Continuous Delivery and Continuous Deployment. Our CI/CD workflow combines these two practices.

A canary deployment is a limited release of a new version. We'll call it: the *canary release*, and the previous version still used by most users is the *stable release*.

We can do a canary deployment by connecting the canary pods to the same load balancer as the rest of the pods. As a result, a set fraction of user traffic goes to the canary. For example, if we have nine stable pods and one canary pod, 10% of the users would get the canary release.

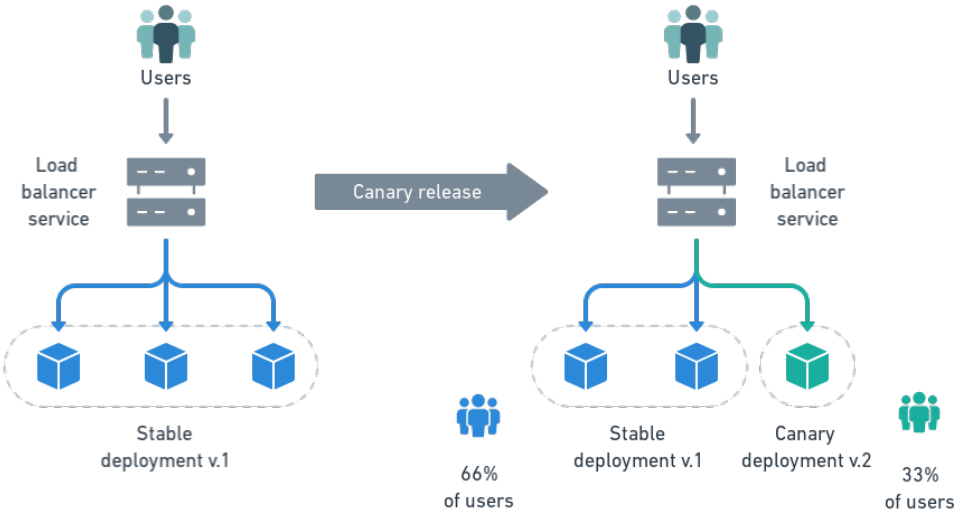


The canary release performs the following steps:

- **Copy** the image from the Semaphore registry to the production registry.
- **Canary deploy** a canary pod.
- **Test** the canary pod to ensure it's working by running automated functional tests. We may optionally also perform manual QA.
- **Stable release**: if test passes, update the rest of the pods.

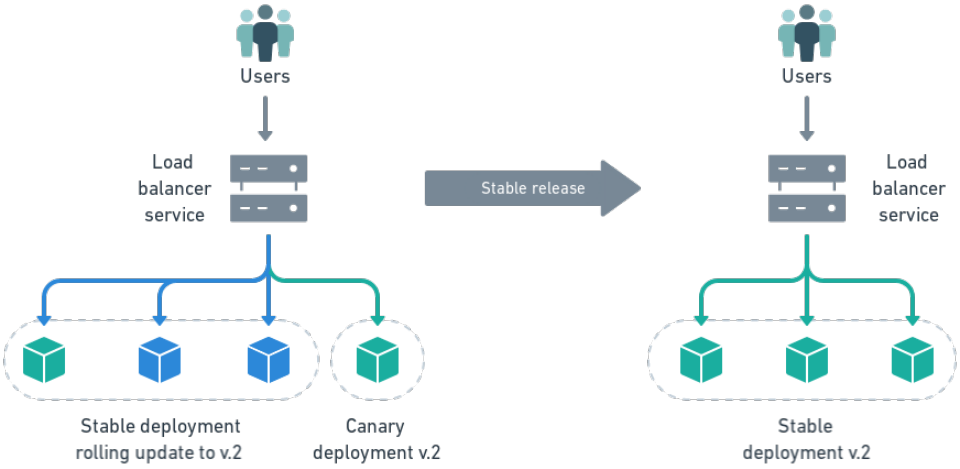
Let's take a closer look at how the stable release works.

Imagine that this is your initial state: you have three pods running version **v1**.



When you deploy **v2** as a canary, you scale down the number of **v1** pods to 2, to keep the total amount of pods to 3.

Then, you can start a rolling update to version **v2** on the stable deployment. One at a time, all its pods are updated and restarted until they are all running on **v2**, and you can get rid of the canary.



4.4 Implementing a CI/CD Pipeline With Semaphore

In this section, we'll learn about Semaphore and how to use it to build cloud-based CI/CD pipelines.

4.4.1 Introduction to Semaphore

For a long time, developers looking for a CI/CD tool had to choose between power and ease of use.

On the one hand, there was Jenkins, which can do just about anything but is challenging to use and requires dedicated ops teams to configure, maintain and scale.

On the other hand, several hosted services let developers push their code and not worry about the rest of the process. However, these services are usually limited to running simple build and test steps. They would often fall short and need more elaborate continuous delivery workflows, which is often the case with containers.

Semaphore (*https://semaphoreci.com*) started as one of the simple hosted CI services, but eventually evolved to support custom continuous delivery pipelines with containers, while retaining a way of being easy to use by any developer, not just dedicated ops teams. As such, it removes all technical barriers to adopting continuous delivery at scale:

- It's a cloud-based service that scales on-demand. There's no software for you to install and maintain.
- It provides a visual interface to model custom CI/CD workflows quickly.
- It's the fastest CI/CD service due to being based on dedicated hardware instead of common cloud computing services.
- It's free for open source and small private projects.

The key benefit of using Semaphore is increased team productivity. Since there is no need to hire supporting staff or expensive infrastructure, and it runs CI/CD workflows faster than any other solution, companies that adopt Semaphore report a very large, 41x ROI comparing to their previous solution [7].

We'll learn about Semaphore's features as we go hands-on in this chapter.

[7] Whitepaper: The 41:1 ROI of Moving CI/CD to Semaphore (*https://semaphoreci.com/resources/roi*)

4.4.2 Creating a Semaphore Account

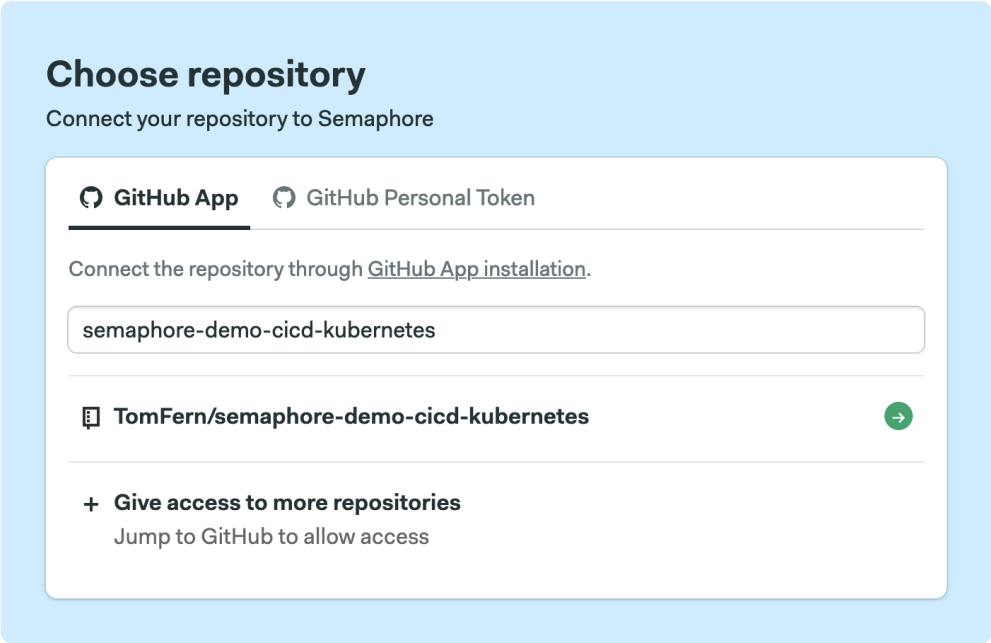
To get started with Semaphore:

- Go to *https://semaphoreci.com* and click to sign up with your GitHub account.
- GitHub will ask you to let Semaphore access your profile information. Allow this so that Semaphore can create an account for you.
- Semaphore will walk you through the process of creating an organization. Since software development is a team sport, all Semaphore projects belong to an organization. Your organization will have its own domain, for example, `awesomecode.semaphoreci.com`.
- You will be asked to choose a plan. In this chapter, we'll use the enterprise plan which features a built-in private Docker Registry. If you're on the free or startup plan, you can use a free public registry like Docker Hub instead. The final workflow is the same.
- Finally, you'll be greeted with a quick product tour.

4.4.3 Creating a Semaphore Project For The Demo Repository

We assume you have previously forked the demo project from *https://github.com/semaphoreci-demos/semaphore-demo-cicd-kubernetes* to your GitHub account.

