Creating a Continuous Delivery Pipeline

Version 1.6

Google Cloud

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

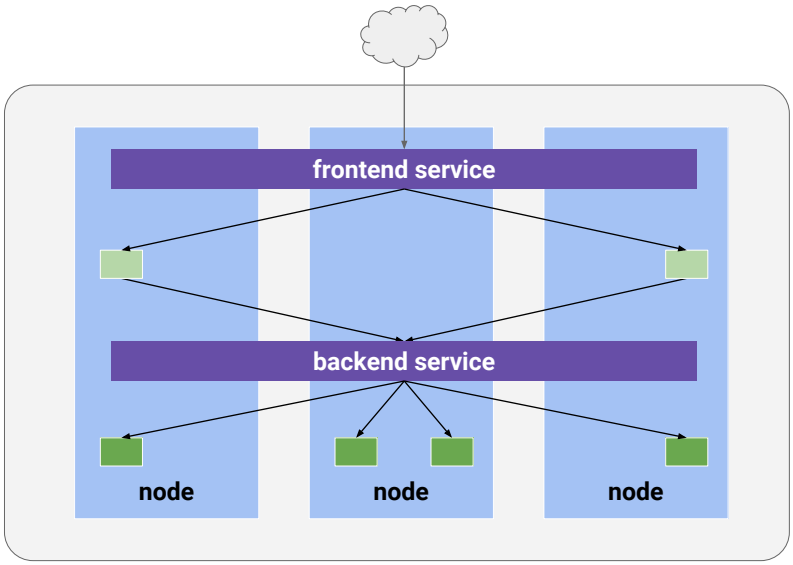
Your tool may vary, and it may but you'll want to set this up for at least one tool in a controlled environment because there are lots of steps to get right.

In this case, you'll use **Spinnaker** or **Jenkins**.

Jenkins is the more popular of the two.

Spinnaker was built by Netflix and released as open source with deployment features to managed cloud services like Kubernetes Engine and AWS.

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



Google Cloud

Here's an overview of the application you're going to build, test, and deploy using continuous delivery.

It's 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 a single cluster.

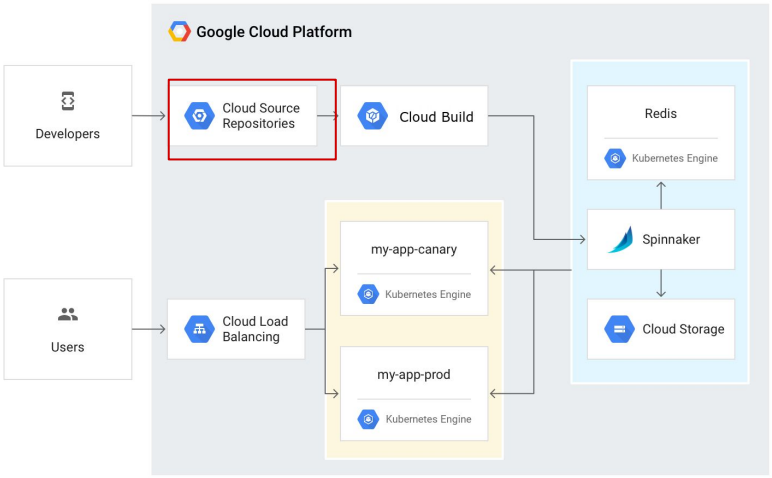
Continuous delivery needs an automated, reliable process

Google Cloud

To continuously deliver app updates to users, you need an automated process that reliably builds, tests, and updates your software. Code changes should automatically flow through a pipeline that includes artifact creation, unit testing, functional testing, and production rollout.

In some cases, you want a code update to apply to only a subset of your users, so that it is exercised realistically before you push it to your entire user base. If one of these canary releases proves unsatisfactory, your automated procedure must be able to quickly roll back the software changes.

