Know Everything About Spinnaker & How to Deploy Using Kubernetes Engine

Velotio Technologies Follow
Jan 2, 2019 · 14 min read

Introduction

Spinnaker is an open-source, multi-cloud continuous delivery platform that helps you release software changes with high velocity and confidence.

Open sourced by Netflix and heavily contributed to by Google, it supports all major cloud vendors (AWS, Azure, App Engine, Openstack, etc.) including Kubernetes.

In this blog I'm going to walk you through all the basic concepts in Spinnaker and help you create a continuous delivery pipeline using Kubernetes Engine, Cloud Source Repositories, Container Builder, Resource Manager, and Spinnaker. After creating a sample application, we will configure these services to automatically build, test, and

deploy it. When the application code is modified, the changes trigger the continuous delivery pipeline to automatically rebuild, retest, and redeploy the new version.

What Spinnaker Provides?

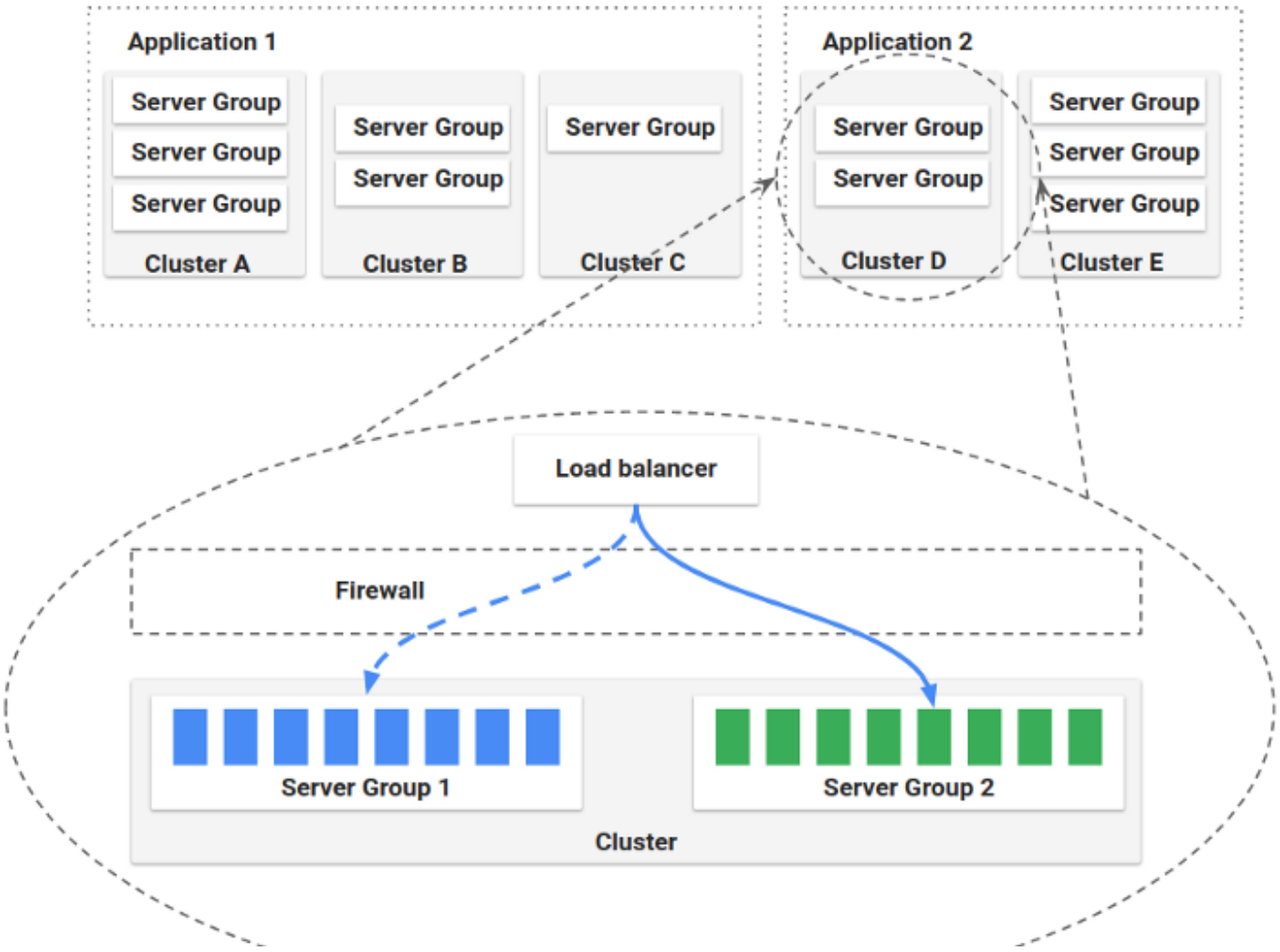
Application management and Application Deployment are its two core features.

