Continuous delivery pipeline

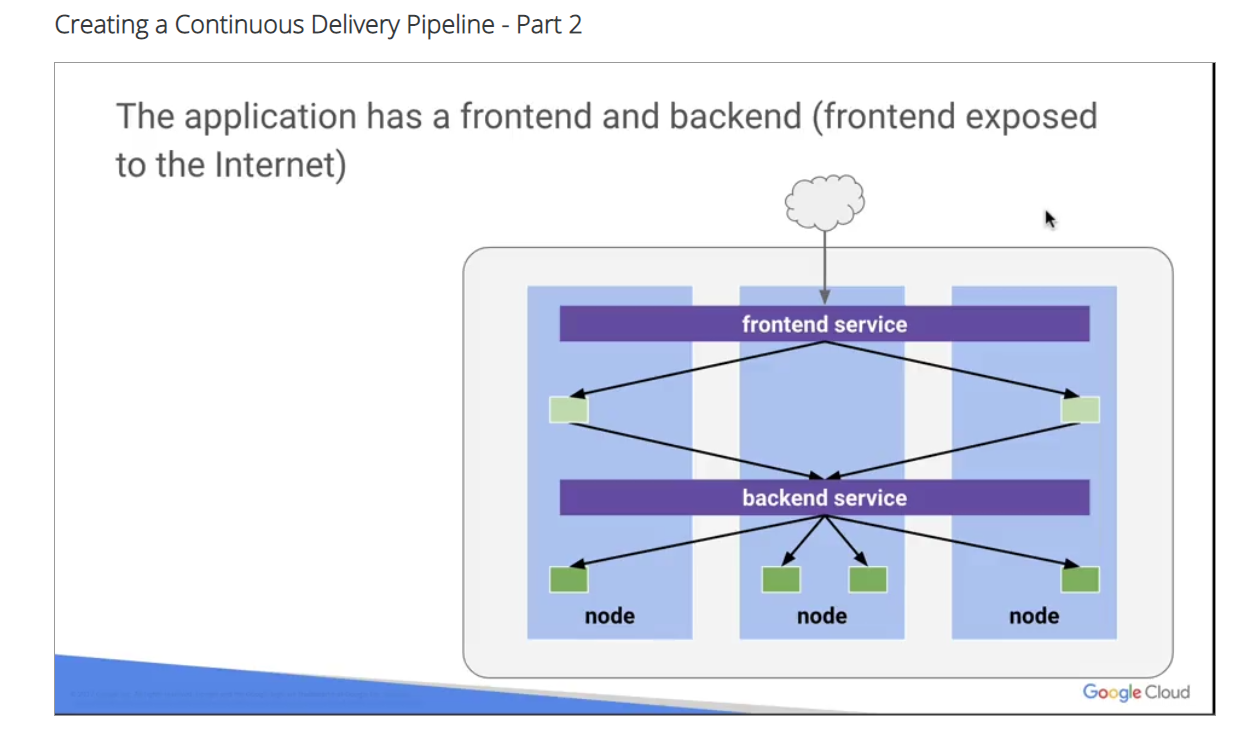
* Part 1

How to deploy a continuous delivery pipeline. This enables you to build, stage, test, and deploy your application, using automated triggers and manual approvals. It's very fast, and it used frequently now in modern day application development. Your tool may vary, but you'll want to set this up at least on one tool in a controlled environment, because there are a lot of steps to get right. In this case, you'll use Spinnaker or Jenkins. Jenkins is the more popular of the two, but Spinnaker was built by Netflix and released as open source deployment features to manage Cloud services like Kubernetes engine and AWS.

* Part 2

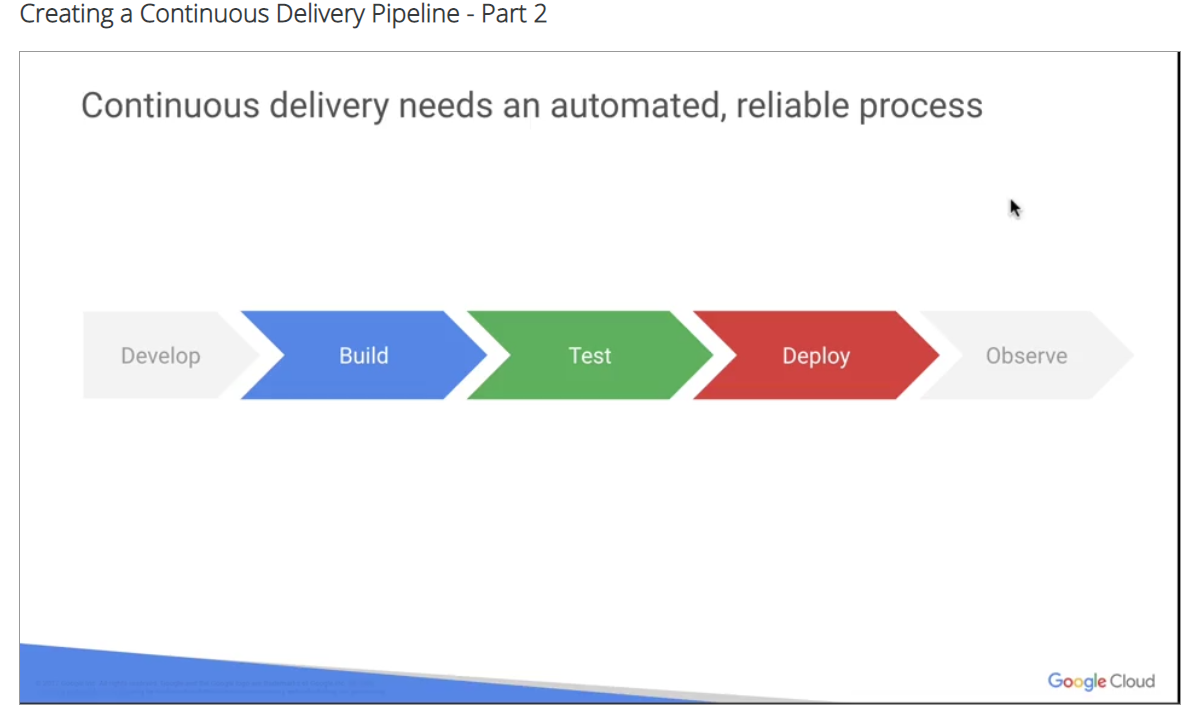
Here's an overview of the application you're going to build, test, and deploy using continuous delivery. It's similar to the one mentioned before.

The front-end is exposed to the internet and talks to the back-end in order to complete the request. You'll have two services each with their own set of pods and running on a single cluster. To continuously deliver application updates to users, you need an automated



process that reliably builds tests and update your software.