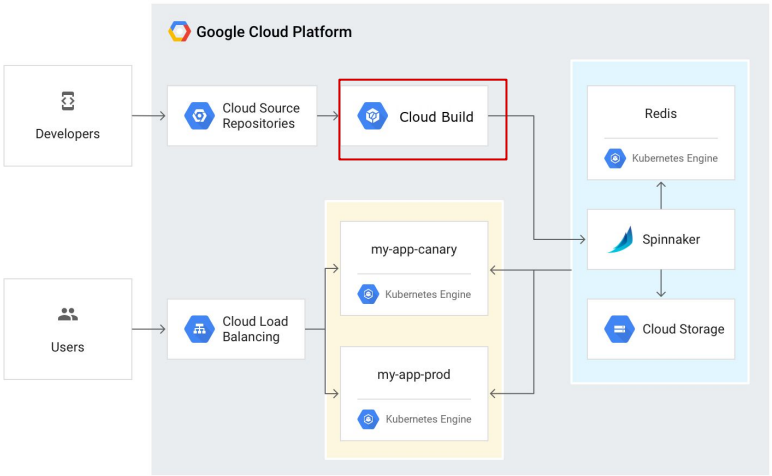
Continuous delivery with Cloud Build and Spinnaker



Google Cloud

In a continuous delivery pipeline with **Spinnaker** and Kubernetes Engine, you can create an app with a Git tag, push it to a Git repository in **Cloud Source Repository**, and configure it to trigger Cloud Build when changes to code occur using the Git tag.

Continuous delivery with Cloud Build and Spinnaker



Google Cloud

You configure **Cloud Build** to detect new Git tag changes, execute a build to your specifications, and produce artifacts such as Docker images, run unit tests, and push images to Spinnaker for deployment.

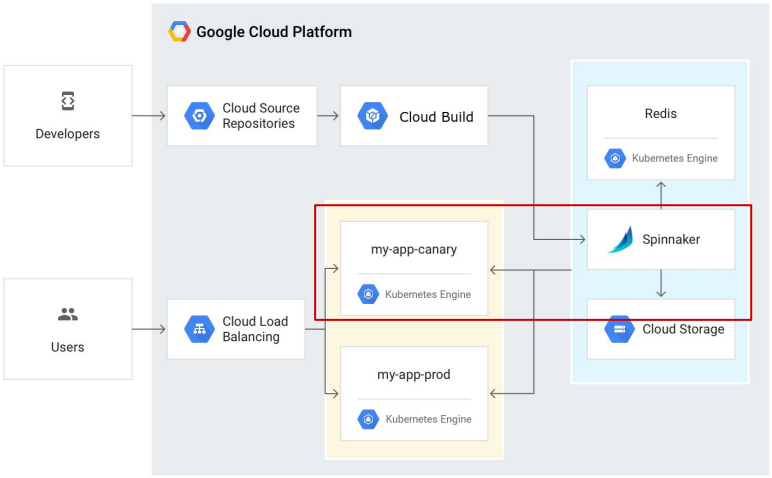
Cloud Build can import source code from a variety of repositories or cloud storage spaces, execute a build to your specifications, and produce artifacts such as Docker images or Java archives.

You can write a build config to provide instructions to Cloud Build on what tasks to perform. You can configure builds to fetch dependencies, run unit tests, static analyses, and integration tests, and create artifacts with build tools such as docker, gradle, maven, bazel, and gulp.

Cloud Build executes your build as a series of build steps, where each build step is run in a Docker container. Executing build steps is analogous to executing commands in a script.

You can either use the build steps provided by Cloud Build and the Cloud Build community, or write your own custom build steps.

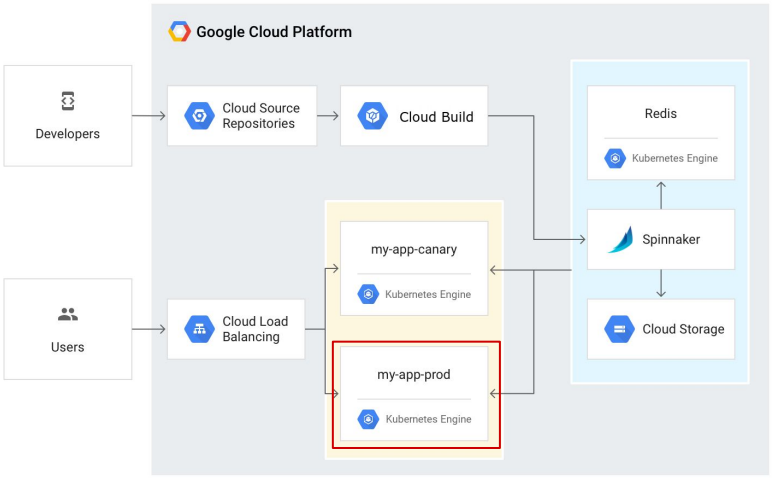
Continuous delivery with Cloud Build and Spinnaker