Application Management

Spinnaker's application management features can be used to view and manage your cloud resources.

Modern tech organizations operate collections of services — sometimes referred to as "applications" or "microservices". A Spinnaker application models this concept.

Applications, Clusters, and Server Groups are the key concepts Spinnaker uses to describe services. Load balancers and Firewalls describe how services are exposed to users.



Application

- An application in Spinnaker is a collection of clusters, which in turn are collections of server groups. The application also includes firewalls and load balancers. An application represents the service which needs to be deployed using Spinnaker, all configuration for that service, and all the infrastructure on which it will run. Normally, a different application is configured for each service, though Spinnaker does not enforce that.

Cluster

- Clusters are logical groupings of Server Groups in Spinnaker.
- **Note**: Cluster, here, does not map to a Kubernetes cluster. It's merely a collection of Server Groups, irrespective of any Kubernetes clusters that might be included in your underlying architecture.

Server Group

- The base resource, the Server Group, identifies the deployable artifact (VM image, Docker image, source location) and basic configuration settings such as number of instances, autoscaling policies, metadata, etc. This resource is optionally associated with a Load Balancer and a Firewall. When deployed, a Server Group is a collection of instances of the running software (VM instances, Kubernetes pods).

Load Balancer

- A Load Balancer is associated with an ingress protocol and port range. It balances traffic among instances in its Server Groups. Optionally, health checks can be enabled for a load balancer, with flexibility to define health criteria and specify the health check endpoint.

Firewall

- A Firewall defines network traffic access. It is effectively a set of firewall rules defined by an IP range (CIDR) along with a communication protocol (e.g., TCP) and port range.

Application Deployment

Pipeline

- The *pipeline* is the key deployment management construct in Spinnaker. It consists of a sequence of actions, known as stages. You can pass parameters from stage to stage along the pipeline.
- You can start a pipeline manually, or you can configure it to be automatically triggered by an event, such as a Jenkins job completing, a new Docker image appearing in your registry, a CRON schedule, or a stage in another pipeline.
- You can configure the pipeline to emit notifications, by email, SMS or HipChat, to interested parties at various points during pipeline execution (such as on pipeline start/complete/fail).

Stage

- A *Stage* in Spinnaker is an atomic building block for a pipeline, describing an action that the pipeline will perform. You can sequence stages in a Pipeline in any order, though some stage sequences may be more common than others. Spinnaker provides a number of stages such as Deploy, Resize, Disable, Manual Judgment, and many more. The full list of stages and read about implementation details for each provider here.

Deployment Strategies

- Spinnaker supports all the cloud native deployment strategies including Red/Black (a.k.a Blue/Green), Rolling red/black and Canary deployments, etc.

What is Spinnaker Made Of?

Spinnaker is composed of a number of independent microservices:

- Deck is the browser-based UI.

- Gate is the API gateway. The Spinnaker UI and all API callers communicate with Spinnaker via Gate.
- Orca is the orchestration engine. It handles all ad-hoc operations and pipelines.
- Clouddriver is responsible for all mutating calls to the cloud providers and for indexing/caching all deployed resources.
- Front50 is used to persist the metadata of applications, pipelines, projects and notifications.
- Rosco is the bakery. It is used to produce machine images (for example GCE images, AWS AMIs, Azure VM images). It currently wraps Packer, but will be expanded to support additional mechanisms for producing images.
- Igor is used to trigger pipelines via continuous integration jobs in systems like Jenkins and Travis CI, and it allows Jenkins/Travis stages to be used in pipelines.
- Echo is Spinnaker's eventing bus. It supports sending notifications (e.g. Slack, email, Hipchat, SMS), and acts on incoming webhooks from services like GitHub.
- Fiat is Spinnaker's authorization service. It is used to query a user's access permissions for accounts, applications and service accounts.
- Kayenta provides automated canary analysis for Spinnaker.
- Halyard is Spinnaker's configuration service. Halyard manages the lifecycle of each of the above services. It only interacts with these services during Spinnaker start-up, updates, and rollbacks.