On Semaphore, click on *New Project* at the top of the screen. Then, click on *Choose a repository*. Next, Semaphore will present you a list of repositories to choose from as the source of your project:

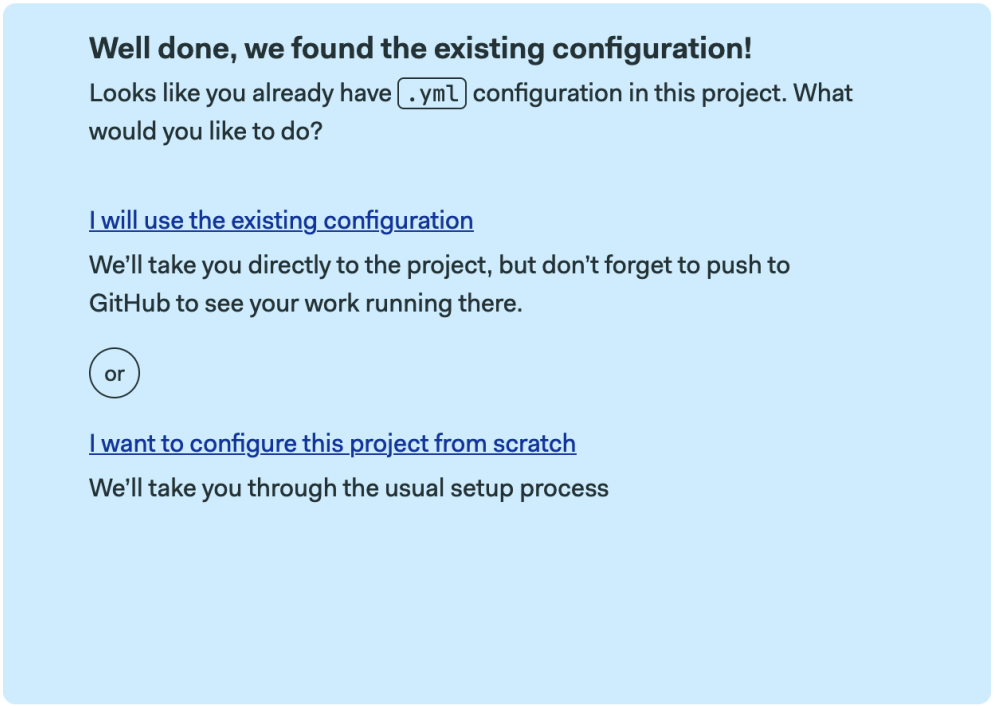


In the search field, start typing `semaphore-demo-cicd-kubernetes` and choose that repository.

Semaphore will quickly initialize the project. Behind the scenes, it will set up everything that's needed to know about every Git push automatically pulling the latest code — without you configuring anything.

The next screen lets you invite collaborators to your project. Semaphore mirrors access permissions of GitHub, so if you add some people to the GitHub repository later, you can "sync" them inside project settings on Semaphore.

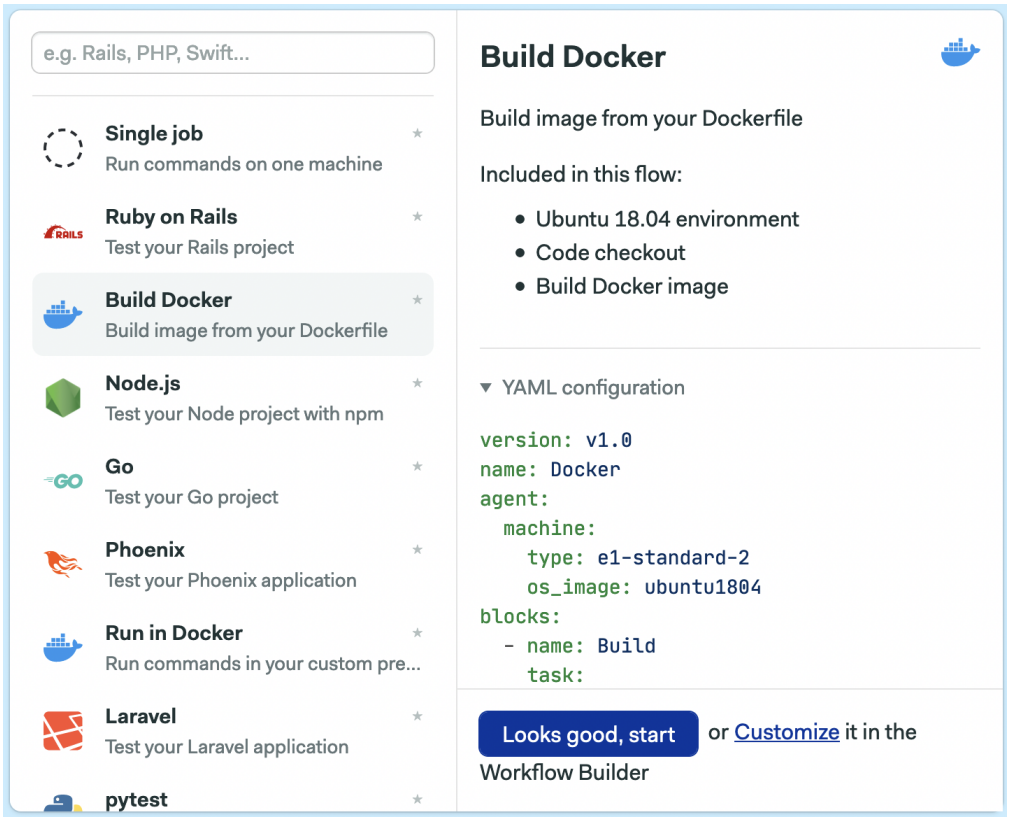
Click on *Continue to Workflow Setup*. Semaphore will ask you if you want to use the existing pipelines or create one from scratch. At this point, you can choose to use the current configuration to get directly to the final workflow. In this chapter, however, we want to learn how to create the pipelines so we'll make a fresh start.



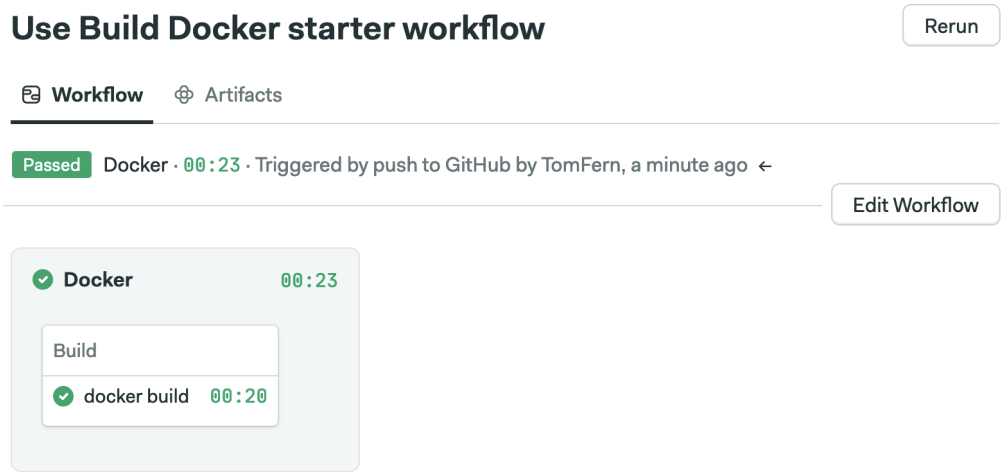
Click on the option to configure the project from scratch.

4.4.4 The Semaphore Workflow Builder

To make the process of creating projects easier, Semaphore provides starter workflows for popular frameworks and languages. Choose the "Build Docker" workflow and click on *Run this workflow*.



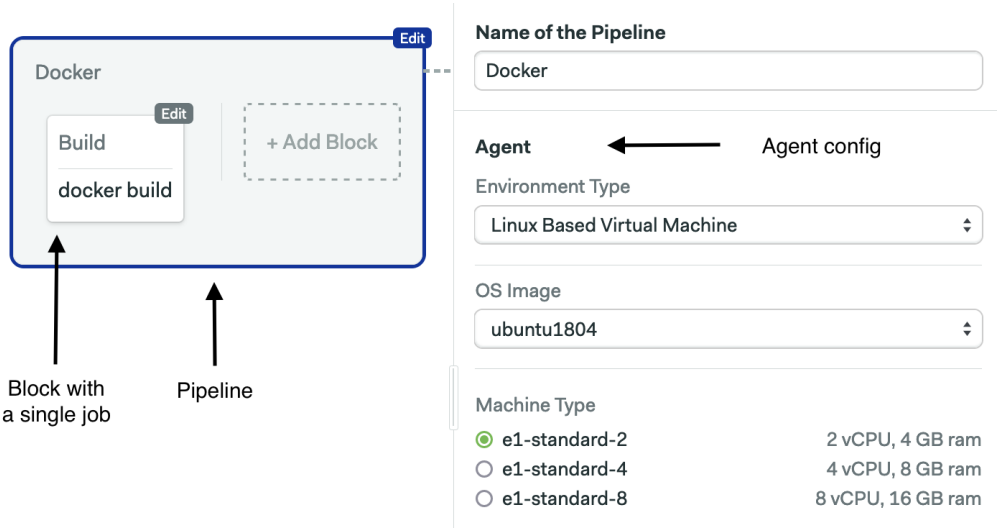
Semaphore will immediately start the workflow. Wait a few seconds, and your first Docker image is ready. Congratulations!



Since we haven't told Semaphore where to store the image yet, it's lost as soon as the job ends. We'll correct that next.

See the *Edit Workflow* button on the top right corner? Click it to open the Workflow Builder.

Now it's a good moment to learn the basic concepts of Semaphore by exploring the Workflow Builder.



Pipelines

Pipelines are represented in Workflow Builder as big gray boxes. Pipelines organize the workflow in blocks that are executed from left to right. Each pipeline usually has a specific objective such as test, build, or deploy. Pipelines can be chained together to make complex workflows.

Agent

The agent is the combination of hardware and software that powers the pipeline. The *machine type* determines the amount of CPUs and memory allocated to the virtual machine[8]. The operating system is controlled by the *Environment Type* and *OS Image* settings.

The default machine is called `e1-standard-2` and has 2 CPUs, 4 GB RAM, and runs a custom Ubuntu 18.04 image.

Jobs and Blocks

Blocks and jobs define what to do at each step. Jobs define the commands that do the work. Blocks contain jobs with a common objective and shared settings.