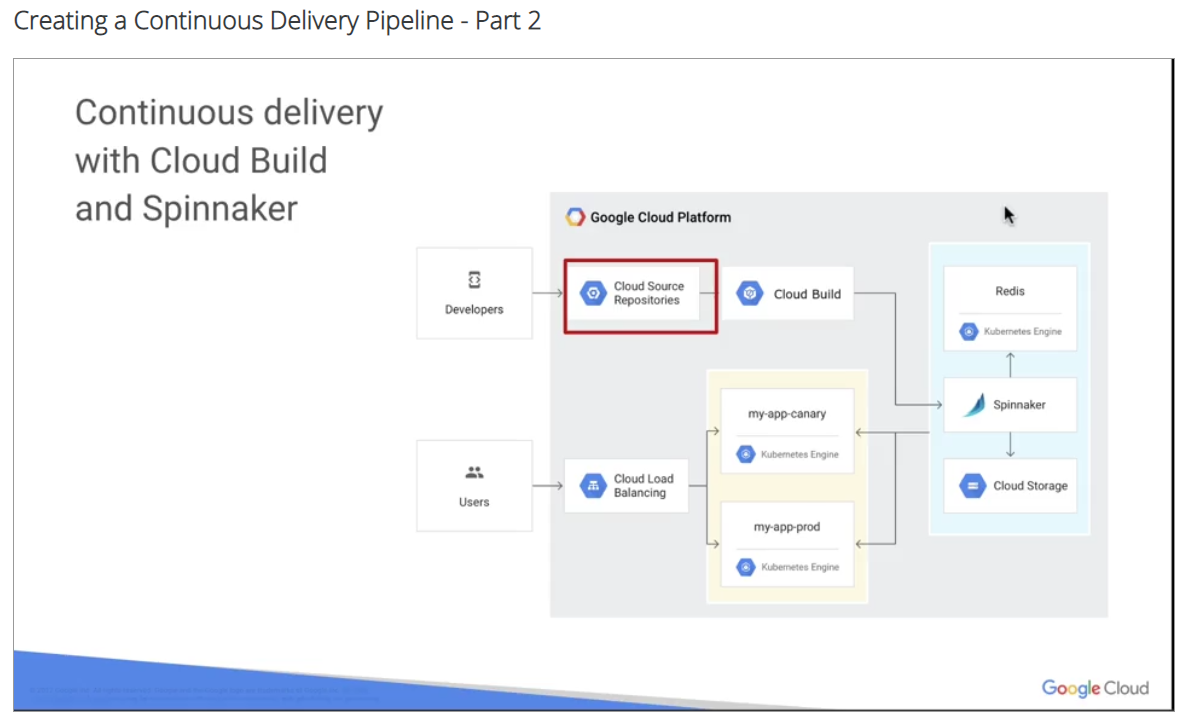
Code changes should automatically flow through a pipeline that includes **artifact creation**, **unit testing**, **functional testing**, and **production rollout**. In some cases, you want a code update to apply to only a subset of your users so that it is exercised realistically before you push it to your entire user base.



If one of these canary releases and proves unsatisfactory, your automated procedure must be able to quickly roll back the software changes.

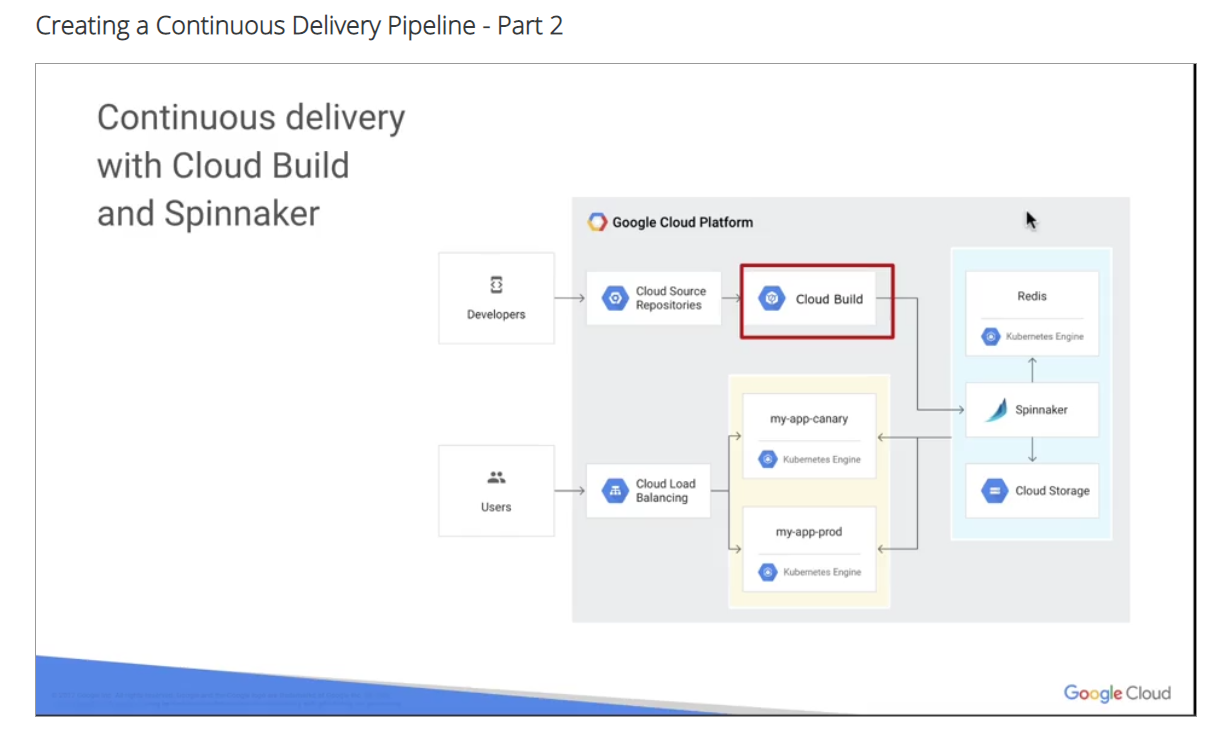
In a continuous delivery pipeline using **Cloud Build**, **Spinnaker** and **Kubernetes engine**, you can

1. create an app with a Git tag
2. push it to the Git repository and Cloud source repository and configure it to
3. trigger Cloud Build when changes to code occur using a change in the Git tag.



You configure

1. Cloud Build to detect new Git tag changes,
2. Execute a bill to your specifications and
3. Produce artifacts such as Docker images,
4. run unit tests,
5. and push images to Spinnaker for deployment.