By default, Spinnaker binds ports accordingly for all the above-mentioned microservices. For us, the UI (Deck) will be exposed onto Port 9000.

What are We Going to Do?

- Set up your environment by launching Cloud Shell, creating a Kubernetes Engine cluster, and configuring your identity and user management scheme.
- Download a sample application, create a Git repository, and upload it to a Cloud Source Repository.

- Deploy Spinnaker to Kubernetes Engine using Helm.
- Build a Docker image from the source code.
- Create triggers to create Docker images when the source code for application changes.
- Configure a Spinnaker pipeline to reliably and continuously deploy your application to Kubernetes Engine.
- Deploy a code change, triggering the pipeline, and watch it roll out to production.

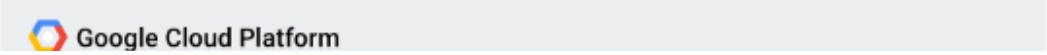
Note: This blog post uses various billable components in GCP like GKE, Container Builder etc.

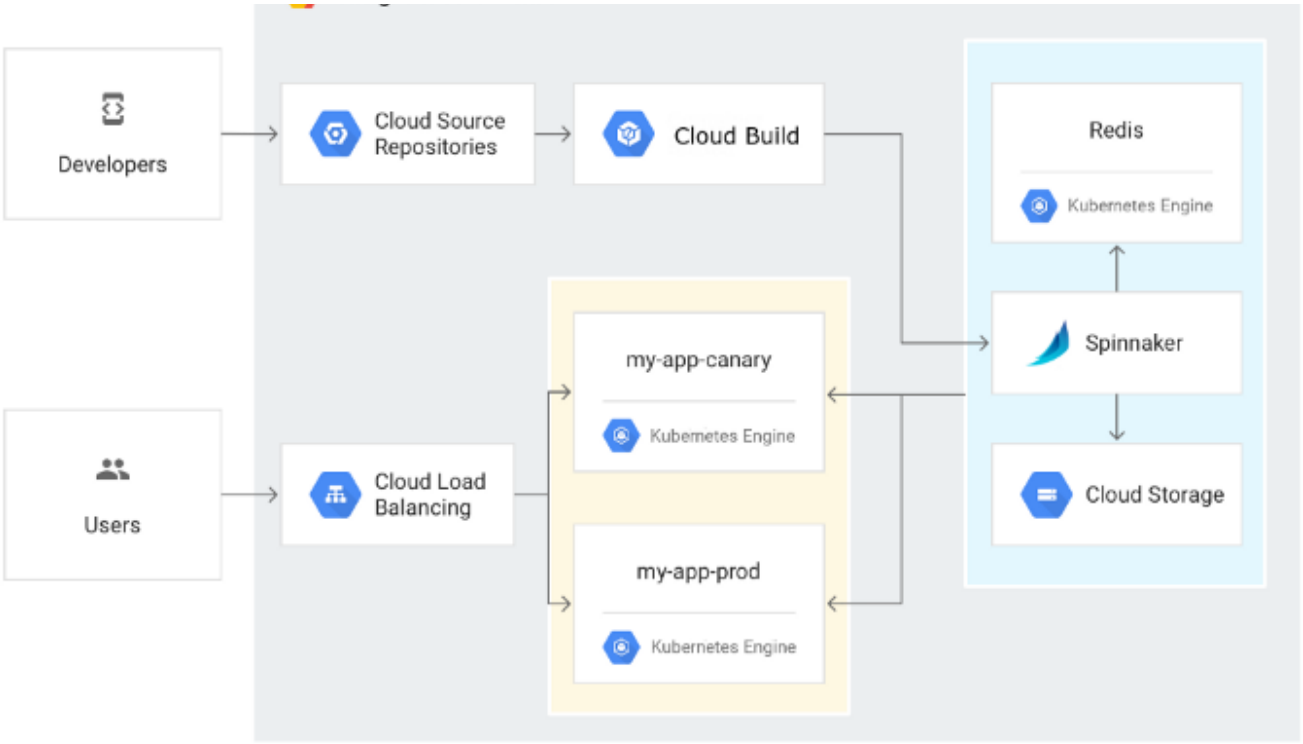
Pipeline Architecture

To continuously deliver application updates to users, companies need an automated process that reliably builds, tests, and updates their software. Code changes should automatically flow through a pipeline