Creating a Continuous Delivery Pipeline

Getting Started With Google Kubernetes Engine

Version 1.5

Google Cloud

Now that you've seen how to deploy your applications to Kubernetes, you'll see how to setup a continuous delivery pipeline.

Your tool may vary, and it may but you'll want to set this up for at least one tool in a controlled environment because there are lots of steps to get right.

In this case, you're going to use Jenkins.

Agenda

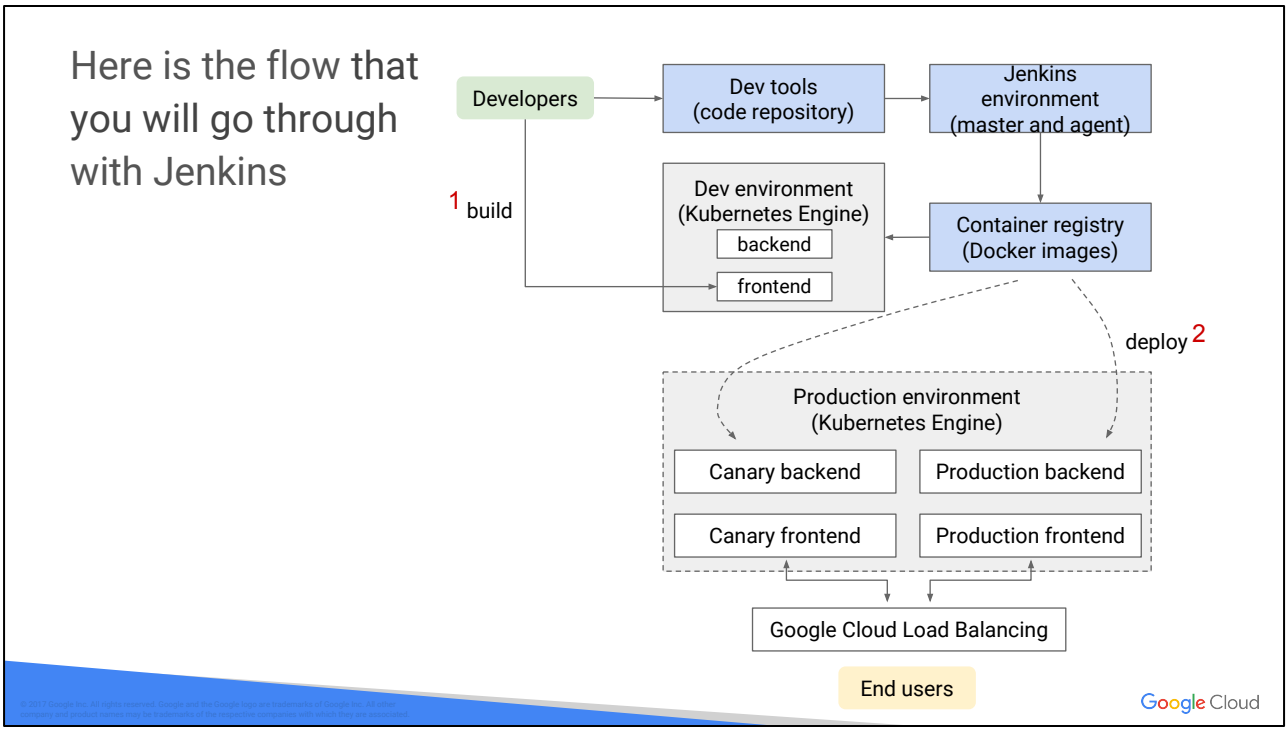
Introduction to Jenkins

Provisioning Jenkins

Understanding the application

Creating the Jenkins pipeline

Deploying a Canary release



Blue boxes represent the build phase in Jenkins. Gray boxes represent dev and production deployments for your application.

1. Developers (top-left green box) check in code to a code repository (top-left blue box). That change is picked up by Jenkins (top-right blue box). Jenkins builds a Docker image from the source code (mid-right blue box) and deploys that to a developer environment for building (mid-left gray box). From there, developers and test and iterate on that code branch in an environment similar to their production environment that is not being hit by live traffic.

2. When they verify the code, they commit their changes to a different branch. That commit changes to a Canary deployment in production (left half of big gray box). As you saw earlier, with a Canary deployment, you're only spinning up a subset of pods and responding to a portion of live traffic. When the Canary backend has been verified, developers merge that code to a production branch (blue boxes again). When te changes are picked up by Jenkins, the image can be built and sent to the rest of the fleet (right half of big gray box) that is serving end users.