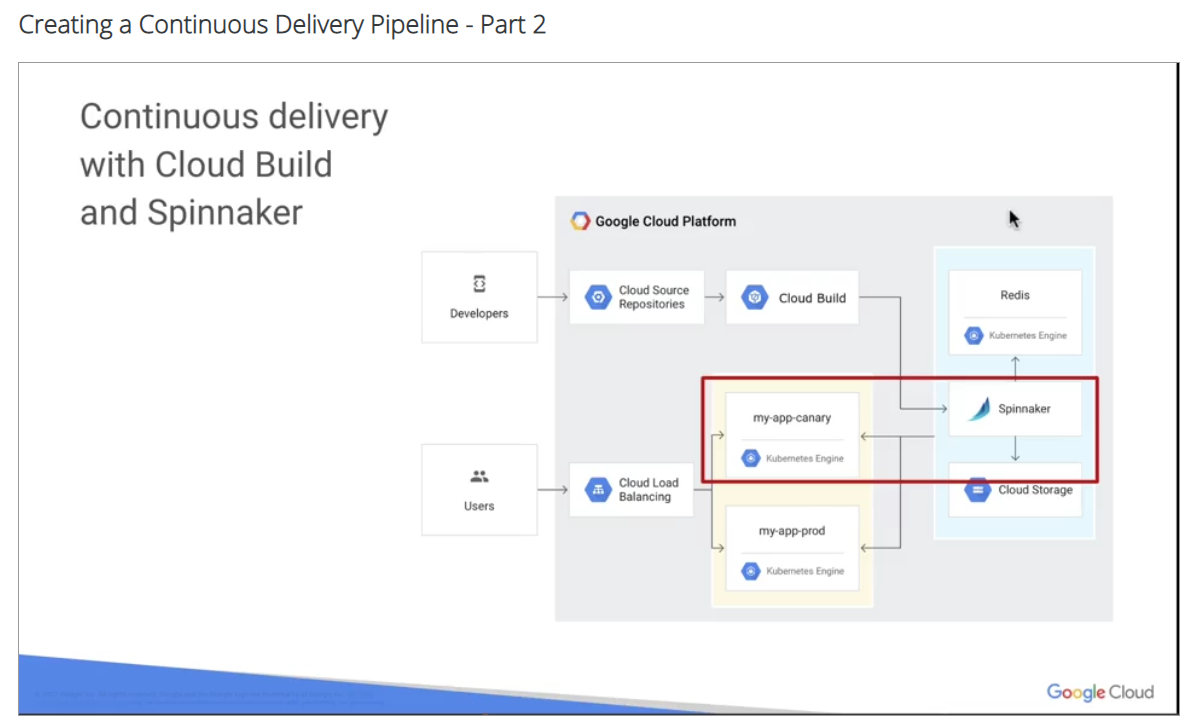
Cloud Build can import source code from a variety of repositories or Cloud Storage Spaces,

execute bill 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 task to perform.

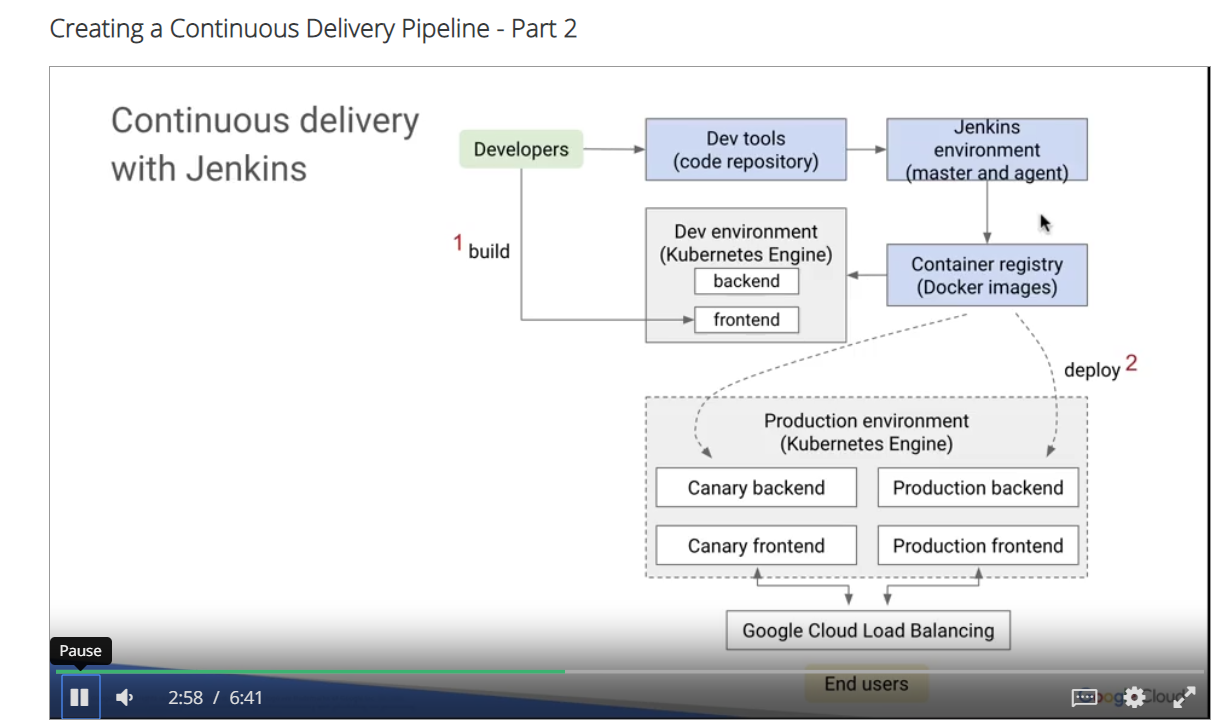
You can configure bills to fetch dependencies, run unit tests, perform static analyses, and integrate tests and create artifacts would build tools such as Docker, Gradle, Maven, Basal and Gulp.

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.