Google Cloud

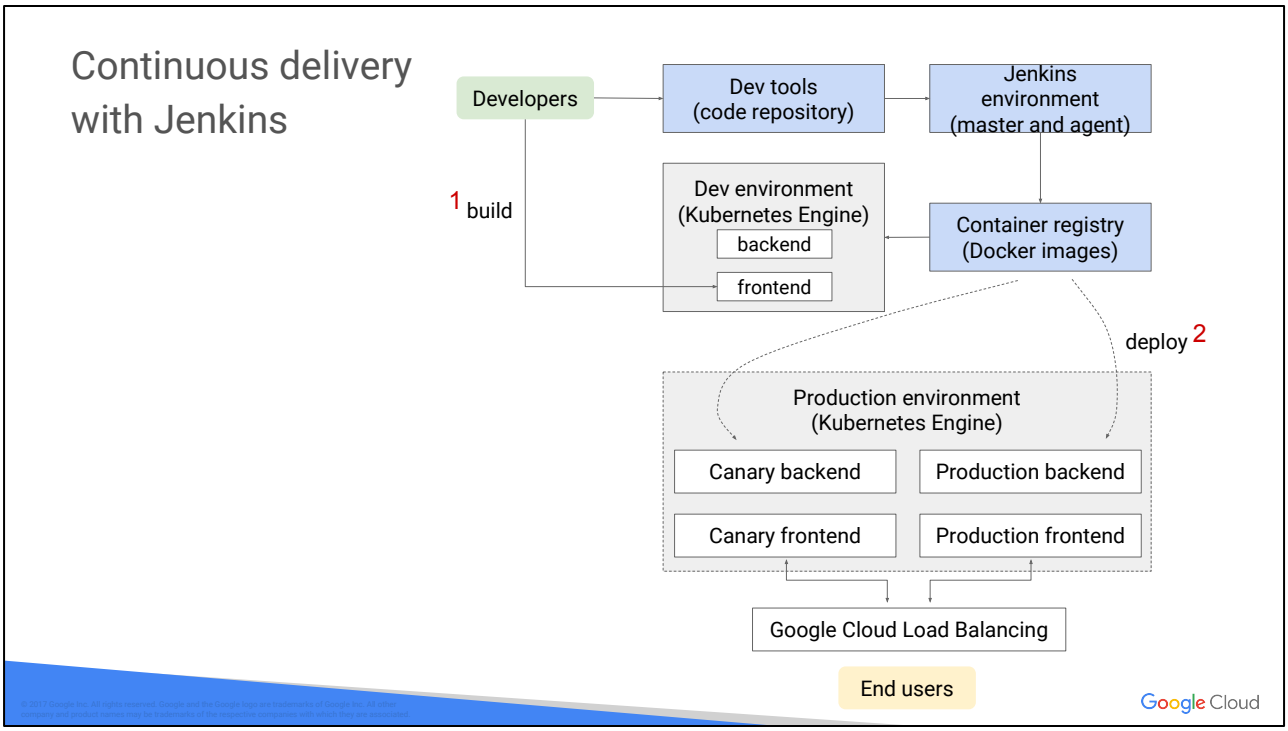
Finally, these changes can trigger the continuous delivery pipeline in Spinnaker to deploy a new version of your code to Canary, perform functional Canary tests, manually approve the changes,

Continuous delivery with Cloud Build and Spinnaker



Google Cloud

and deploy the new version to production.