that includes artifact creation, unit testing, functional testing, and production rollout. In some cases, they want a code update to apply to only a subset of their users, so that it is exercised realistically before pushing it to entire user base. If one of these canary releases proves unsatisfactory, the automated procedure must be able to quickly roll back the software changes.

With Kubernetes Engine and Spinnaker, we can create a robust continuous delivery flow that helps us to ensure that software is shipped as quickly as it is developed and validated. Although rapid iteration is the end goal, we must first ensure that each application revision passes through a series of automated validations before becoming a candidate for production rollout. When a given change has been vetted through automation, we can also validate the application manually and conduct further pre-release testing.

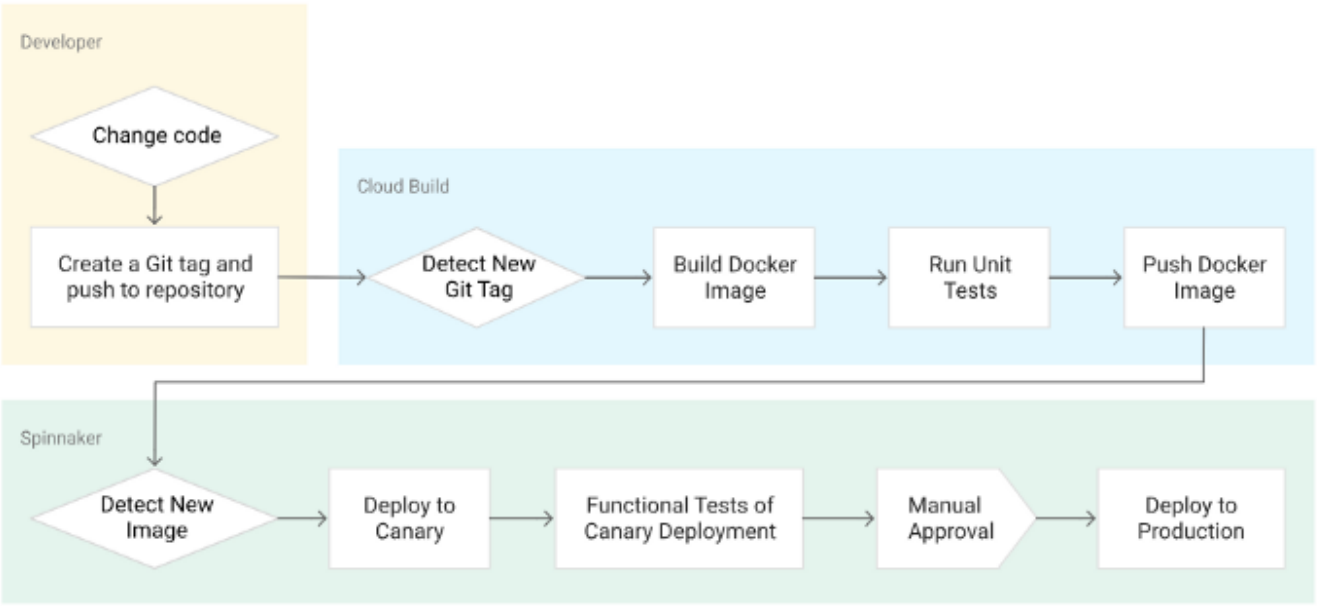
After the team decides the application is ready for production, one of the team members can approve it for production deployment.





Application Delivery Pipeline

We are going to build the continuous delivery pipeline shown in the following diagram.