[8]To see all the available machines, go to https://docs.semaphoreci.com/ci-cd-environment/machine-types

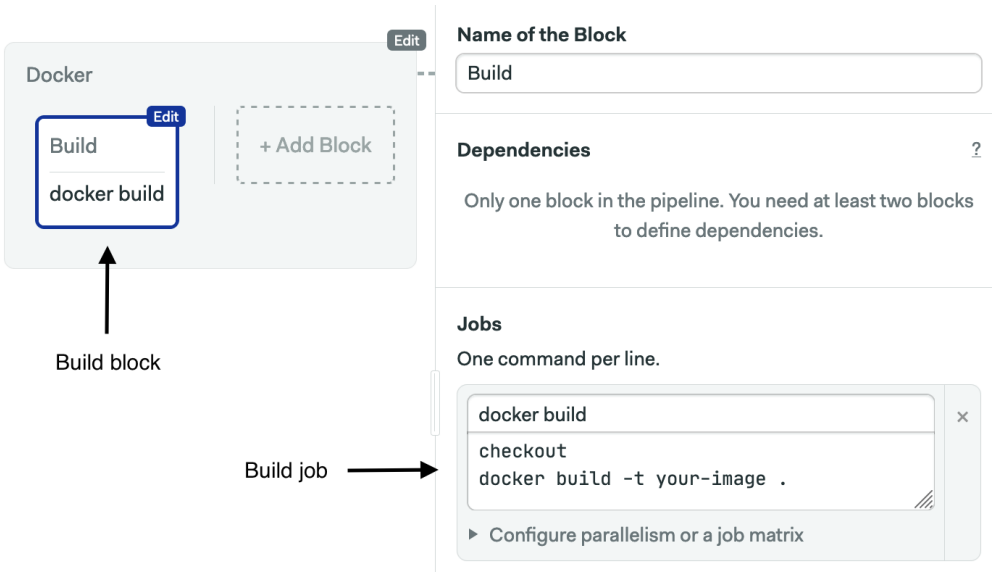
Jobs inherit their configuration from their parent block. All the jobs in a block run in parallel, each in its isolated environment. If any of the jobs fails, the pipeline stops with an error.

Blocks run sequentially. Once all the jobs in the block are complete, the next one starts.

4.4.5 The Continuous Integration Pipeline

We talked about the benefits of CI/CD in chapter 3. In the previous section, we created our very first pipeline. In this section, we'll extend it with tests and a place to store the images.

At this point, you should be seeing the Workflow Builder with the Docker Build starter workflow. Click on the *Build* block so we can see how it works.



Each line on the job is a command to execute. The first command in the job is `checkout`, which is a built-in script that clones the repository at the correct revision[9]. The following command, `docker build`, builds the image using our `Dockerfile`.

Note: Long commands have been broken down into two or more lines with backslash (\) to fit on the page. Semaphore expects one command per line, so when typing them, remove the backslashes and newlines.

[9]You can find the complete Semaphore toolbox at https://docs.semaphoreci.com/reference/toolbox-reference

Replace the contents of the job with the following commands:

```
checkout

docker login \
  -u $SEMAPHORE_REGISTRY_USERNAME \
  -p $SEMAPHORE_REGISTRY_PASSWORD \
  $SEMAPHORE_REGISTRY_URL

docker pull \
  $SEMAPHORE_REGISTRY_URL/demo:latest || true

docker build \
  --cache-from $SEMAPHORE_REGISTRY_URL/demo:latest \
  -t $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID .

docker push \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID
```

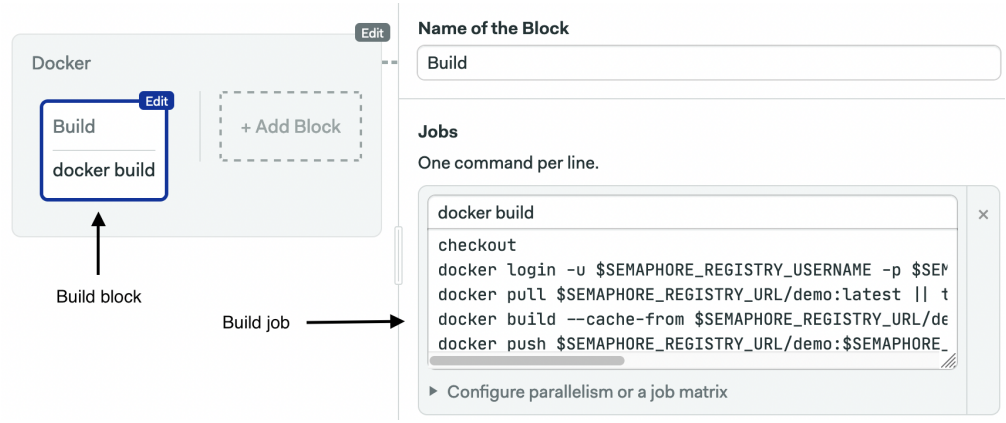
Each command has its purpose:

1. Clones the repository with `checkout`.
2. Logs in the Semaphore private Docker registry[10].
3. Pulls the Docker image tagged as `latest`.
4. Builds a newer version of the image using the latest code.
5. Pushes the new image to the registry.

The discerning reader will note that we introduced special environment variables; these come predefined in every job[11]. The variables starting with `SEMAPHORE_REGISTRY_*` are used to access the private registry. Also, we're using `SEMAPHORE_WORKFLOW_ID`, which is guaranteed to be unique for each run, to tag the image.

[10] Semaphore's built-in Docker registry is available under the enterprise plan. If you're using a free, open-source, or startup plan, use an external service like Docker Hub instead. The pipeline will be slower, but the workflow will be the same.

[11] The full environment reference can be found at https://docs.semaphoreci.com/ci-cd-environment/environment-variables



Now that we have a Docker image that we can test, let's add a second block. Click on the *+Add Block* dotted box.

The Test block will have jobs:

- Static tests.
- Integration tests.
- Functional tests.

The general sequence is the same for all tests:

1. Pull the image from the registry.
2. Start the container.
3. Run the tests.

Blocks can have a *prologue* in which we can place shared initialization commands. Open the prologue section on the right side of the block and type the following commands, which will be executed before each job:

```
docker login \
  -u $SEMAPHORE_REGISTRY_USERNAME \
  -p $SEMAPHORE_REGISTRY_PASSWORD \
  $SEMAPHORE_REGISTRY_URL

docker pull \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID
```

Next, rename the first job as "Unit test" and type the following command, which runs JSHint, a static code analysis tool:

```
docker run -it \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  npm run lint
```

Next, click on the *+Add another job* link below to create a new one called "Functional test". Type these commands:

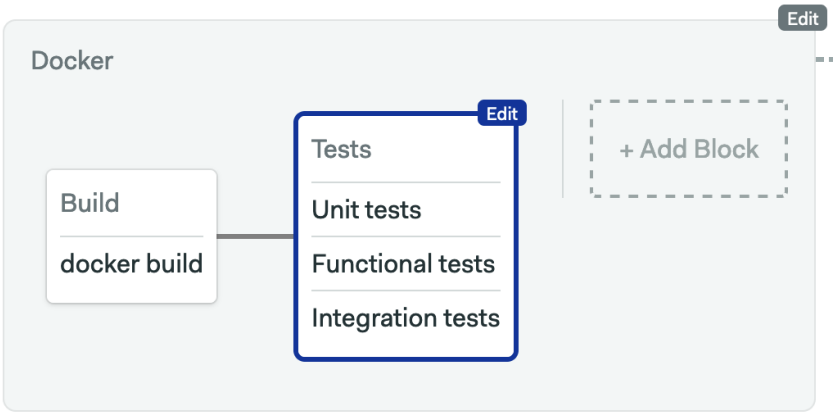
```
sem-service start postgres

docker run --net=host -it \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  npm run ping

docker run --net=host -it \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  npm run migrate
```

This job tests two things: that the container connects to the database (`ping`) and can create the tables (`migrate`). Obviously, we'll need a database for this to work; fortunately, we have `sem-service`, which lets us start database engines like MySQL, Postgres, or MongoDB with a single command[12].

Finally, add a third job called "Integration test" and type these commands:

```
sem-service start postgres

docker run --net=host -it \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  npm run test
```

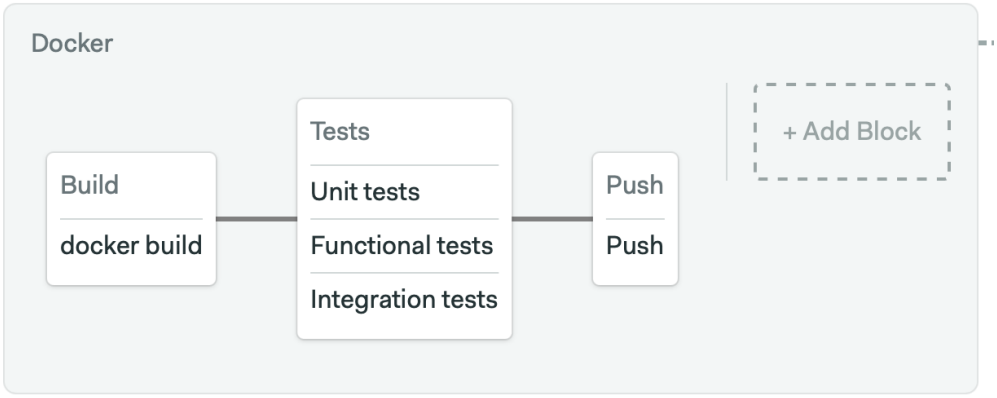
This last test runs the code in `src/database.test.js`, checking if the application can write and delete rows in the database.