Agenda

Introduction to Jenkins

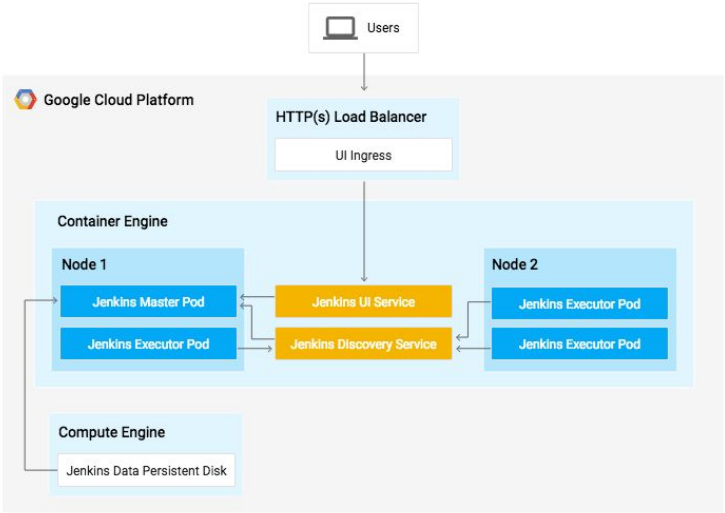
Provisioning Jenkins

Understanding the application

Creating the Jenkins pipeline

Deploying a Canary release

Here is how Jenkins gets deployed to Kubernetes



Google Cloud

In order to deploy Jenkins, you use an **ingress** resource in Kubernetes. An ingress resource maps an HTTP load balancer to your pods.

Within Container Engine, you have a master Jenkins pod (top-left blue box in Node 1) that has two services: one for the user interface (top yellow box), which is where you configure your pipeline; the other is for a discovery service (bottom yellow box).

As Jenkins tries builds, it launches pods in your cluster (bottom-left blue box in Node 1), and those pods communicate back with the master to figure out what to do (flow to bottom yellow box and up to top-left blue box in Node 1).

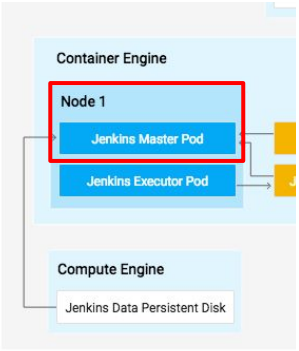
When the master requests the work, the pods run the build steps you told them to do, and then terminate.

Jenkins also has a persistent disk (bottom-left) which stores its configuration data.

Jenkins is run through a Kubernetes deployment

For the master, you define:

- 1 replica
- Image
- Ports
- Mount volume and path




```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jenkins
  namespace: jenkins
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: master
    spec:
      containers:
      - name: master
        image: jenkins:1.642.4
        ports:
        - containerPort: 8080
        - containerPort: 50000
        env:
        volumeMounts:
        - mountPath: /var/jenkins_home
          name: jenkins-home
      volumes:
      - name: jenkins-home
        gcePersistentDisk:
          pdName: jenkins-home
          fsType: ext4
          partition: 1
```


Google Cloud

Jenkins is run through a Kubernetes deployment with replicas set to 1. Jenkins does not have high availability so you only run one replica.

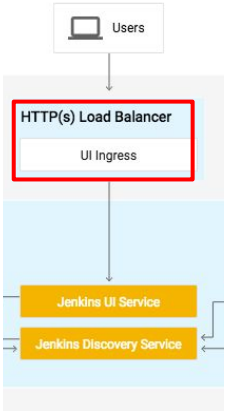
You're going to define which image to use, give it a name, and define which ports are exposed from this container.

And then you're going to mount in your persistent disk that has your configuration data for Jenkins.

So at the bottom you define that volume and mount it onto the **/var/jenkins_home** path inside that container.

For the ingress you define

- TLS cert secret
- Service name
- Service port




```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: jenkins
  namespace: jenkins
spec:
  tls:
  - secretName: tls
  backend:
    serviceName: jenkins-ui
    servicePort: 8080
```


Google Cloud

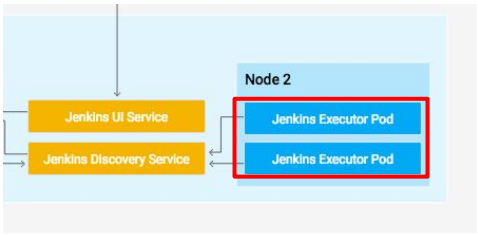
For the ingress, your definition specifies:

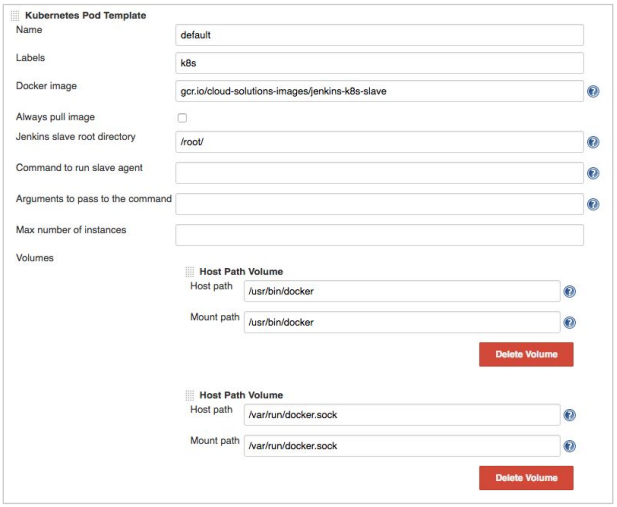
- where the TLS certs are (which is the secret named **tls**)
- what service mounts to this load balancer (in this case, the **service name** is going to be **jenkins-ui**)
- and the port it's going to hit, which is **8080**.

The Jenkins executors (agents) are defined inside Jenkins

You define

- Your Docker image to run
- Docker binary/socket





Google Cloud

The Jenkins executors are defined inside Jenkins in the Kubernetes Pod Template configuration.

In there, you say which Docker image you want to run on each build (3rd field from the top)

Another interesting thing you're doing is leveraging the Docker binary and socket from the node that this pod is running on so you can use Docker within your build to build the image (two bottom right fields).

Agenda

Introduction to Jenkins

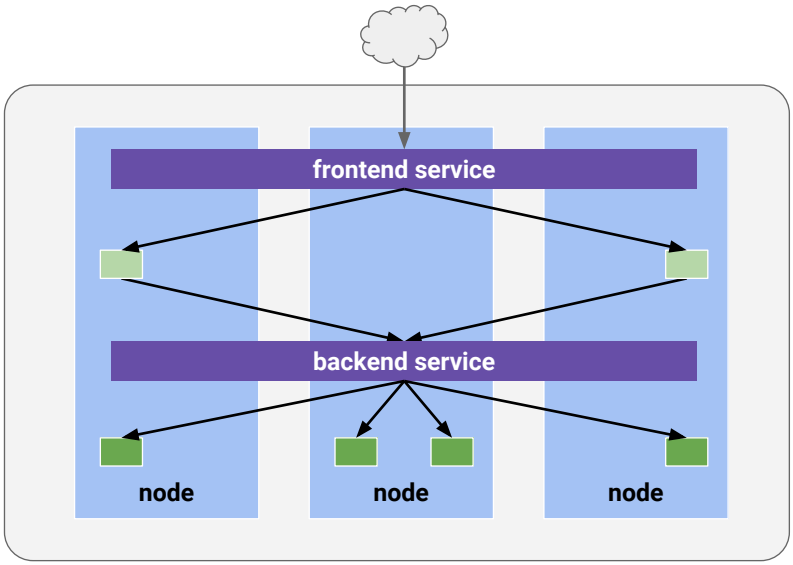
Provisioning Jenkins

Understanding the application

Creating the Jenkins pipeline

Deploying a Canary release

The application has a frontend and backend (frontend exposed to the Internet)



Google Cloud

The application that you're going to run is very similar to the one mentioned before.

The front end is exposed to the internet and talks to the backend in order to complete the requests.

You'll have two services, each with their own set of pods, and running on one cluster.

Agenda

Introduction to Jenkins

Provisioning Jenkins

Understanding the application

Creating the Jenkins pipeline

Deploying a Canary release

You build the Jenkins pipeline that defines how your build, test, and deploy cycle managed

Google Cloud

Then you'll deploy your application using the Jenkins pipeline.

Jenkins pipeline allows you to create a set of steps, in code, and check it into source code management that defines how your build, test, and deploy cycle will be orchestrated.