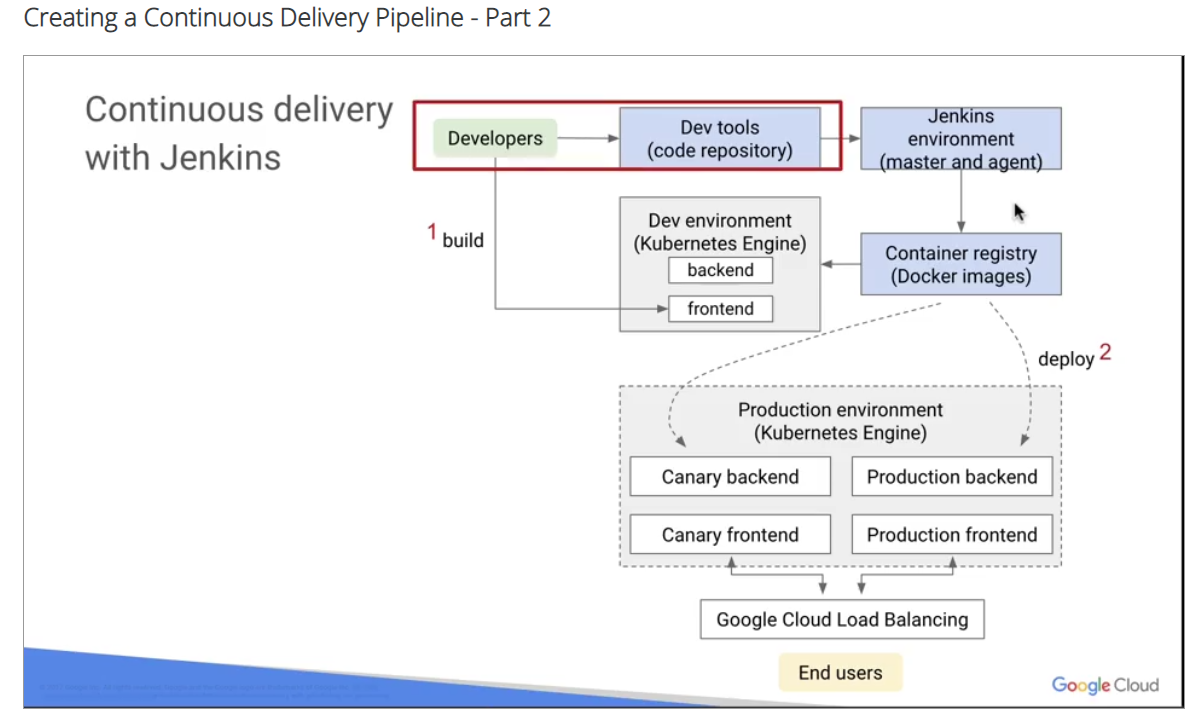


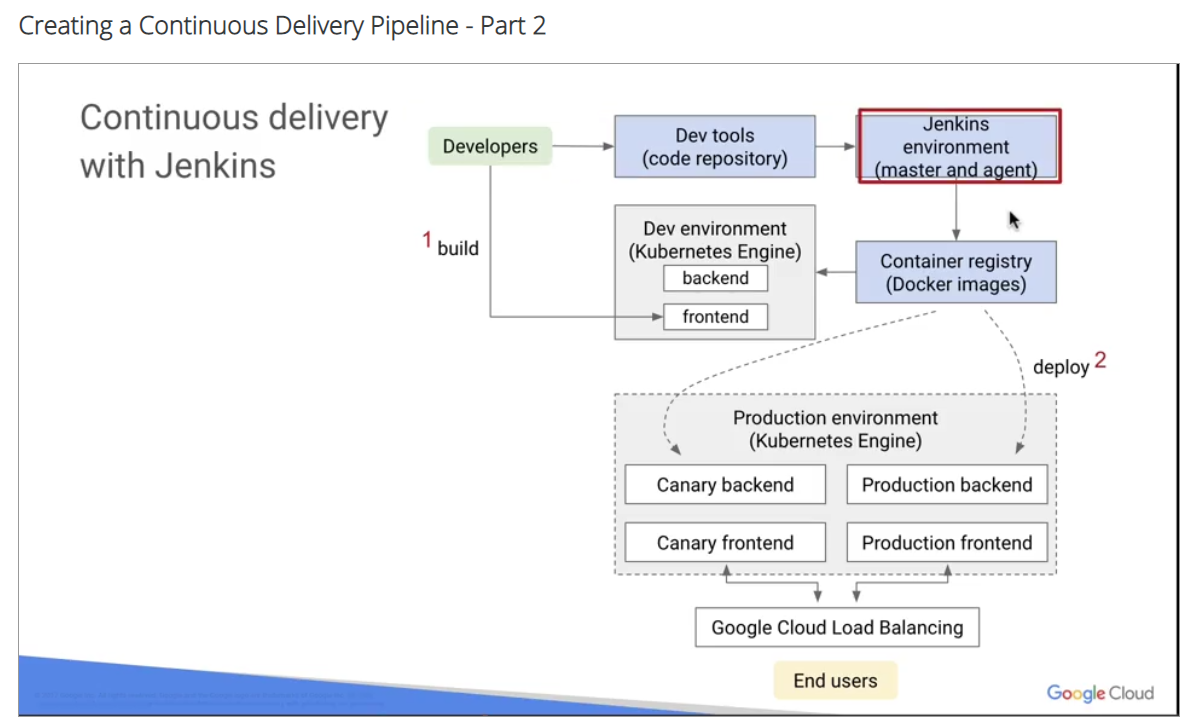
Finally, these changes can trigger the continuous delivery pipeline and Spinnaker to deploy a new version of your code to Canary, perform functional Canary testing and allow you to manually approve the changes.