[12]For the complete list of services, sem-service can manage check: https://docs.semaphoreci.com/ci-cd-environment/sem-service-managing-databases-and-services-on-linux/

Create the third block in the pipeline and call it "Push". This last job will tag the current Docker image as `latest`. Type these commands in the job:

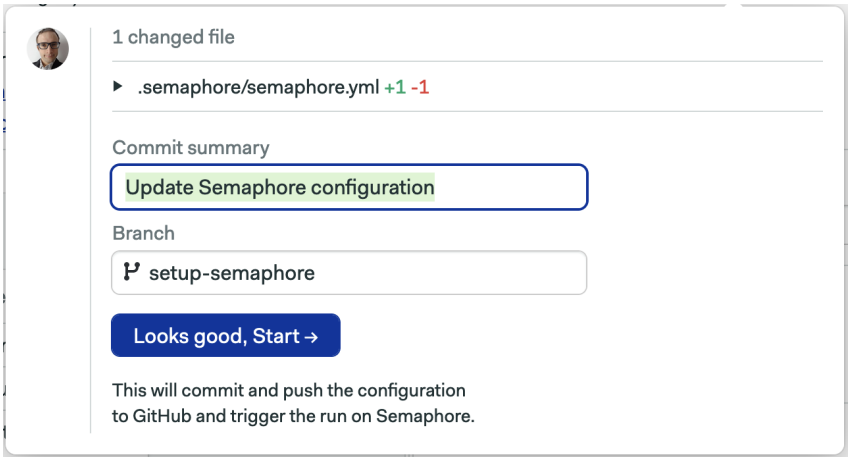
```
docker login \
  -u $SEMAPHORE_REGISTRY_USERNAME \
  -p $SEMAPHORE_REGISTRY_PASSWORD $SEMAPHORE_REGISTRY_URL

docker pull \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID

docker tag \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  $SEMAPHORE_REGISTRY_URL/demo:latest

docker push \
  $SEMAPHORE_REGISTRY_URL/demo:latest
```



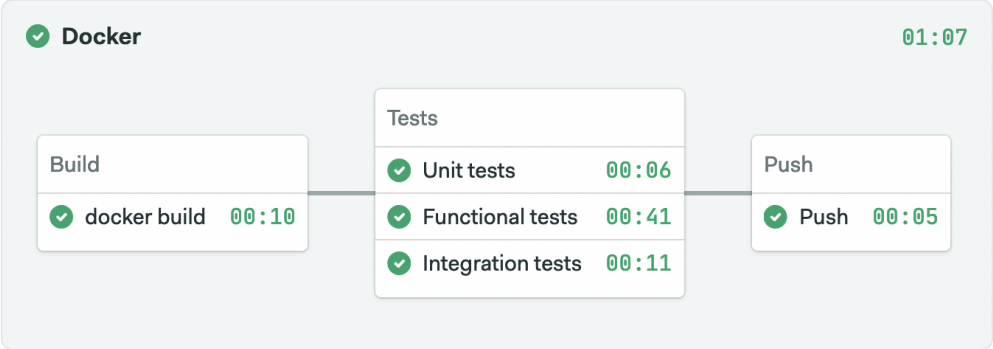
This completes the setup of the CI pipeline.

4.4.6 Your First Build

We've covered many things in a few pages; here, we have the chance to pause for a little bit and try the CI pipeline. Click on the *Run the workflow* button on the top-right corner and click on *Start*.



After a few seconds, the pipeline will start building and testing the container.



4.5 Provisioning Kubernetes

This book will show you how to deploy to Kubernetes hosted on three public cloud providers: Amazon AWS, Google Cloud Platform, and DigitalOcean. With slight modifications, the process will work with any other cloud or Kubernetes instance.

We'll deploy the application in a three-node Kubernetes cluster. You can pick a different size based on your needs, but you'll need at least three nodes to run an effective canary deployment with rolling updates.

4.5.1 DigitalOcean Cluster

DigitalOcean provides everything needed to deploy the application: a managed Kubernetes, a Container Registry, and Postgres databases.

To create the Kubernetes cluster:

- Sign up or log in to your account on digitalocean.com.
- Create a *New Project*.
- Create a *Kubernetes* cluster: select the latest version and choose one of the available regions. Name your cluster "semaphore-demo-cicd-kubernetes".
- While DigitalOcean is working on the cluster, go to the *API* menu and generate a *Personal Access Token* with Read & Write permissions.

Next, create a Container Registry with the following actions:

- Go to *Container Registry*.
- Click *Create*.
- Set the registry name. Names are unique across all DigitalOcean customers.
- Select the *Starter* free plan.

On Semaphore, store the DigitalOcean Access Token as a secret:

1. Log in to your organization on id.semaphoreci.com.
2. In the sidebar on the left-hand side, under *Configuration*, select *Secrets* and click on the *Create New Secret* button.
3. The name of the secret is "do-key".
4. Add the `DO_ACCESS_TOKEN` variable and set its value with your personal token.
5. Click on *Save Secret*.

4.5.2 Google Cloud Cluster

Google Cloud calls its service *Kubernetes Engine*. To create the services:

- Sign up or log in to your Google Cloud account on cloud.google.com.
- Create a *New Project*. In *Project ID* type "semaphore-demo-cicd-kubernetes".
- Go to *Kubernetes Engine* > *Clusters* and enable the service. Create a public **autopilot** cluster in one of the available zones.
- Name your cluster "semaphore-demo-cicd-kubernetes".
- Go to *IAM* > *Service Accounts*.
- Generate an account Basic > Owner role.
- Click on the menu for the new roles, select *Manage Keys* > *Add Keys*.
- Generate and download a **JSON** Access Key file.

On Semaphore, create a secret for your Google Cloud Access Key file:

1. Log in to your organization on id.semaphoreci.com.
2. Open your account menu and click on Settings. Go to *Secrets* > *New Secret*.
3. Name the secret "gcp-key".
4. Add this file: `/home/semaphore/gcp-key.json` and upload the Google Cloud Access JSON from your computer.
5. Click on *Save Secret*.

4.5.3 AWS Cluster

AWS calls its service *Elastic Kubernetes Service* (EKS). The Docker private registry is called *Elastic Container Registry* (ECR).

Creating a cluster on AWS is, unequivocally, a complex affair. So tough that there is a specialized tool for it:

- Sign up or log in to your AWS account at aws.amazon.com.
- Select one of the available regions.
- Find and go to the *ECR* service. Create a new private repository called "semaphore-demo-cicd-kubernetes" and copy its address.
- Install *eksctl* from `eksctl.io` and *awscli* from `aws.amazon.com/cli` in your machine.
- Find the *IAM* console in AWS and create a user with Administrator permissions. Get its *Access Key Id* and *Secret Access Key* values.

Open a terminal and sign in to AWS:

```
$ aws configure

AWS Access Key ID: TYPE YOUR ACCESS KEY ID
AWS Secret Access Key: TYPE YOUR SECRET ACCESS KEY
Default region name: TYPE A REGION
```

To create a three-node cluster of the most inexpensive machine type use:

```
$ eksctl create cluster \
  -t t2.nano -N 3 \
  --region YOUR_REGION \
  --name semaphore-demo-cicd-kubernetes
```

Note: Select the same region for all AWS services.

Once it finishes, eksctl should have created a kubeconfig file at `$HOME/.kube/config`. Check the output from eksctl for more details.

On Semaphore, create a secret to store the AWS Secret Access Key and the kubeconfig file:

1. Log in to your organization on id.semaphoreci.com.
2. In the sidebar on the left-hand side, under *Configuration*, select *Secrets* and click on the *Create New Secret* button.
3. Call the secret "aws-key".
4. Add the following variables:
   - `AWS_ACCESS_KEY_ID` should have your AWS Access Key ID string.
   - `AWS_SECRET_ACCESS_KEY` has the AWS Access Secret Key string.
5. Add the following file:
   - `/home/semaphore/aws-key.yml` and upload the Kubeconfig file created by eksctl earlier.
6. Click on *Save Secret*.

4.6 Provisioning a Database

We'll need a database to store data. For that, we'll use a managed PostgreSQL service.

4.6.1 DigitalOcean Database

- Go to *Databases*.
- Create a PostgreSQL database. Select the same region where the cluster is running.

- Once the database is ready, go to the *Users & Databases* tab and create a database called "demo" and a user named "demouser".
- In the *Overview* tab, take note of the PostgreSQL IP address and port.

4.6.2 Google Cloud Database

- Select *SQL* on the console menu.
- Create a new **PostgreSQL** database instance.
- Select the same region and zone where the Kubernetes cluster is running.
- Open the *Customize your instance* section.
- Enable the *Private IP* network with the default options and an automatically allocated IP range.
- Create the instance.

Once the cloud database is running:

- Open the left-side menu and select *Users*. Create a new built-in user called "demouser".
- Go to the *Databases* and create a new DB called "demo".
- In the *Overview* tab (you can skip the getting started part), take note of the database IP address and port.

4.6.3 AWS Database

- Find the service called *RDS*.
- Create a PostgreSQL database (choose Standard Create) and call it "demo". Type in a secure password for the `postgres` account.
- Select one of the available *templates*. The dev/test option is perfect for demoing the application. Under *Connectivity* select all the VPCs and subnets where the cluster is running (they should have appeared in eksctl's output).
- In Availability Zone, select the same region the Kubernetes cluster is running.
- Under *Connectivity & Security*, take note of the endpoint address and port.