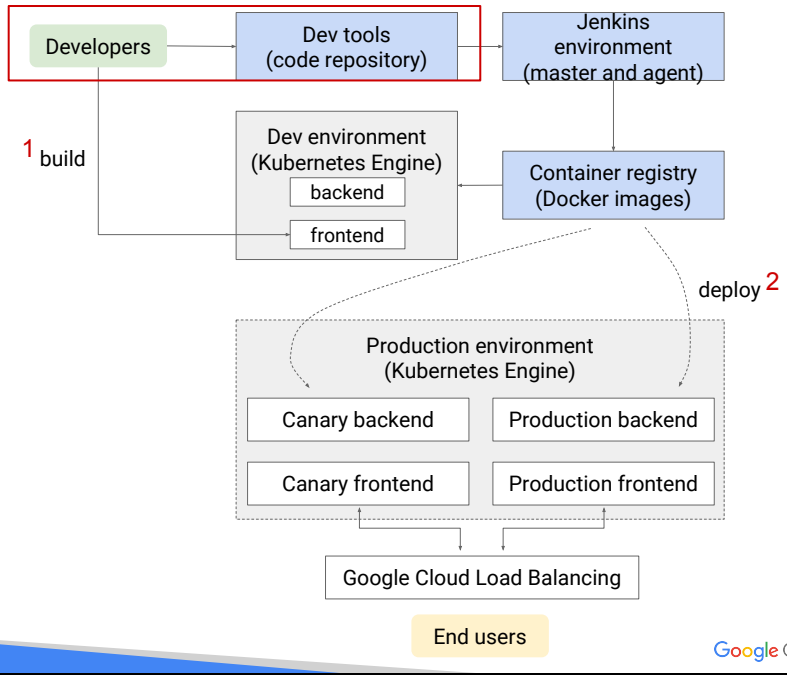


Jenkins' pipeline can be similar.

It 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.

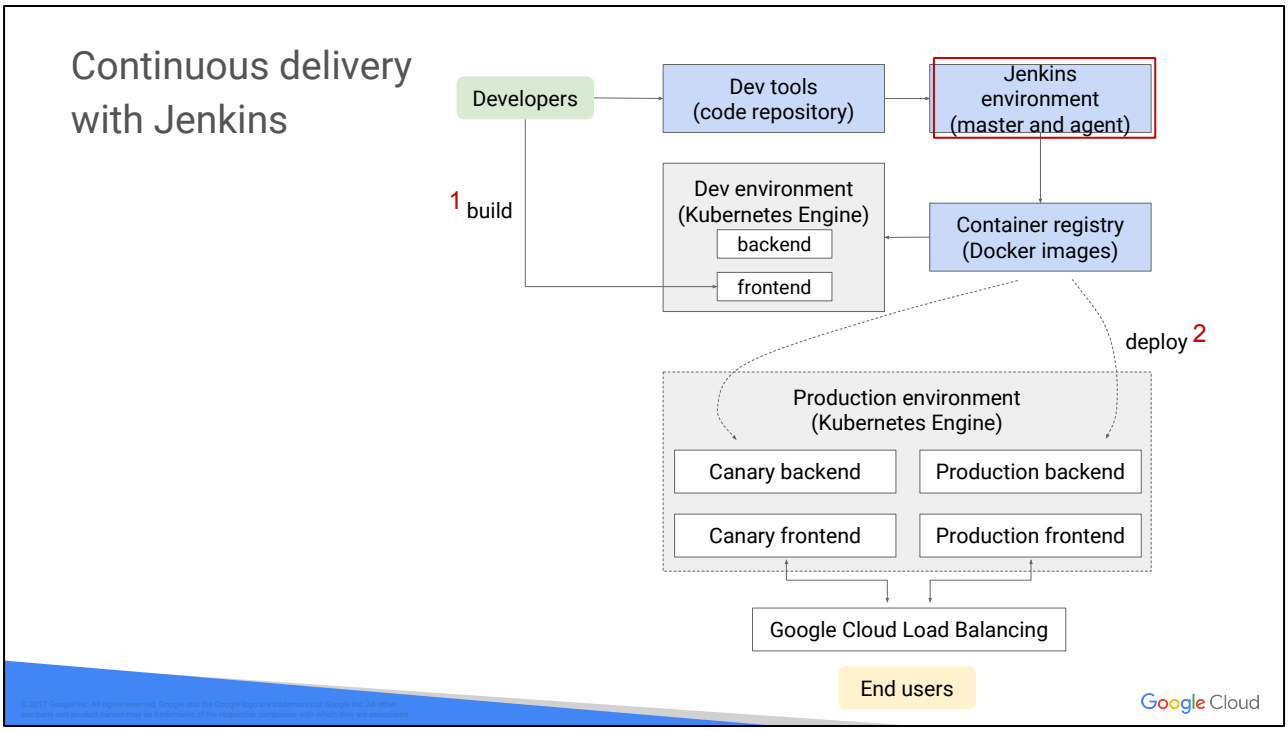
Blue boxes represent the build phase in Jenkins. Gray boxes represent dev and production deployments for your application.