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

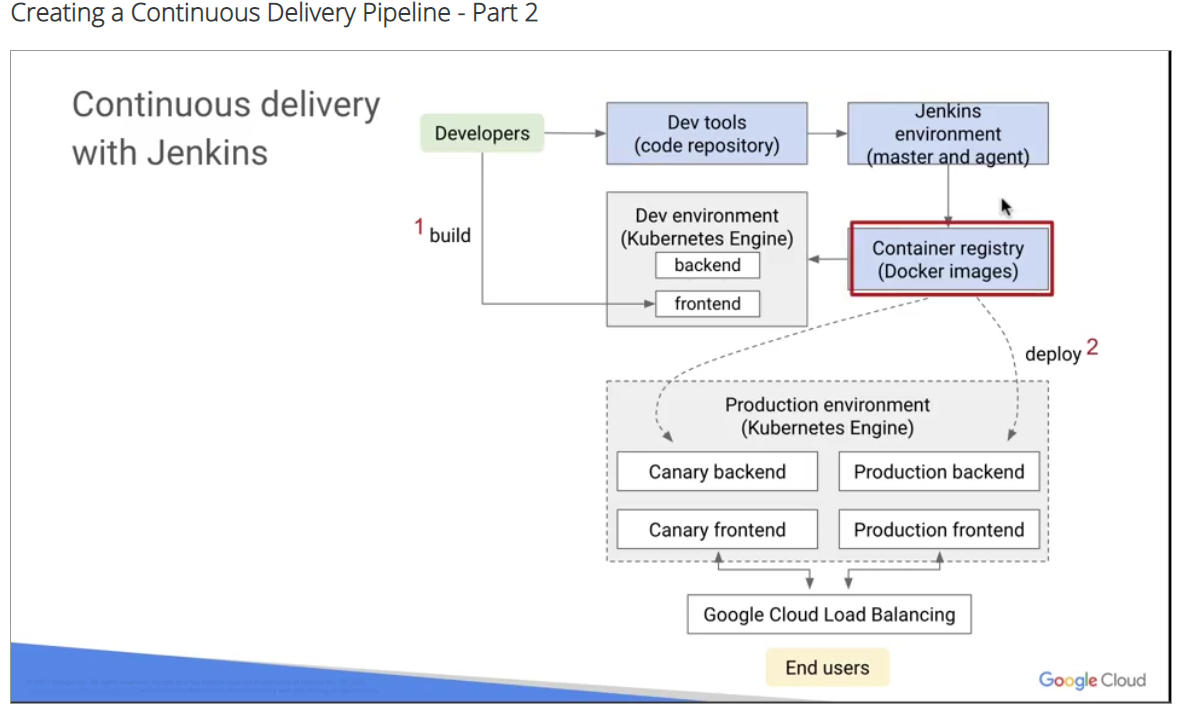


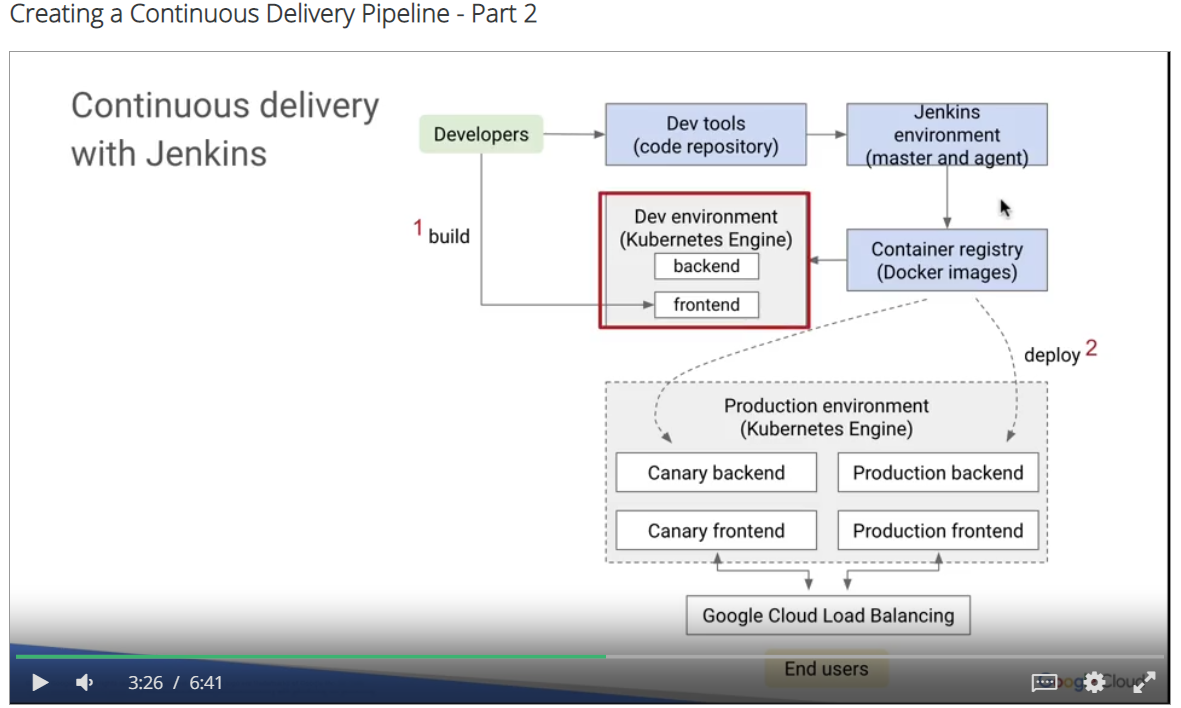
Jenkins pipeline can be similar. It allows you to create a set of steps in code and check it into source code management that defines how your Build test and deploy cycles will be orchestrated. Blue boxes here represent the build phase in Jenkins. Gray boxes represent development and production deployments for your application. Developers checking code to a repository in Jenkins that change is picked up by Jenkins





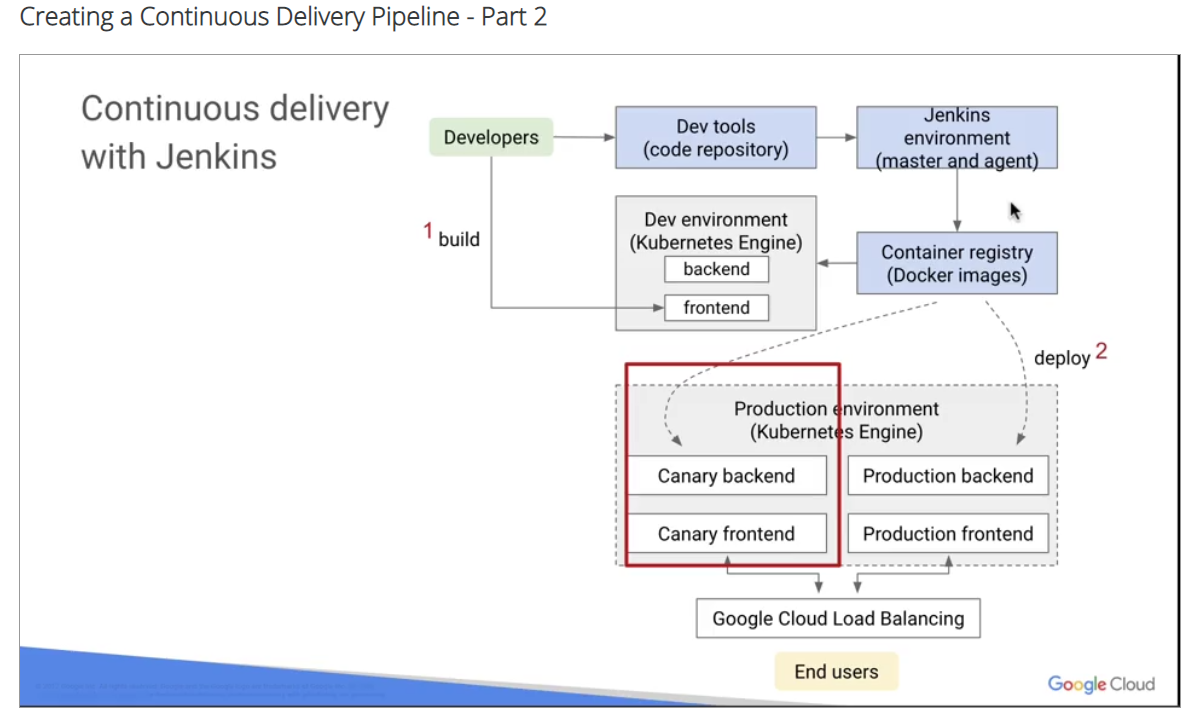
and Jenkins builds a Docker image from the source code and deploys that to a developer environment for building.