4.6.4 Creating the Database Secret on Semaphore

The database secret is the same for all clouds. Create a secret to store the database credentials:

1. Log in to your organization on id.semaphoreci.com.

2. On the main page, under *Configuration* select *Secrets* and click on the *Create New Secret* button.
3. The secret name is "db-params".
4. Add the following variables:
    - `DB_HOST` with the database hostname or private IP.
    - `DB_PORT` points to the database port (default is 5432).
    - `DB_SCHEMA` for AWS should be called "postgres". For the other clouds, its value should be "demo".
    - `DB_USER` for the database user.
    - `DB_PASSWORD` with the password.
    - `DB_SSL` should be "true" for DigitalOcean. It can be left empty for the rest.
5. Click on *Save Secret*.

4.7 The Canary Pipeline

Now that we have our cloud services, we're ready to prepare the canary deployment pipeline.

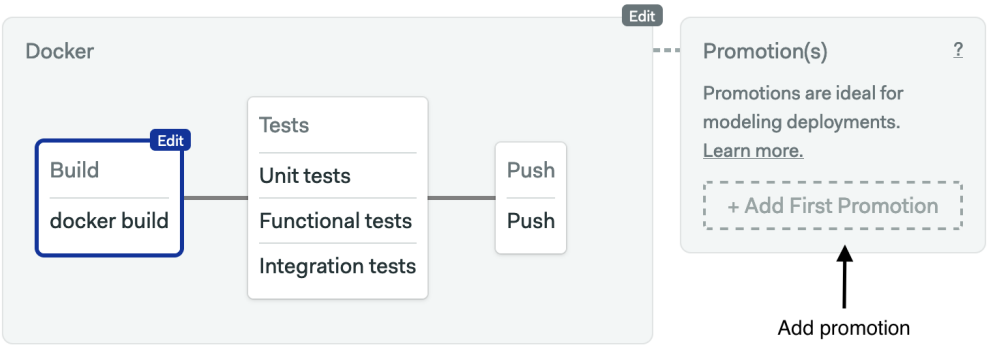
Our project on GitHub includes three ready-to-use reference pipelines for deployment. They should work out-of-the-box in combination with the secrets as described earlier. For further details, check the `.semaphore` folder in the project.

In this section, you'll learn how to create deployment pipelines on Semaphore from scratch. We'll use DigitalOcean as an example, but the process is essentially the same for other clouds.

4.7.1 Creating a Promotion and Deployment Pipeline

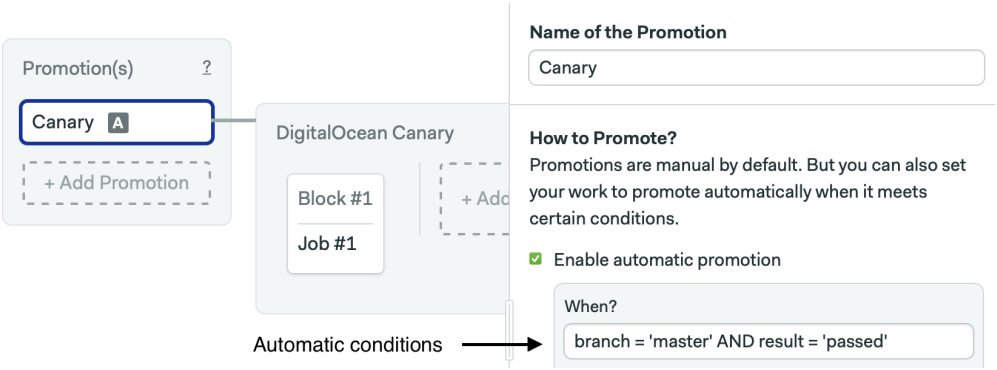
On Semaphore, open the Workflow Builder to create a new pipeline.

Create a new promotion using the *+Add First Promotion* button. Promotions connect pipelines together to create complex workflows. Let's call the new pipeline "Canary".



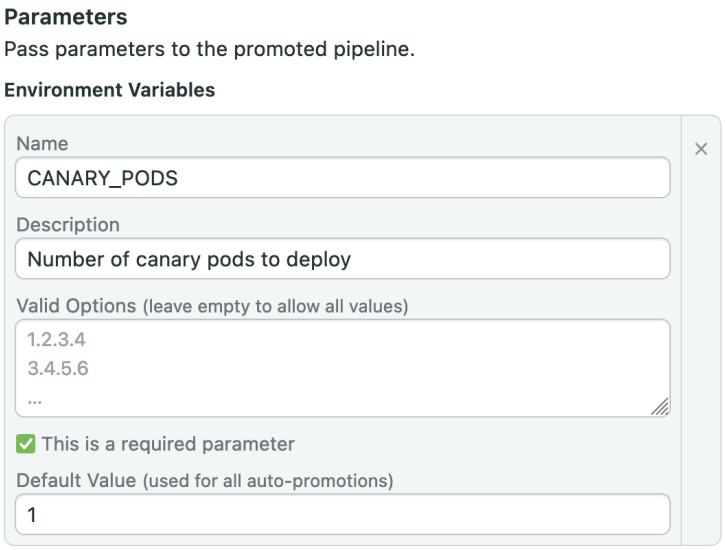
Check the *Enable automatic promotion* box. Now we can define the following auto-starting conditions for the new pipeline:

```
result = 'passed' and (branch = 'master' or tag =~ '^hotfix*')
```

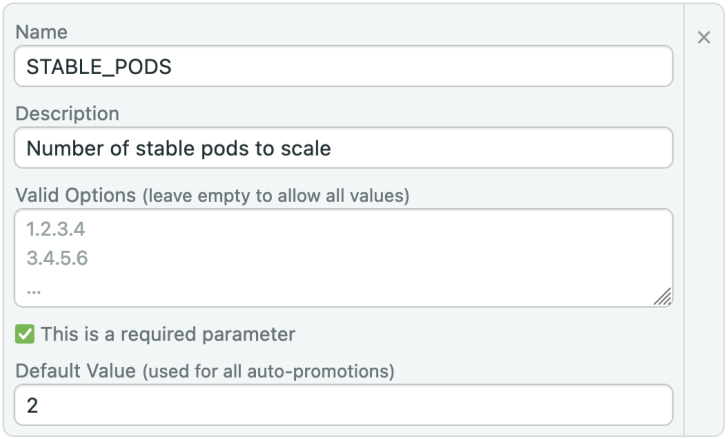


Below the promotion options, click on *+ Add Environment Variables* to create a parameter for a pipeline. Parametrization lets us set runtime values and reuse a pipeline for similar tasks.

The parameter we're going to add will let us specify the number of Canary pods to deploy. Set the variable name to `CANARY_PODS`, ensure that "This is a required parameter" is checked, and type "1" in the default value.



Create a second parameter called `STABLE_PODS`. Set the default value to "2".



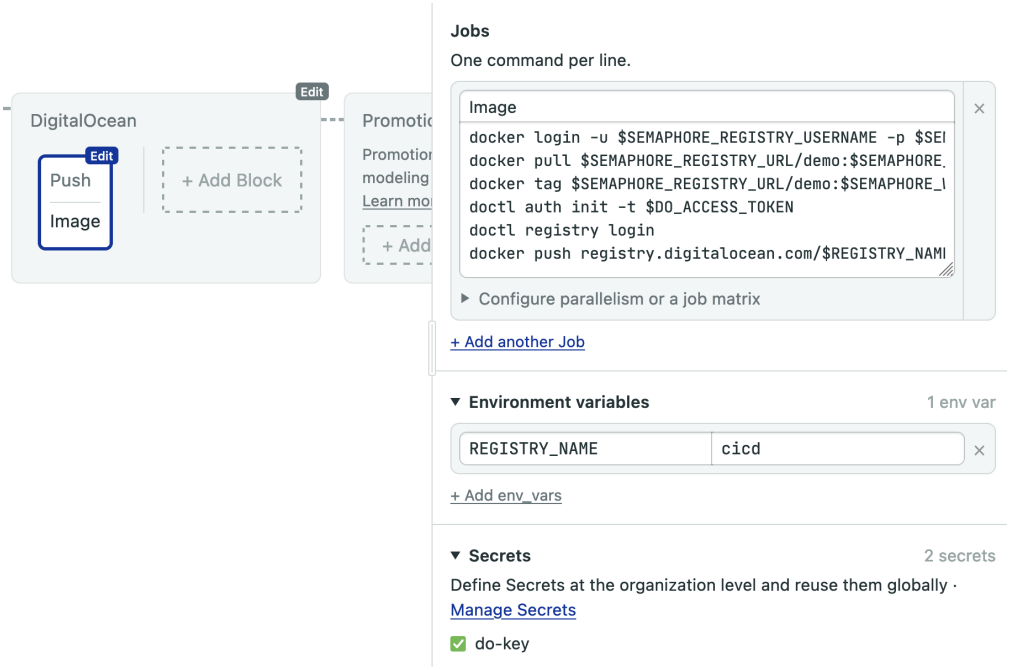
In the new pipeline, click on the first block. Let's call it "Push". The push block takes the Docker image that we built earlier and uploads it to the private Container Registry. The secrets and the login command will vary depending on the cloud of choice.

Open the *Secrets* section and check the `do-key` secret.

Type the following commands in the job:

```
docker login \
  -u $SEMAPHORE_REGISTRY_USERNAME \
  -p $SEMAPHORE_REGISTRY_PASSWORD \
```

```
  $SEMAPHORE_REGISTRY_URL

docker pull \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID

docker tag \
  $SEMAPHORE_REGISTRY_URL/demo:$SEMAPHORE_WORKFLOW_ID \
  registry.digitalocean.com/$REGISTRY_NAME/demo:$SEMAPHORE_WORKFLOW_ID

doctl auth init -t $DO_ACCESS_TOKEN
doctl registry login

docker push \
  registry.digitalocean.com/$REGISTRY_NAME/demo:$SEMAPHORE_WORKFLOW_ID
```



Create a new block called "Deploy" and enable secrets:

- `db-params` to use the cloud database;
- `do-key` is the cloud-specific access token.