Prerequisites

A fair bit of experience in GCP services like:

- GKE (Google Kubernetes Engine)
- Google Compute
- Google APIs
- Cloud Source Repository
- Container Builder
- Cloud Storage
- Cloud Load Balancing
- Knowledge in K8s terminology like Services, Deployments, Pods, etc
- Familiarity with Kubectl and Helm package manager

Before Starting just enable the APIs needed on GCP

- Kubernetes API
- Compute API
- Resource Manger API
- IAM API

Set Up a Kubernetes Cluster

1. Go to the Console and scroll the left panel down to **Compute->Kubernetes Engine->Kubernetes Clusters**.
2. Click **Create Cluster.**
3. Choose a name or leave as the default one.
4. Under **Machine Type**, click **Customize**.
5. Allocate at least **2 vCPU** and **10GB of RAM**.
6. Change the cluster size to **2**.
7. **Enable Legacy Authorization** while customizing the cluster.

8. Keep the rest of the defaults and click **Create**.

In a minute or two the cluster will be created and ready to go.

Configure identity and access management

Create a Cloud Identity and Access Management (Cloud IAM) service account to delegate permissions to Spinnaker, allowing it to store data in Cloud Storage. Spinnaker stores its pipeline data in Cloud Storage to ensure reliability and resiliency. If our Spinnaker deployment unexpectedly fails, we can create an identical deployment in minutes with access to the same pipeline data as the original.

1. Create a service account:

```
$ gcloud iam service-accounts create spinnaker-storage-account \ --display-name spinnaker-storage
```

create-service.js hosted with ♡ by GitHub view raw

2. Store the service account email address and our current project ID in environment variables for use in later commands:

```
$ export SA_EMAIL=$(gcloud iam service-accounts list \ --filter="displayName:spinnaker-storage-ac
$ export PROJECT=$(gcloud info --format='value(config.project)')
```

store_service.js hosted with ♡ by GitHub view raw

3. Bind the **storage.admin** role to our service account:

```
$ gcloud projects add-iam-policy-binding \ $PROJECT --role roles/storage.admin --member serviceAc
```

storage_admin.js hosted with ♡ by GitHub view raw

4. Download the service account key. We will need this key later while installing Spinnaker and we need to also upload the key to Kubernetes Engine.

```
$ gcloud iam service-accounts keys create spinnaker-sa.json --iam-account $SA_EMAIL
```

download_service.js hosted with ♡ by GitHub view raw

Deploying Spinnaker using Helm

In this section, we will deploy Spinnaker onto the K8s cluster via Charts with the help of K8s package manager Helm. Helm has made it very easy to deploy Spinnaker, it can be a very painful act to deploy it manually via Halyard and configure it.

Install Helm

1. Download and install the helm binary:

```
$ wget https://storage.googleapis.com/kubernetes-helm/helm-v2.9.0-linux-amd64.tar.gz
```

download_helm.js hosted with ♡ by GitHub view raw

2. Unzip the file to your local system:

```
$ tar zxfv helm-v2.9.0-linux-amd64.tar.gz$ sudo chmod +x linux-amd64/helm && sudo mv linux-amd64/
```

unzip_file.js hosted with ♡ by GitHub view raw

3. Grant Tiller, the server side of Helm, the cluster-admin role in your cluster:

```
$ kubectl create clusterrolebinding user-admin-binding \ --clusterrole=cluster-admin --user=$(gcl
$ kubectl create serviceaccount tiller --namespace kube-system
$ kubectl create clusterrolebinding tiller-admin-binding \ --clusterrole=cluster-admin --servicea
```

grant_tiller.js hosted with ♡ by GitHub view raw

4. Grant Spinnaker the **cluster-admin** role so it can deploy resources across all namespaces:

```
$ kubectl create clusterrolebinding --clusterrole=cluster-admin       --serviceaccount=default:de
```

grant_spinnaker.js hosted with ♡ by GitHub view raw

5. Initialize Helm to install Tiller in your cluster:

```
$ helm init --service-account=tiller --upgrade
$ helm repo update
```

initialize.js hosted with ♡ by GitHub view raw

6. Ensure that Helm is properly installed by running the following command. If Helm is correctly installed, **v2.9.0** appears for both client and server.

```
$ helm version
```

verify_helm.js hosted with ♡ by GitHub view raw

Configure Spinnaker

1. Create a bucket for Spinnaker to store its pipeline configuration:

```
$ export PROJECT=$(gcloud info --format='value(config.project)')
$ export BUCKET=$PROJECT-spinnaker-configgsutil mb -c regional -l us-central1 \ gs://$BUCKET
```