Continuous delivery with Jenkins

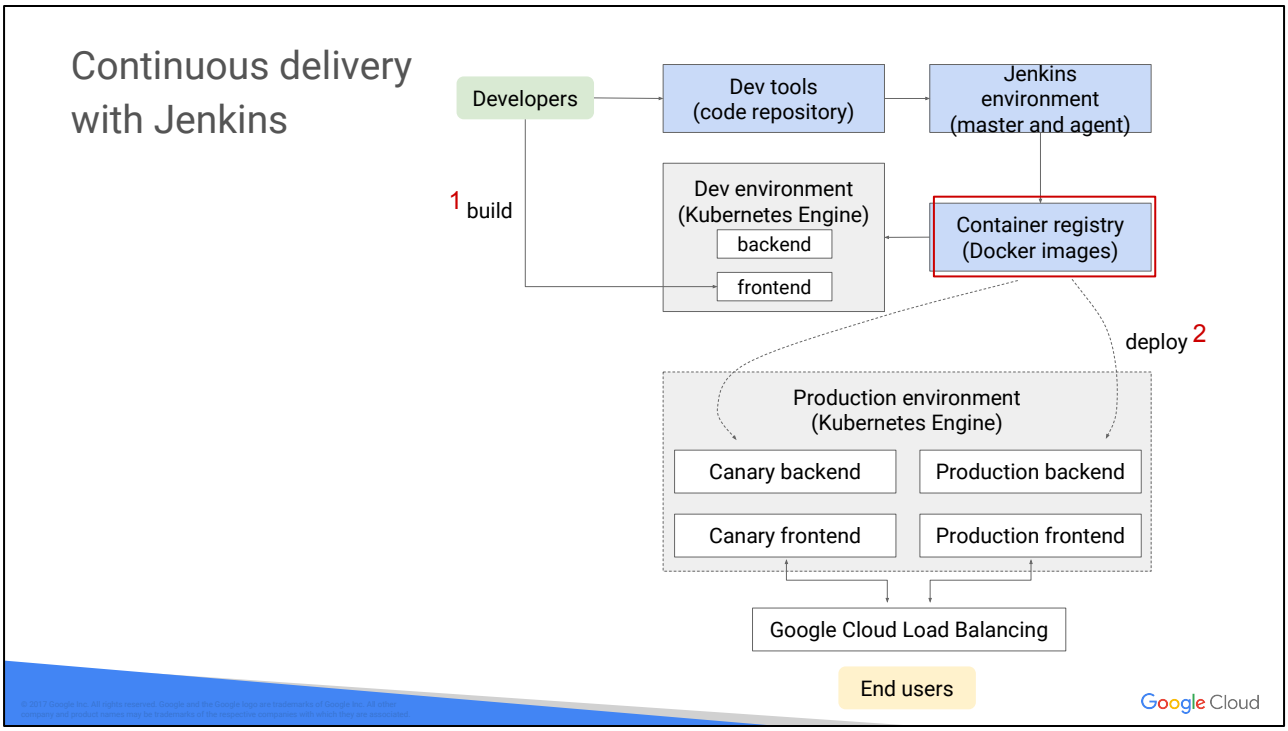


Google Cloud

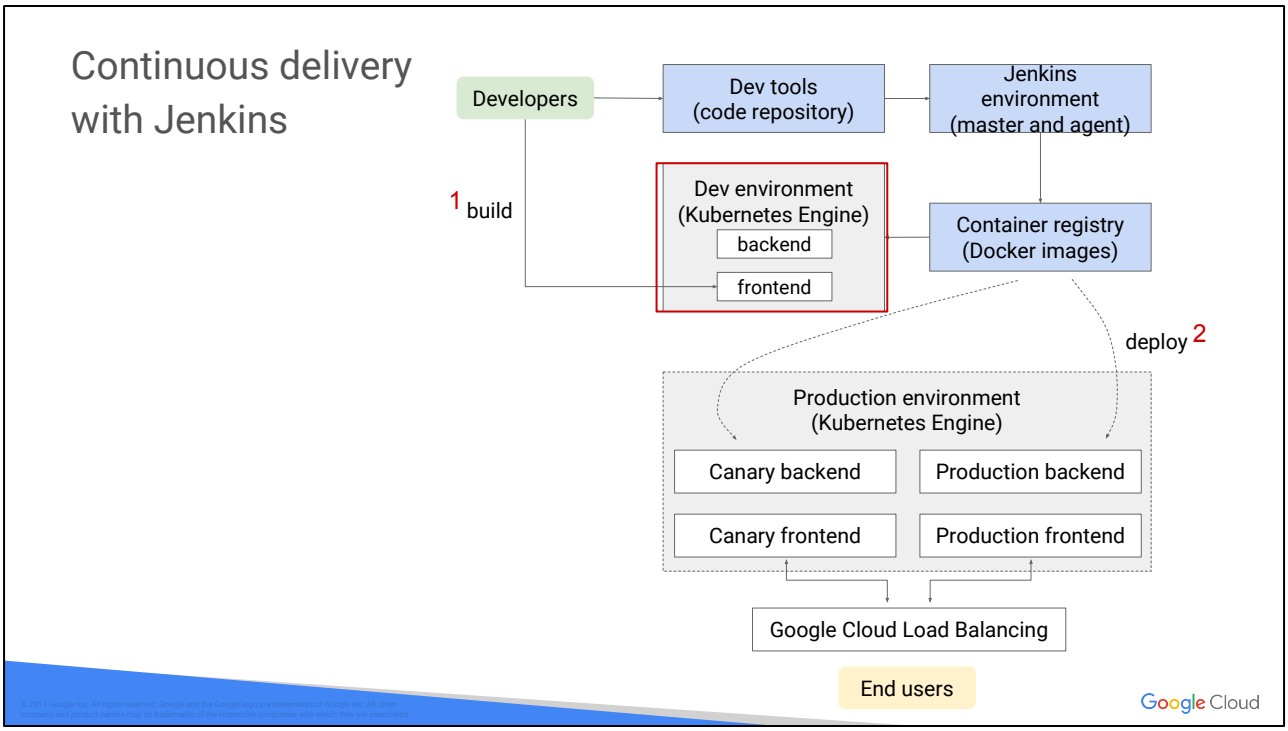
Developers check in code to a repository in Jenkins.



That change is picked up by Jenkins.