Open the *Environment Variables* section:

- Create a variable called `CLUSTER_NAME` with the DigitalOcean cluster name (`semaphore-demo-cicd-kubernetes`)
- Create a variable called `REGISTRY_NAME` with the name of the DigitalOcean container registry name.

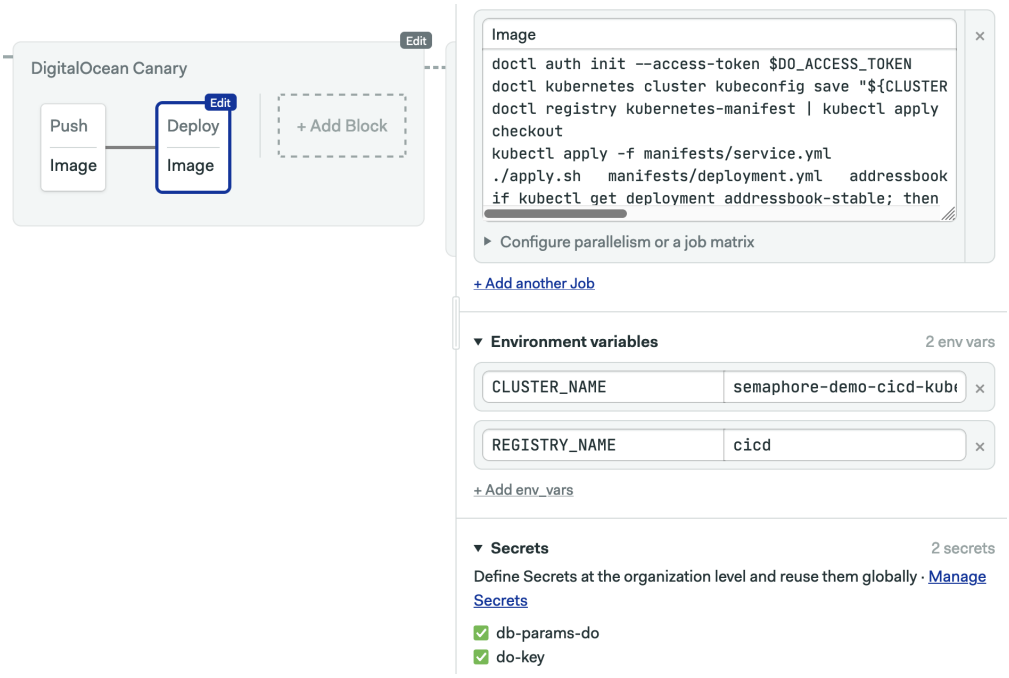
To connect with the DigitalOcean cluster, we can use the preinstalled official `doctl` tool.

Add the following commands to the *job*:

```
doctl auth init --access-token $DO_ACCESS_TOKEN
doctl kubernetes cluster kubeconfig save "${CLUSTER_NAME}"
doctl registry kubernetes-manifest | kubectl apply -f -
checkout
kubectl apply -f manifests/service.yml

./apply.sh \
  manifests/deployment.yml \
  addressbook-canary $CANARY_PODS \
  registry.digitalocean.com/$REGISTRY_NAME/demo:$SEMAPHORE_WORKFLOW_ID

if kubectl get deployment addressbook-stable; then \
  kubectl scale --replicas=$STABLE_PODS deployment/addressbook-stable; \
fi
```

This is the canary job sequence:

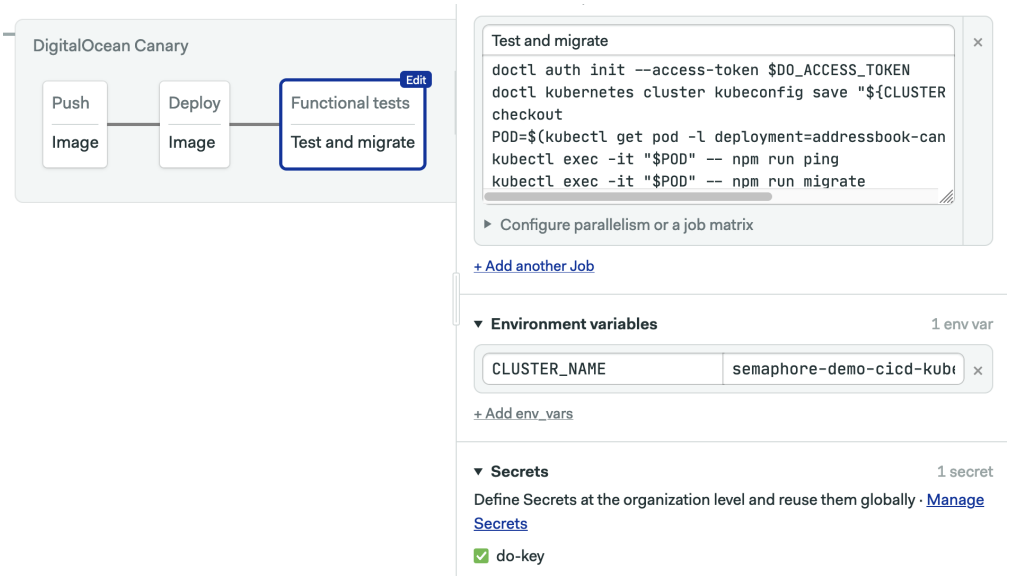
- Initialize the cluster config.
- Generate the Container Service credentials and import them into the cluster.
- Clone the GitHub repository with `checkout`.
- Create a load balancer service with `kubectl apply`.
- Execute `apply.sh`, which creates the canary deployment. The number of pods in the deployment is held `$CANARY_PODS`.
- Reduce the size of the stable deployment with `kubectl scale` to `$STABLE_PODS`.



Create a third block called "Functional Test" and enable the `do-key` secret. Repeat the environment variables. This is the last block in the pipeline, and it runs some automated tests on the canary. By combining `kubectl get pod` and `kubectl exec`, we can run commands inside the pod.

Type the following commands in the job:

```
doctl auth init --access-token $DO_ACCESS_TOKEN
doctl kubernetes cluster kubeconfig save "${CLUSTER_NAME}"
doctl registry kubernetes-manifest | kubectl apply -f -
checkout
POD=$(kubectl get pod -l deployment=addressbook-canary -o name | head -n 1)
kubectl exec -it "$POD" -- npm run ping
kubectl exec -it "$POD" -- npm run migrate
```



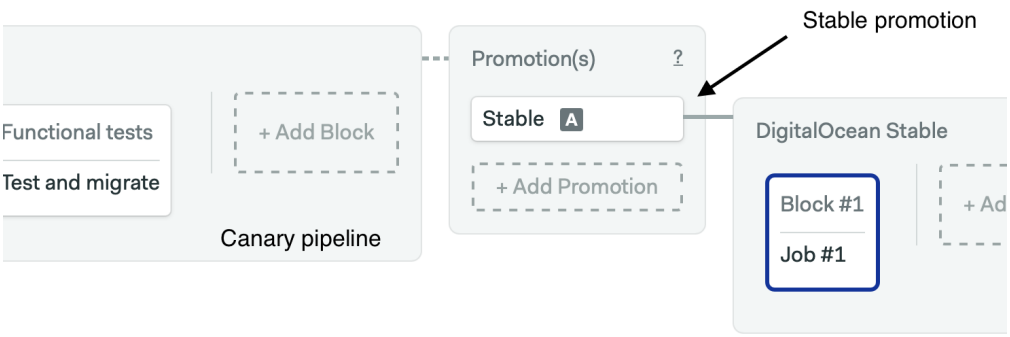
4.8 Your First Release

So far, so good. Let's see where we are: we built the Docker image, and, after testing it, we've set up the one-pod canary deployment pipeline. In this section, we'll extend the workflow with a stable deployment pipeline.

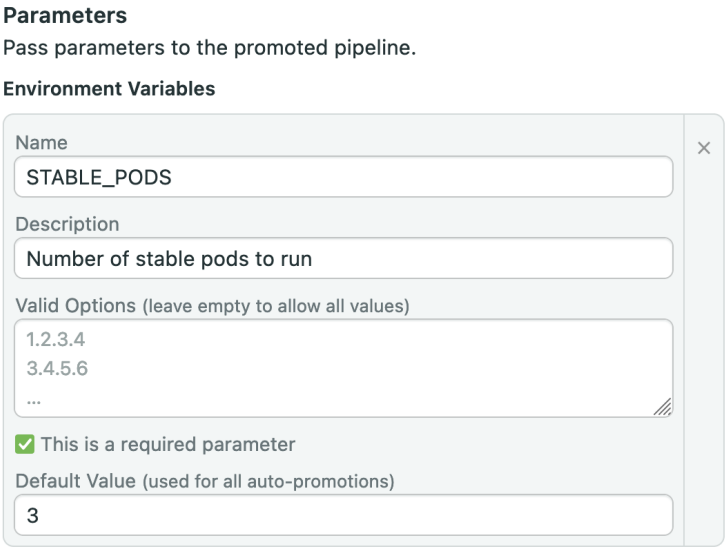
4.8.1 The Stable Deployment Pipeline

The stable pipeline completes the deployment cycle. This pipeline doesn't introduce anything new; again, we use `apply.sh` script to start a rolling update and `kubectl delete` to clean the canary deployment.

Create a new pipeline (using the *Add promotion* button) branching out from the canary and name it "Deploy Stable (DigitalOcean)".