create_bucket.js hosted with ♡ by GitHub view raw

2. Create the configuration file:

```
$ export SA_JSON=$(cat spinnaker-sa.json)
$ export PROJECT=$(gcloud info --format='value(config.project)')
$ export BUCKET=$PROJECT-spinnaker-config
$ cat > spinnaker-config.yaml <
```

export.js hosted with ♡ by GitHub view raw

```
# Disable minio as the defaultminio:
enabled: false

# Configure your Docker registries here accounts:
name: gcr
address: https://gcr.io
username: _json_key
password: '$SA_JSON'
email: 1234@5678.com EOF
```

configuration.js hosted with ♡ by GitHub view raw

Deploy the Spinnaker chart

1. Use the Helm command-line interface to deploy the chart with the configuration set earlier. This command typically takes five to ten minutes to complete, so we will be providing a deploy timeout with ` — timeout`

```
$ helm install -n cd stable/spinnaker -f spinnaker-config.yaml --timeout \ 600 --version 0.3.1
```

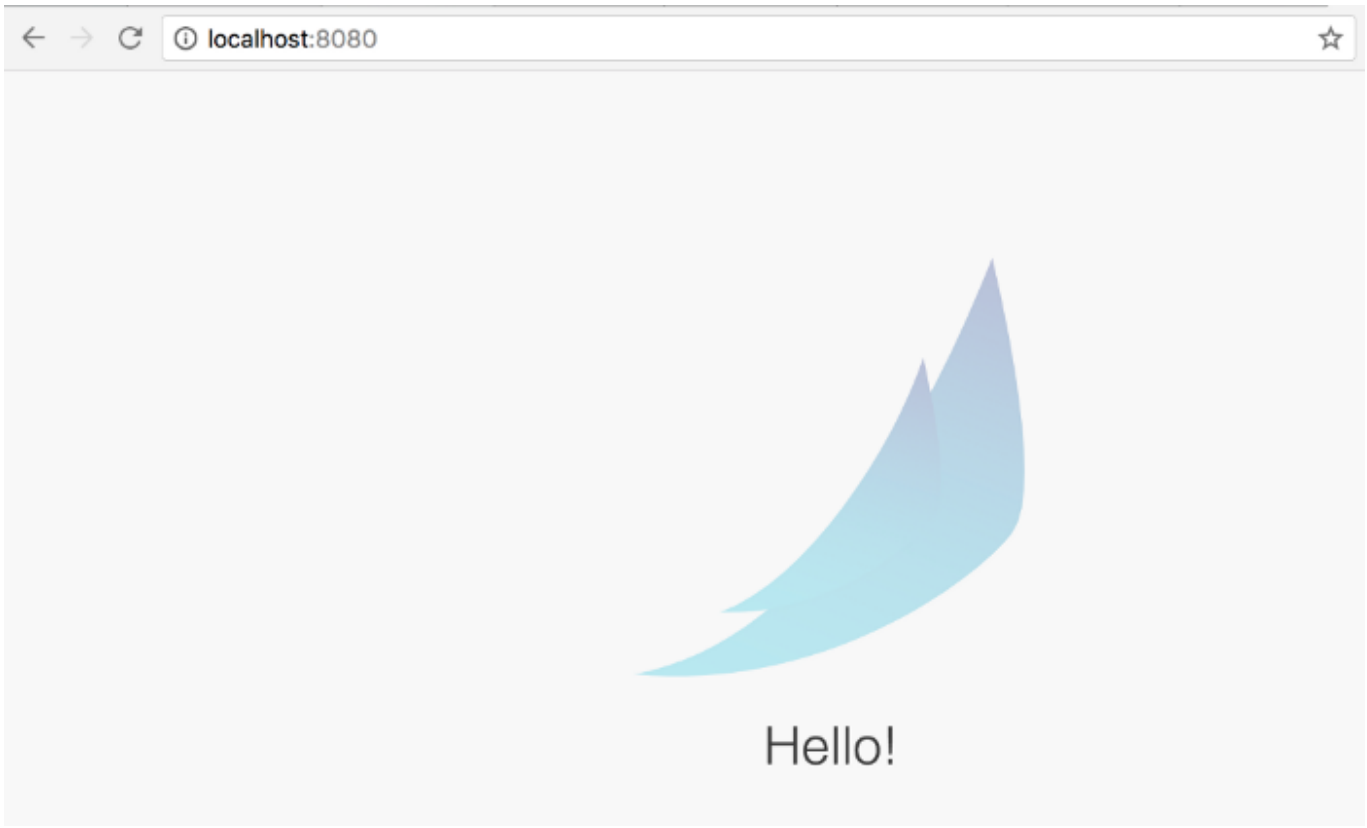
spinnaker_config.js hosted with ♡ by GitHub view raw