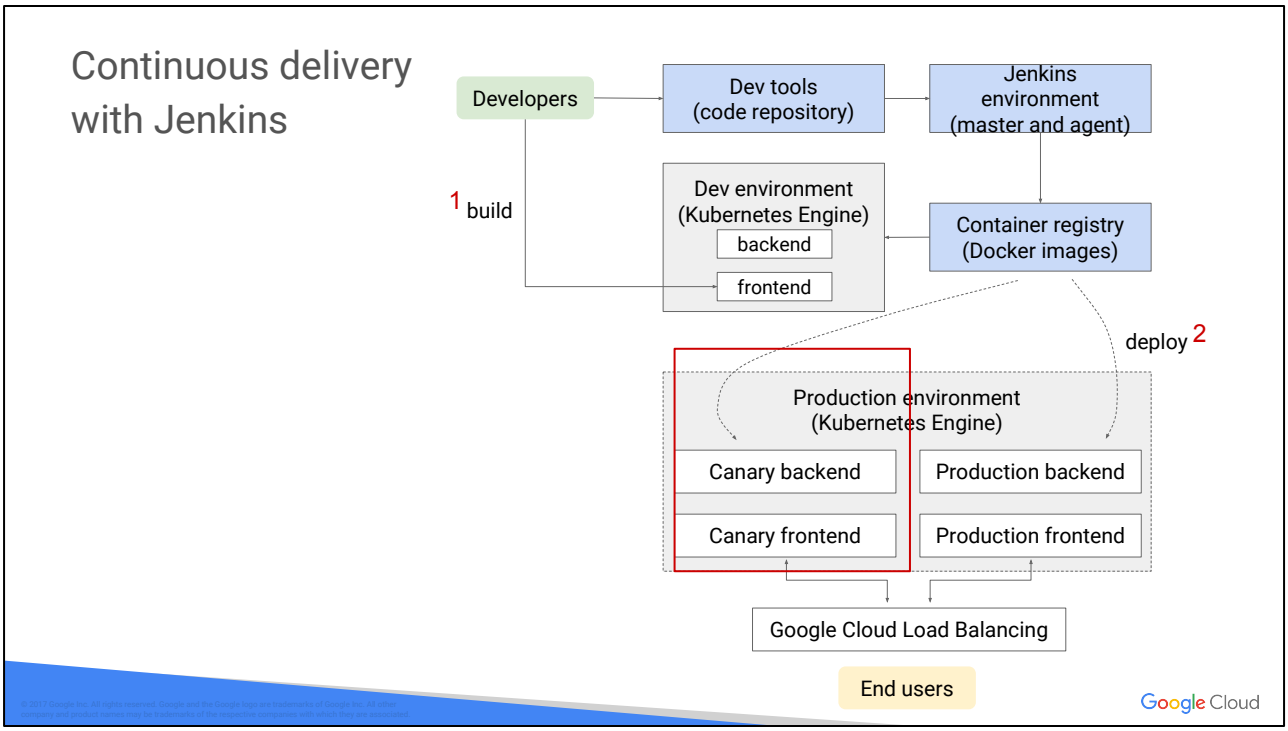


Jenkins builds a Docker image from the source code.



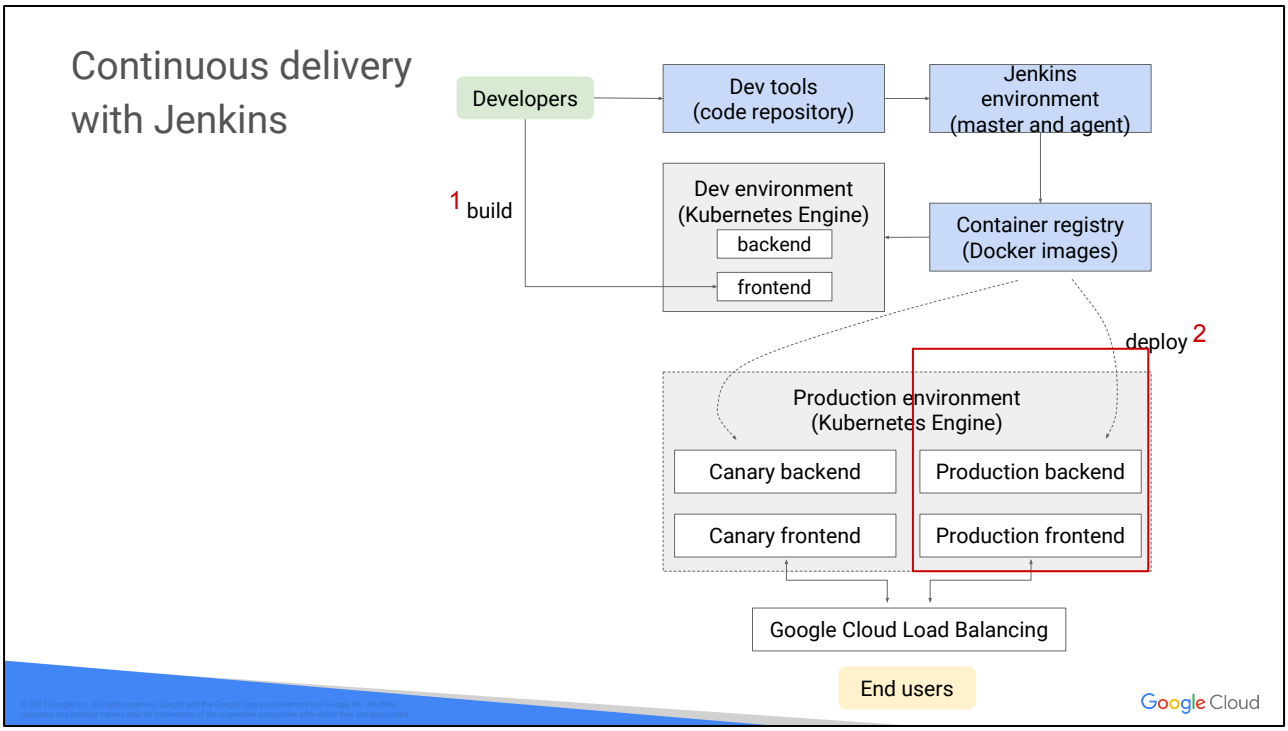
And deploys that to a developer environment for building.