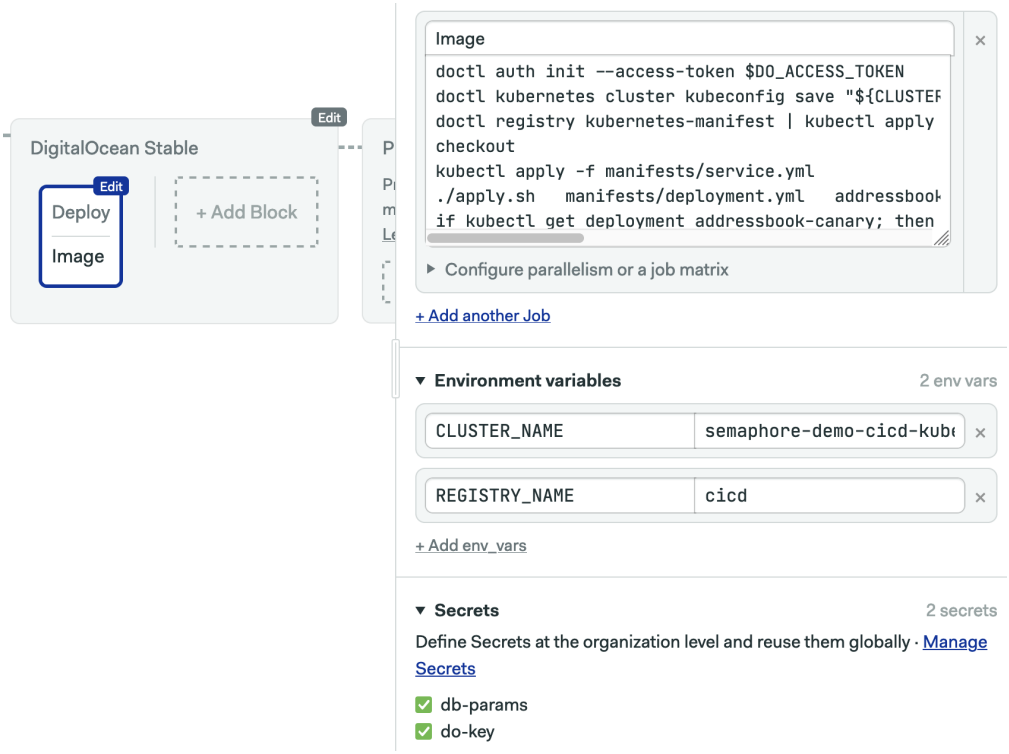
Add a parameter called `$STABLE_PODS` with default value "3".



Create the "Deploy to Kubernetes" block with the `do-key` and `db-params` secrets. Also, create the `CLUSTER_NAME` and `REGISTRY_NAME` variables as we did in the previous step.

In the job command box, type the following lines to make the rolling deployment and delete the canary pods:

```
doctl auth init --access-token $DO_ACCESS_TOKEN
doctl kubernetes cluster kubeconfig save "${CLUSTER_NAME}"
doctl registry kubernetes-manifest | kubectl apply -f -
checkout
kubectl apply -f manifests/service.yml

./apply.sh \
  manifests/deployment.yml \
  addressbook-stable $STABLE_PODS \
  registry.digitalocean.com/$REGISTRY_NAME/demo:$SEMAPHORE_WORKFLOW_ID

if kubectl get deployment addressbook-canary; then \
  kubectl delete deployment/addressbook-canary; \
fi
```



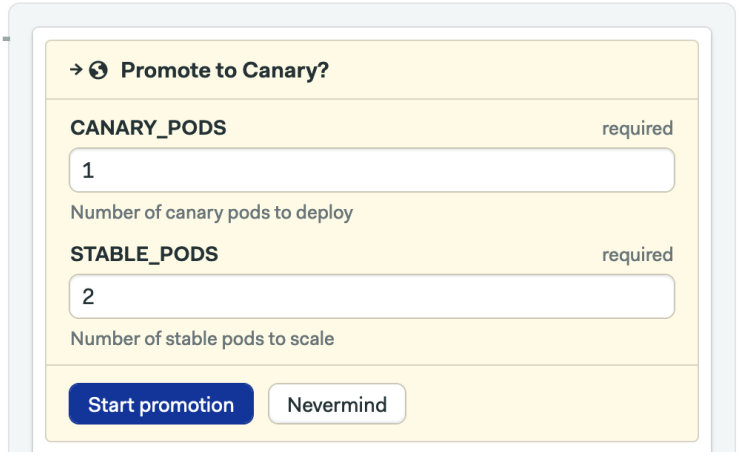
Good! We're done with the release pipeline.

4.8.2 Releasing the Canary

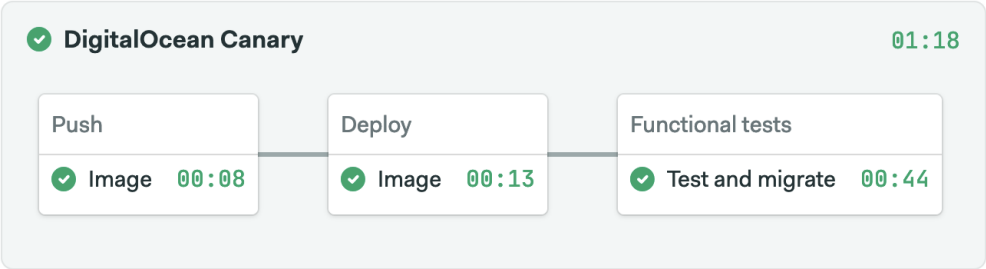
Here is the moment of truth. Will the canary work? Click on *Run the workflow* and then *Start*.

Wait until the CI pipeline is done an click on *Promote* to start the canary pipeline[13]. As you can see on the screenshot below, manually starting a promotion lets you customize the parameters.

[13]You might be wondering why the automatic promotion hasn't kicked in for the canary pipeline. The reason is that we set it to trigger only for the master branch, and the Workflow Builder by default saves all its changes on a separate branch called `setup-semaphore`.



Press *Start Promotion* to run the canary pipeline.



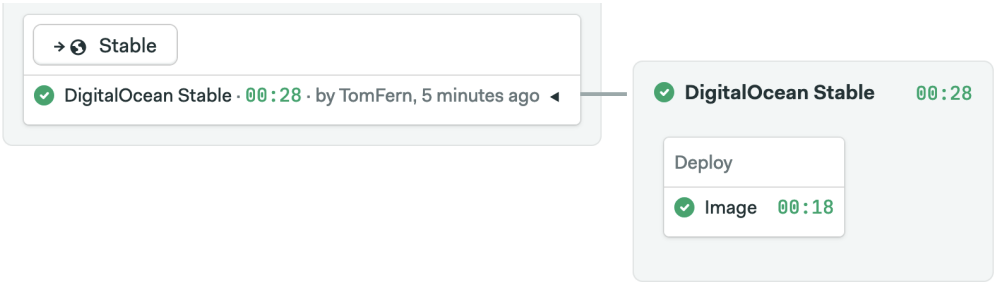
Once it completes, we can check how the canary is doing.

```
$ kubectl get deployment

NAME                READY UP-TO-DATE AVAILABLE AGE
addressbook-canary  1/1   1          1         8m40s
```

4.8.3 Releasing the Stable

In tandem with the canary deployment, we should have a dashboard to monitor errors, user reports, and performance metrics to compare against the baseline. After some predetermined amount of time, we would reach a go vs. no-go decision. Is the canary version good enough to be promoted to stable? If so, the deployment continues. If not, after collecting the necessary error reports and stack traces, we roll back and regroup.

Let's say we decide to go ahead. So go on and hit the *Promote* button. You can tweak the number of final pods to deploy. The stable pipeline should be done in a few seconds.



If you're fast enough, you can see both the existing canary and a new "addressbook-stable" deployment while the block runs.

```
$ kubectl get deployment

NAME                READY UP-TO-DATE AVAILABLE AGE
addressbook-canary  1/1   1          1         110s
addressbook-stable  0/3   3          0         1s
```

One at a time, the numbers of replicas should increase until reaching the target of three:

```
$ kubectl get deployment

NAME                READY UP-TO-DATE AVAILABLE AGE
addressbook-canary  1/1   1          1         114s
addressbook-stable  2/3   3          2         5s
```

With that completed, the canary is no longer needed, so it goes away:

```
$ kubectl get deployment

NAME                READY UP-TO-DATE AVAILABLE AGE
addressbook-stable  3/3   3          3         12s
```

Check the service status to see the external IP:

```
$ kubectl get service

NAME            TYPE         EXTERNAL-IP    PORT(S)
addressbook-lb  LoadBalancer 35.225.210.248 80:30479/TCP
kubernetes      ClusterIP    <none>         443/TCP
```

We can use curl to test the API endpoint directly. For example, to create a person in the addressbook:

```
$ curl -w "\n" -X PUT \
  -d "firstName=Sammy&lastName=David Jr" \
  34.68.150.168/person

{
    "id": 1,
    "firstName": "Sammy",
    "lastName": "David Jr",
    "updatedAt": "2019-11-10T16:48:15.900Z",
    "createdAt": "2019-11-10T16:48:15.900Z"
}
```

To retrieve all persons, try:

```
$ curl -w "\n" 34.68.150.168/all

[
    {
        "id": 1,
        "firstName": "Sammy",
        "lastName": "David Jr",
        "updatedAt": "2019-11-10T16:48:15.900Z",
        "createdAt": "2019-11-10T16:48:15.900Z"
    }
]
```

The deployment was a success; that was no small feat. Congratulations!