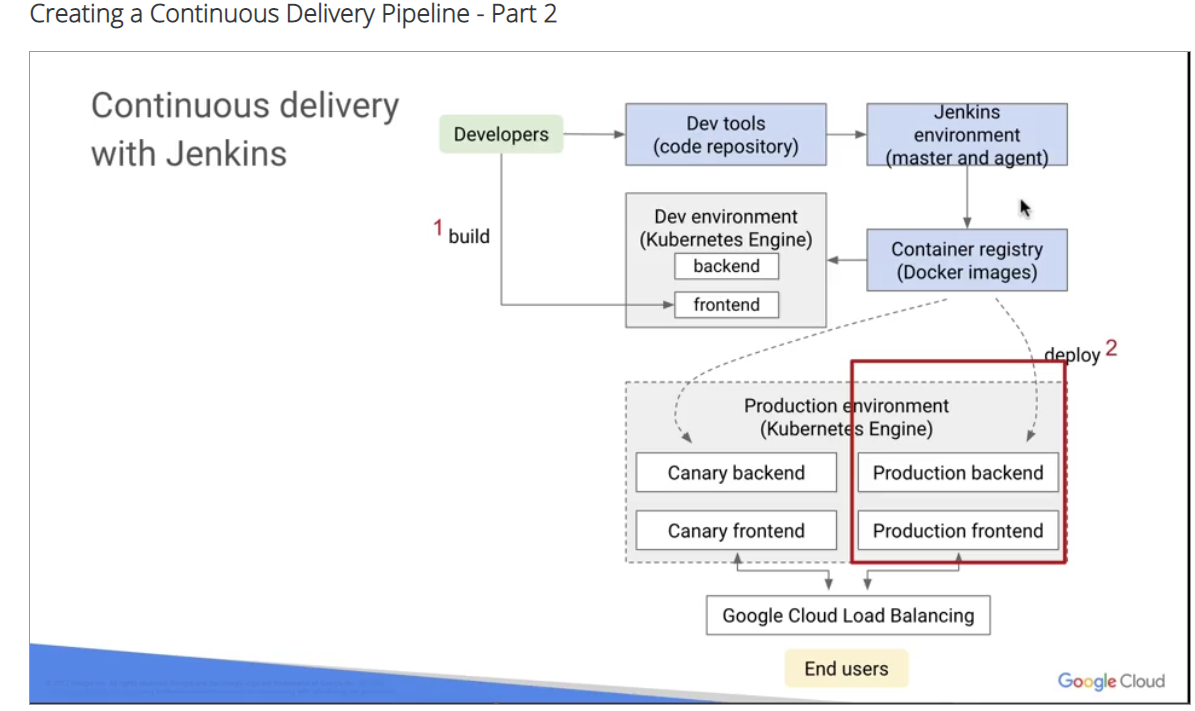


From there, developers can unit test and iterate on that code branch in an environment similar to their production environment that is not being hit by live traffic.

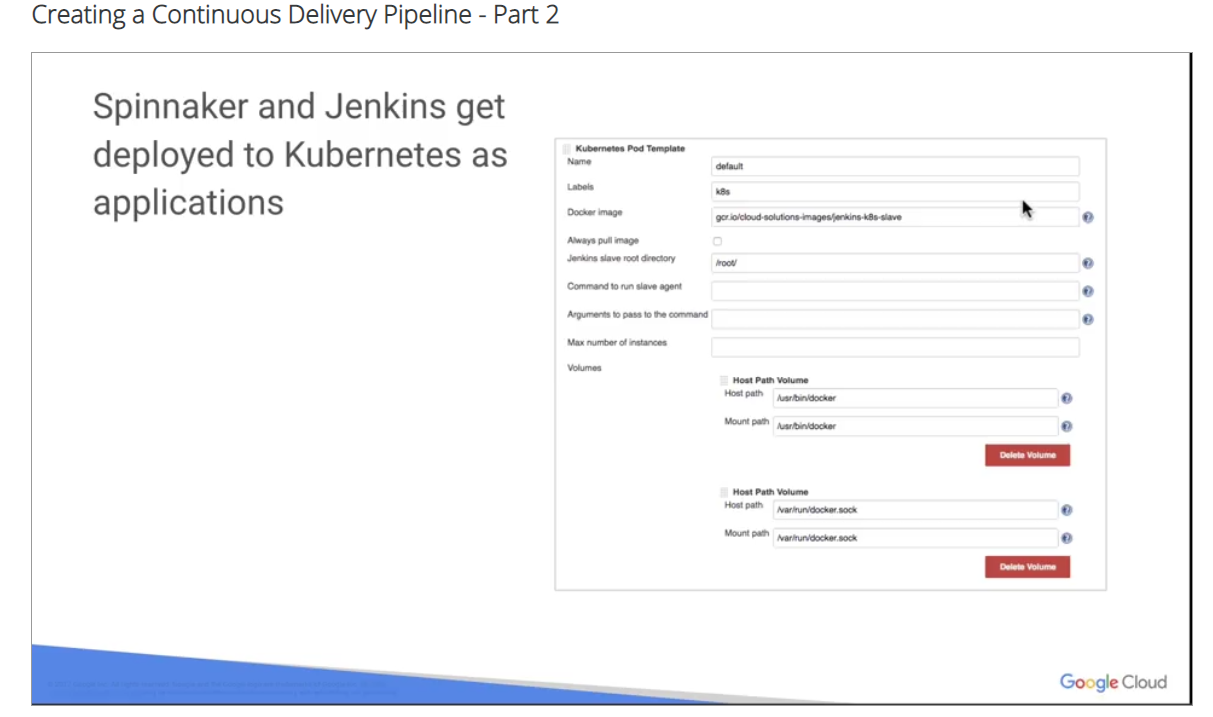
When they verify the unit code, they can commit their changes to a different branch that commit changes to a Canary deployment in production. As you saw earlier with a Canary deployment, you're only spinning up a subset of pods and repositories to a portion of live traffic.