From there, developers and unit test and iterate on that code branch in an environment similar to their production environment that is not being hit by live traffic.



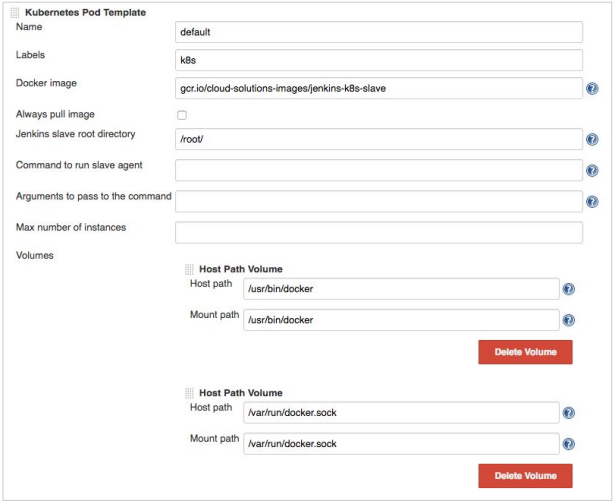
When they verify the unit code, they commit their changes to a different branch. That commit changes to a Canary deployment in production.

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. When the changes are picked up by Jenkins, the image can be built and sent to the rest of the fleet that is serving end users.

Spinnaker and Jenkins get deployed to Kubernetes as applications



Google Cloud

You deploy Spinnaker or Jenkins as Kubernetes applications. They are not stand alone services.

Here is a screenshot of the Jenkins application configuration wizard.

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

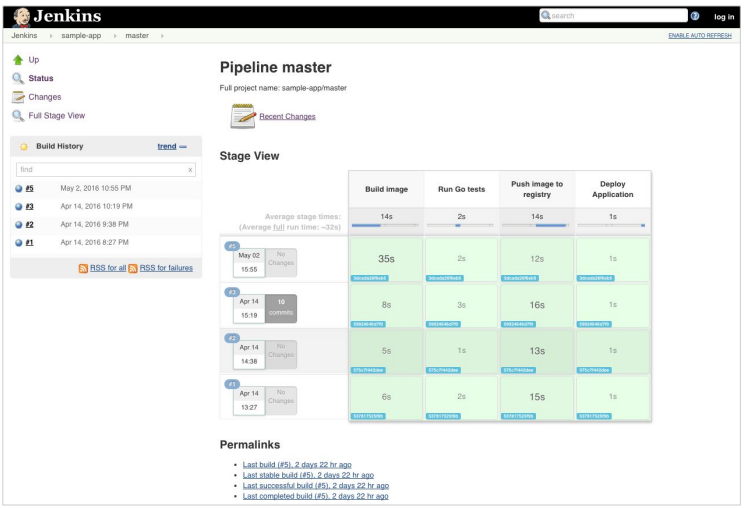
```
node {
  def project = 'vic-goog'
  def appName = 'gceme'
  def feSvcName = "${appName}-frontend"
  def imageTag =
"gcr.io/${project}/${appName}:${env.BUILD_NUMBER}"

  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

And an example of a Jenkins pipeline file:

- You checkout your user application code from a source code repository
- You build an image from your source
- You run tests after that image has been built
- You push that image once the tests pass
- If the image push is successful, it deploys your application using kubectl which is baked into your 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.

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 Spinnaker and Jenkins.

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

Lab