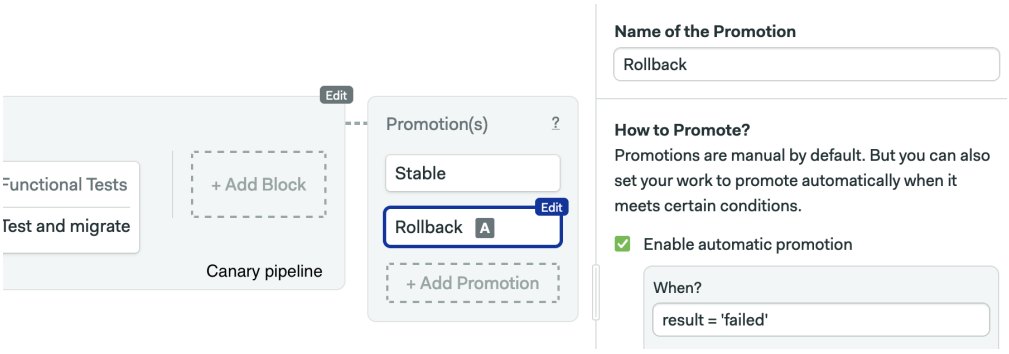
4.8.4 The Rollback Pipeline

Fortunately, Kubernetes and CI/CD make an exceptional team when it comes to recovering from errors. Let's say that we don't like how the canary performs or, even worse, the functional tests at the end of the canary deployment pipeline fail. In that case, wouldn't it be great to have the system go back to the previous state automatically? What about being able to undo the change with a click of a button? This is exactly what we will configure at this point, a rollback pipeline [14].

Open the Workflow Builder once more and go to the end of the canary pipeline. Create a new promotion branching out of it, check the *Enable automatic promotion* box, and set this condition:

[14] This isn't technically true for applications that use databases. Changes to the database are not automatically rolled back. We should use database backups and migration scripts to manage upgrades.

```
result = 'failed'
```

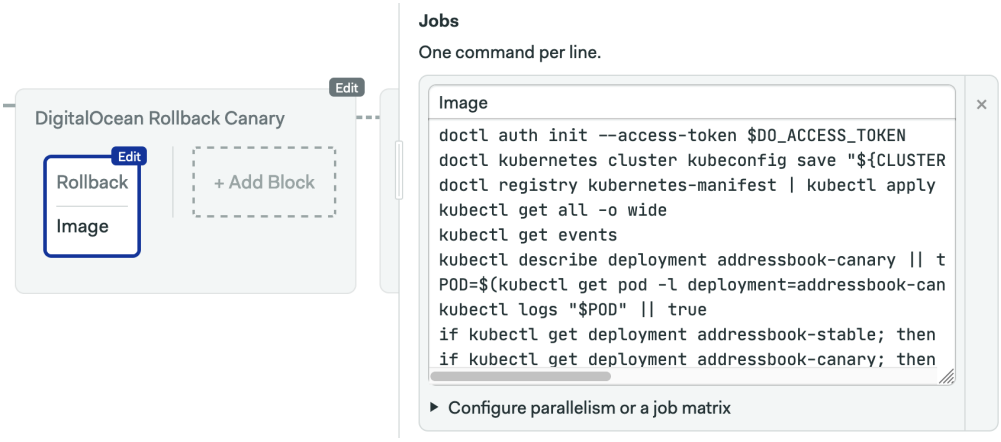


Create the `STABLE_PODS` parameter with default value "3" to finalize the promotion configuration.

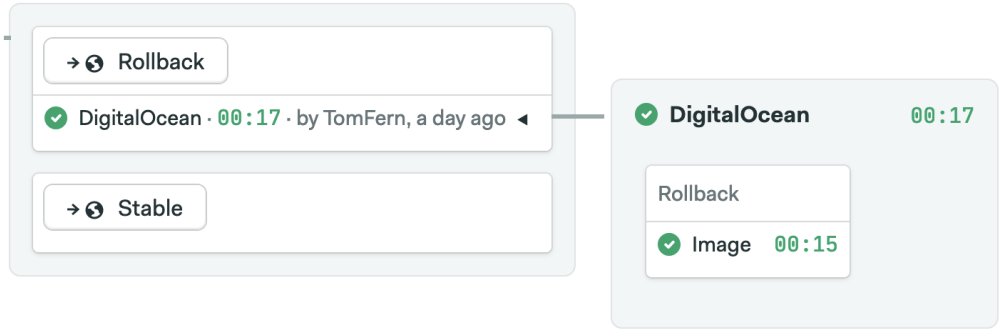
The rollback job collects information to help diagnose the problem. Create a new block called "Rollback Canary", import the `do-ctl` secret, and create `CLUSTER_NAME` and `REGISTRY_NAME`. Type these lines in the job:

```
doctl auth init --access-token $DO_ACCESS_TOKEN
doctl kubernetes cluster kubeconfig save "${CLUSTER_NAME}"
doctl registry kubernetes-manifest | kubectl apply -f -
kubectl get all -o wide
kubectl get events
kubectl describe deployment addressbook-canary || true
POD=$(kubectl get pod -l deployment=addressbook-canary -o name | head -n 1)
kubectl logs "$POD" || true

if kubectl get deployment addressbook-stable; then \
  kubectl scale --replicas=$STABLE_PODS \
  deployment/addressbook-stable; \
fi

if kubectl get deployment addressbook-canary; then \
  kubectl delete deployment/addressbook-canary;  \
fi
```

The first four lines print out information about the cluster. The last two, undoes the changes by scaling up the stable deployment and removing the canary.



Run the workflow once more and make a canary release, but this time try rolling back by clicking on its promote button:



And we're back to normal, phew! Now it's time to check the job logs to see what went wrong and fix it before merging to master again.

But what if we discover a problem after we deploy a stable release? Let's imagine that a defect sneaked its way into production. It can happen, maybe there was some subtle bug that no one found hours or days in. Or perhaps some error was not picked up by the functional test. Is it too late? Can we go back to the previous version?

The answer is yes, we can go to the previous version, but manual intervention is required. Remember that we tagged each Docker image with a unique ID (the `SEMAPHORE_WORKFLOW_ID`)? We can re-promote the stable deployment pipeline from the last good version in Semaphore. If the Docker image is no longer in the registry, we can regenerate it using the *Rerun* button in the top right corner.

4.8.5 Troubleshooting and Tips

Even the best plans can fail; a failure is certainly an option in software development. Maybe the canary is presented with some unexpected errors, perhaps it has performance problems, or we merged the wrong branch into master. The important thing is (1) learn something from them and (2) know how to go back to solid ground.

Kubectl can give us a lot of insights into what is happening. First, get an overall picture of the resources on the cluster.

```
$ kubectl get all -o wide
```

Describe can show detailed information of any or all your pods:

```
$ kubectl describe <pod-name>
```

It also works with deployments:

```
$ kubectl describe deployment addressbook-stable
$ kubectl describe deployment addressbook-canary
```

And services:

```
$ kubectl describe service addressbook-lb
```

We also see the events logged on the cluster with:

```
$ kubectl get events
```

And the log output of the pods using:

```
$ kubectl logs <pod-name>
$ kubectl logs --previous <pod-name>
```

If you need to jump in one of the containers, you can start a shell as long as the pod is running with:

```
$ kubectl exec -it <pod-name> -- sh
```

To access a pod network from your machine, forward a port with `port-forward`, for instance:

```
  $ kubectl port-forward <pod-name> 8080:80
```

These are some common error messages that you might run into:

- Manifest is invalid: it usually means that the manifest YAML syntax is incorrect. Use `kubectl --dry-run` or `--validate` options to verify the manifest.
- `ImagePullBackOff` or `ErrImagePull`: the requested image is invalid or was not found. Check that the image is in the registry and that the reference in the manifest is correct.
- `CrashLoopBackOff`: the application is crashing, and the pod is shutting down. Check the logs for application errors.
- Pod never leaves `Pending` status: this could mean that one of the Kubernetes secrets is missing.
- Log message says that "container is unhealthy": this message may show that the pod is not passing a probe. Check that the probe definitions are correct.
- Log message says that there are "insufficient resources": this may happen when the cluster is running low on memory or CPU.

4.9 Summary

You have learned how to put together the puzzle of CI/CD, Docker, and Kubernetes into a practical application. In this chapter, you have put into practice all that you've learned in this book:

- How to set up pipelines in Semaphore CI/CD and use them to deploy to the cloud.
- How to build Docker images and start a dev environment with the help of Docker Compose.
- How to do canary deployments and rolling updates in Kubernetes.
- How to scale deployments and how to recover when things don't go as planned.

Each piece had its role: Docker brings portability, Kubernetes adds orchestration, and Semaphore CI/CD drives the test and deployment process.

5 Final Words

Congratulations, you've made it through the entire book. We wrote it with the goal to help you deliver great cloud native applications. Now it's up to you—go make something awesome!

5.1 Share This Book With The World

Please share this book with your colleagues, friends and anyone who you think might benefit from it.

Share the book on Twitter:

> *Learn CI/CD with Docker and Kubernetes with this free ebook by @semaphoreci: https://bit.ly/3bJELLQ (Click to Tweet!)*

You can also share it on Facebook, share it on LinkedIn and star the repository on GitHub.

5.2 Tell Us What You Think

We would absolutely love to hear your feedback. What did you get out of reading this book? How easy/hard was it to follow? Is there something that you'd like to see in a new edition?

This book is open source and available at https://github.com/semaphoreci/book-cicd-docker-kubernetes.

- Send comments and feedback, ask questions, and report problems by opening a new issue.
- Contribute to the quality of this book by submitting pull requests for improvements to explanations, code snippets, etc.
- Write to us privately at learn@semaphoreci.com.

5.3 About Semaphore

Semaphore https://semaphoreci.com helps developers continuously build, test and deploy code at the push of a button. It provides the fastest, enterprise-grade CI/CD pipelines as a serverless service. Trusted by thousands of organizations around the globe, Semaphore can help your team move faster too.