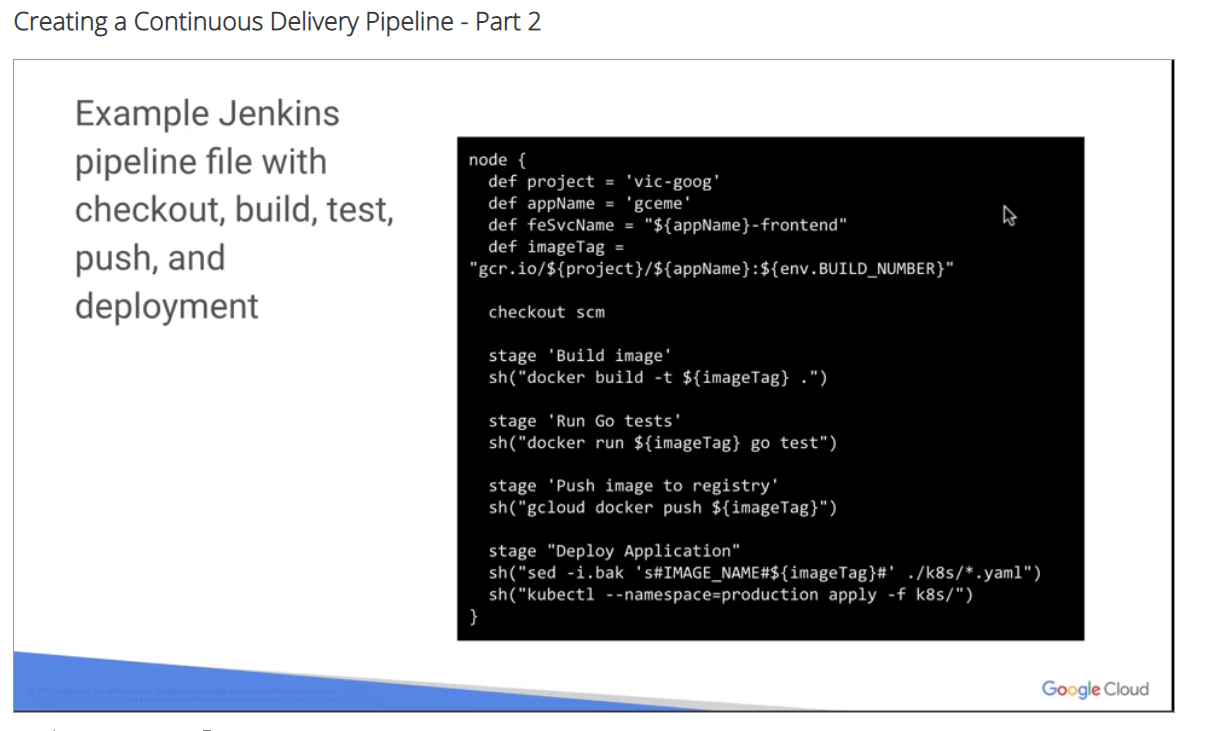
When the Canary back-end has been verified, developers merge that code to a production branch. When the changes are picked up my Jenkins, the image can be built and sent to the rest of the fleet that is serving end-users.



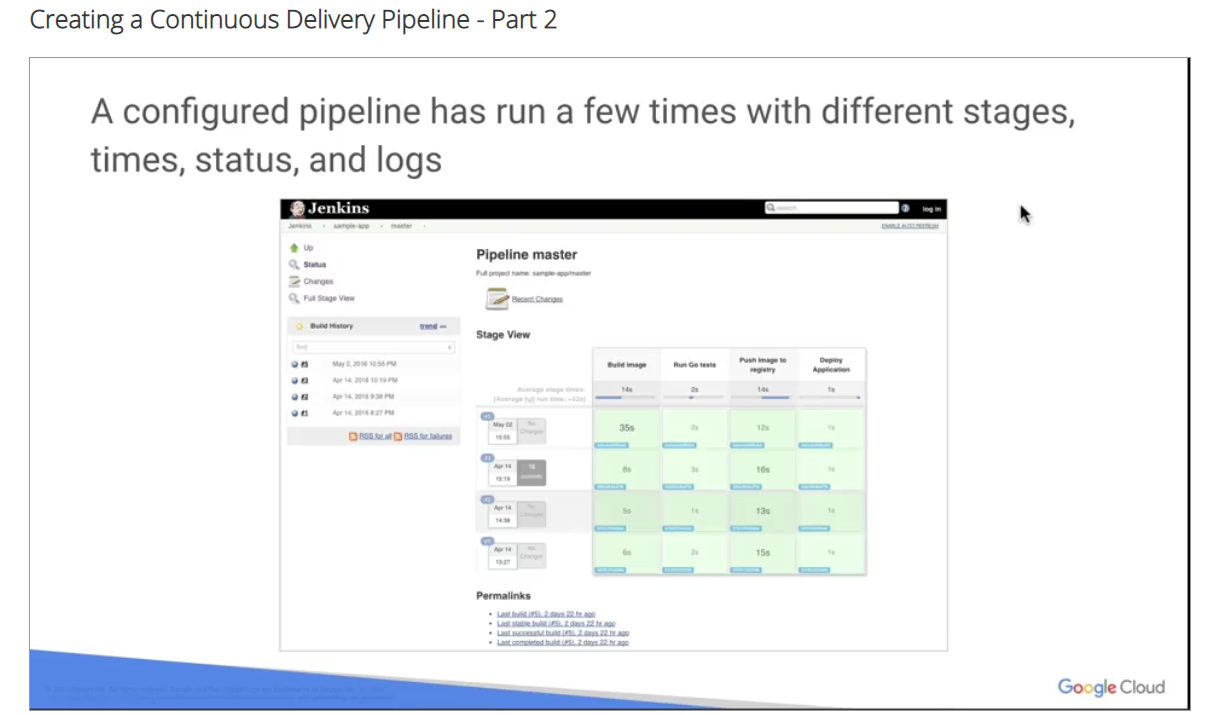
You deploy Spinnaker or Jenkins as Kubernetes applications. They are not standalone services. Here's a screenshot of the Jenkins application configuration wizard.