Example Jenkins pipeline file with checkout, build, test, push, and deployment

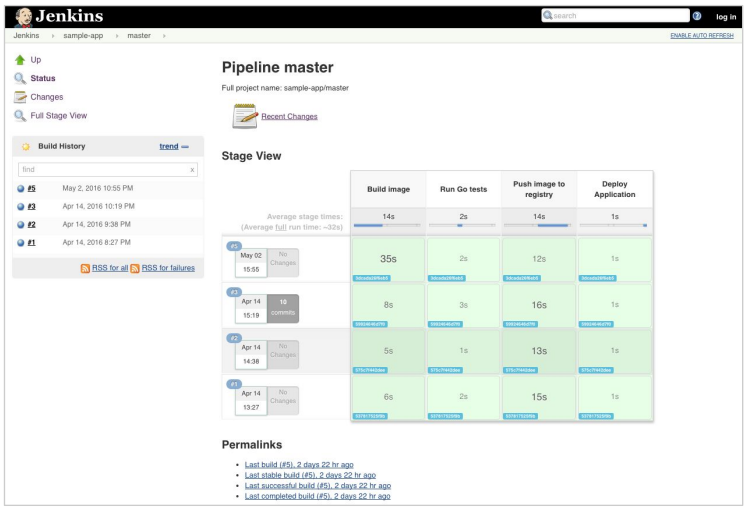
```
node {
  def project = 'vic-goog'
  def appName = 'gceme'
  def feSvcName = "${appName}-frontend"
  def imageTag =
"gcr.io/${project}/${appName}:${env.BUILD_NUMBER}"

  checkout scm

  stage 'Build image'
  sh("docker build -t ${imageTag} .")

  stage 'Run Go tests'
  sh("docker run ${imageTag} go test")

  stage 'Push image to registry'
  sh("gcloud docker push ${imageTag}")

  stage "Deploy Application"
  sh("sed -i.bak 's#IMAGE_NAME#${imageTag}#' ./k8s/*.yaml")
  sh("kubectl --namespace=production apply -f k8s/")
}
```


Google Cloud

So here's an example of a Jenkins pipeline file:

- At the top, you define some variables
- Next, you checkout our code from the source code management that has been configured
- In the first stage, you build an image from our source
- In the second stage, you run tests after that image has been built
- In the third stage, you push that image once the tests pass
- In the fourth stage, if the image push is successful, it's going to deploy our application using kubectl which is baked into our container image.

A configured pipeline has run a few times with different stages, times, status, and logs



Google Cloud

Here's what it looks like when a pipeline is configured and it's been run a few times.

You see the different stages that have been set up. It tells you how long each stage is, which is interesting for figuring out where you can optimize your time for deployment. But also gives you very clear output on which stages have passed and gives you any easy way to get to your logs for each stage.

Image source is from https://jenkins.io.

Agenda

Introduction to Jenkins

Provisioning Jenkins

Understanding the application

Creating the Jenkins pipeline

Deploying a Canary release

With Canary, you have the same labels across deployments

```
kind: Service
apiVersion: v1
metadata:
  name: frontend
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: awesome-stuff
    role: frontend
```

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: frontend-prod
spec:
  replicas: 90
  template:
    metadata:
      name: frontend
      labels:
        app: awesome-stuff
        role: frontend
        env: prod
    spec:
      containers:
      - name: frontend
        image: my-img:v1
        ports:
        - name: ui
          containerPort: 80
```

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: frontend-staging
spec:
  replicas: 10
  template:
    metadata:
      name: frontend
      labels:
        app: awesome-stuff
        role: frontend
        env: staging
    spec:
      containers:
      - name: frontend
        image:my-img:v2
        ports:
        - name: ui
          containerPort: 80
```


Google Cloud

You'll stage a portion of your live release to a Canary deployment for first-user testing.

Canaries can be run at various levels of sophistication. An example of a maturity progression can be found here: https://cloudplatform.googleblog.com/2018/04/introducing-Kayenta-an-open-automated-canary-analysis-tool-from-Google-and-Netflix.html.

With deploying to Canary, you use the same labels across all deployments.

In this case, you use 'awesome-stuff' app label and a 'frontend' role label to the service for our frontend.

But you have another label to distinguish production from staging

```
kind: Service
apiVersion: v1
metadata:
  name: frontend
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: awesome-stuff
    role: frontend
```

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: frontend-prod
spec:
  replicas: 90
  template:
    metadata:
      name: frontend
      labels:
        app: awesome-stuff
        role: frontend
        env: prod
    spec:
      containers:
      - name: frontend
        image: my-img:v1
        ports:
        - name: ui
          containerPort: 80
```

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: frontend-staging
spec:
  replicas: 10
  template:
    metadata:
      name: frontend
      labels:
        app: awesome-stuff
        role: frontend
        env: staging
    spec:
      containers:
      - name: frontend
        image:my-img:v2
        ports:
        - name: ui
          containerPort: 80
```


Google Cloud

But then you also have an env label that says prod and staging.

So then you can change the prod and staging capacity so that it has only 90% of your traffic going to production and only 10% of your traffic going to staging.

That's how you define how much traffic is goes between prod and staging for a Canary deployment.

Now you've seen an overview of how to set up continuous deployment in Kubernetes using Jenkins.

Next, you'll go through the lab that covers all the details.

Lab