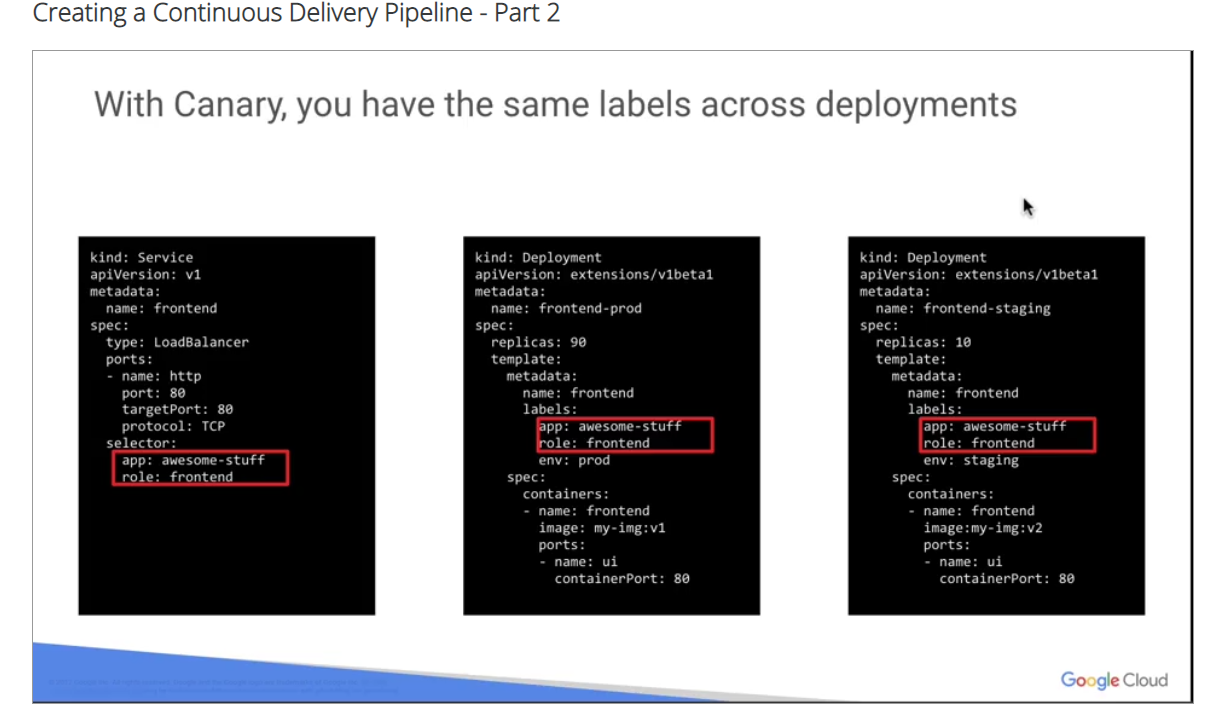
An example of a Jenkins pipeline file.



Here you check out your own 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 the image. Once the test pass, if the image push is successful, it deploys your application using kubectl which is baked into your container image.

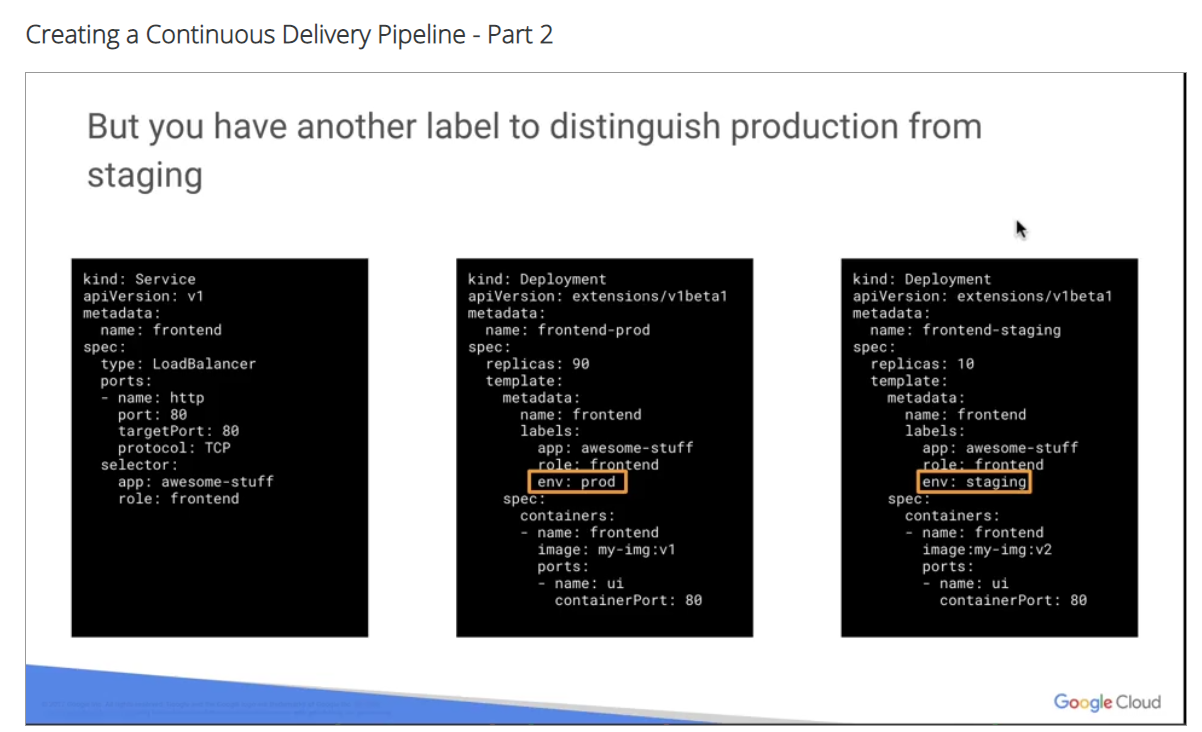


Here's what it looks like when a pipeline is configured and it's been run a few times. You can see the different stages that have been set up. It tells you how long each stage is which is interesting for figuring out where you can optimize your time for deployment. But also gives you very clear output on which stages have passed and gives you an easy way to get to your logs for each stage. You'll stage a portion of your live released to a Canary deployment for first user testing. The configuration on the left is your service.



Configuration in the middle is for your production deployment and the configuration on the right is for your Canary deployment. Notice that they all use the same labels. The only difference here for deployment and Canary is the number of replicas. In the case of the production configuration, there are 90 replicas and in the case of the Canary there are 10 replicas.

Canaries can be run at various levels of sophistication. An example of maturity progression can be found on the blog post under an article **called Introducing Kayenta an open automated Canary analysis tool from Google and Netflix**. With deploying to Canary, you use the same labels across the deployments. In this case, you use awesome stuff app label and front-end role label to the service for your front-end.



But to distinguish your production from Canary, you also have an environment label that says prod and staging. So then you can change the prod and staging capacity so that it has only 90 percent of your traffic going to production and only 10 percent of your traffic going to staging. That's how you define how much traffic goes between prod and staging for Canary deployment. Now you've seen an overview of how to set up continuous deployment and Kubernetes using Spinnaker and Jenkins. Next, you'll go through the lab that covers all the details.