2. After the command completes, run the following command to set up port forwarding to the Spinnaker UI from Cloud Shell:

```
$ export DECK_POD=$(kubectl get pods --namespace default -l \ "component=deck" -o jsonpath="{.ite
$ kubectl port-forward --namespace default $DECK_POD 8080:9000 \ >> /dev/null &
```

component_deck.js hosted with ♡ by GitHub view raw

3. The above command exposes the Spinnaker UI onto the local machine that we're using to run all the commands. We can use any port of our choosing instead of 8080 in the above command. Now the UI can be opened onto the URL http://localhost:8080.



Building the Docker image

In this section, we will configure **Container Builder** to detect changes to the application source code, if yes then build a Docker image, and then push it to **Container Registry**.

For this step we will use a sample app provided by the Google community

Create your source code repository

1. Download the source code:

```
$ wget https://gke-spinnaker.storage.googleapis.com/sample-app.tgz
```

sample_app.js hosted with ♡ by GitHub view raw

2. Unpack the source code:

```
$ tar xzfv sample-app.tgz
```

unpack.js hosted with ♡ by GitHub view raw

3. Change directories to source code:

```
$ cd sample-app
```

change_directory.js hosted with ♡ by GitHub view raw

4. Set the username and email address for Git commits in this repository. Replace [EMAIL_ADDRESS] with Git email address, and replace [USERNAME] with Git username.

```
$ git config --global user.email "[EMAIL_ADDRESS]"
$ git config --global user.name "[USERNAME]"
```

git.js hosted with ♡ by GitHub view raw

5. Make the initial commit to source code repository:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

commit.js hosted with ♡ by GitHub view raw

6. Create a repository to host the code:

```
$ gcloud source repos create sample-app
$ git config credential.helper gcloud.sh
```

helper.js hosted with ♡ by GitHub view raw

7. Add our newly created repository as remote:

```
$ export PROJECT=$(gcloud info --format='value(config.project)')
$ git remote add origin \ https://source.developers.google.com/p/$PROJECT/r/sample-app
```

remote.js hosted with ♡ by GitHub view raw

8. Push the code to the new repository's master branch:

9. Check that we can see our source code in the console:

Configuring the build triggers

In this section, we configure **Google Container Builder** to build and push your Docker images every time we push Git tags to our source repository. Container Builder automatically checks out the source code, builds the Docker image from the Dockerfile in the repository, and pushes that image to Container Registry.

1. In the GCP Console, click **Build Triggers** in the Container Registry section.
2. Select **Cloud Source Repository** and click **Continue**.
3. Select your newly created sample-app repository from the list, and click **Continue**.
4. Set the following trigger settings:

Name:sample-app-tags

Trigger type: Tag

Tag (regex): v.*

Build configuration: cloudbuild.yaml

cloudbuild.yaml location: /cloudbuild.yaml

5. Click **Create trigger**.

From now on, whenever we push a Git tag prefixed with the letter **"v"** to source code repository, Container Builder automatically builds and pushes our application as a

Docker image to Container Registry.

Let's build our first image:

Push the first image using the following steps:

1. Go to source code folder in Cloud Shell.

2. Create a Git tag:

3. Push the tag:

4. In **Container Registry**, click **Build History** to check that the build has been triggered. If not, verify the trigger was configured properly in the previous section.

Configuring your deployment pipelines

Now that our images are building automatically, we need to deploy them to the Kubernetes cluster.

We deploy to a scaled-down environment for integration testing. After the integration tests pass, we must manually approve the changes to deploy the code to production services.

Create the application

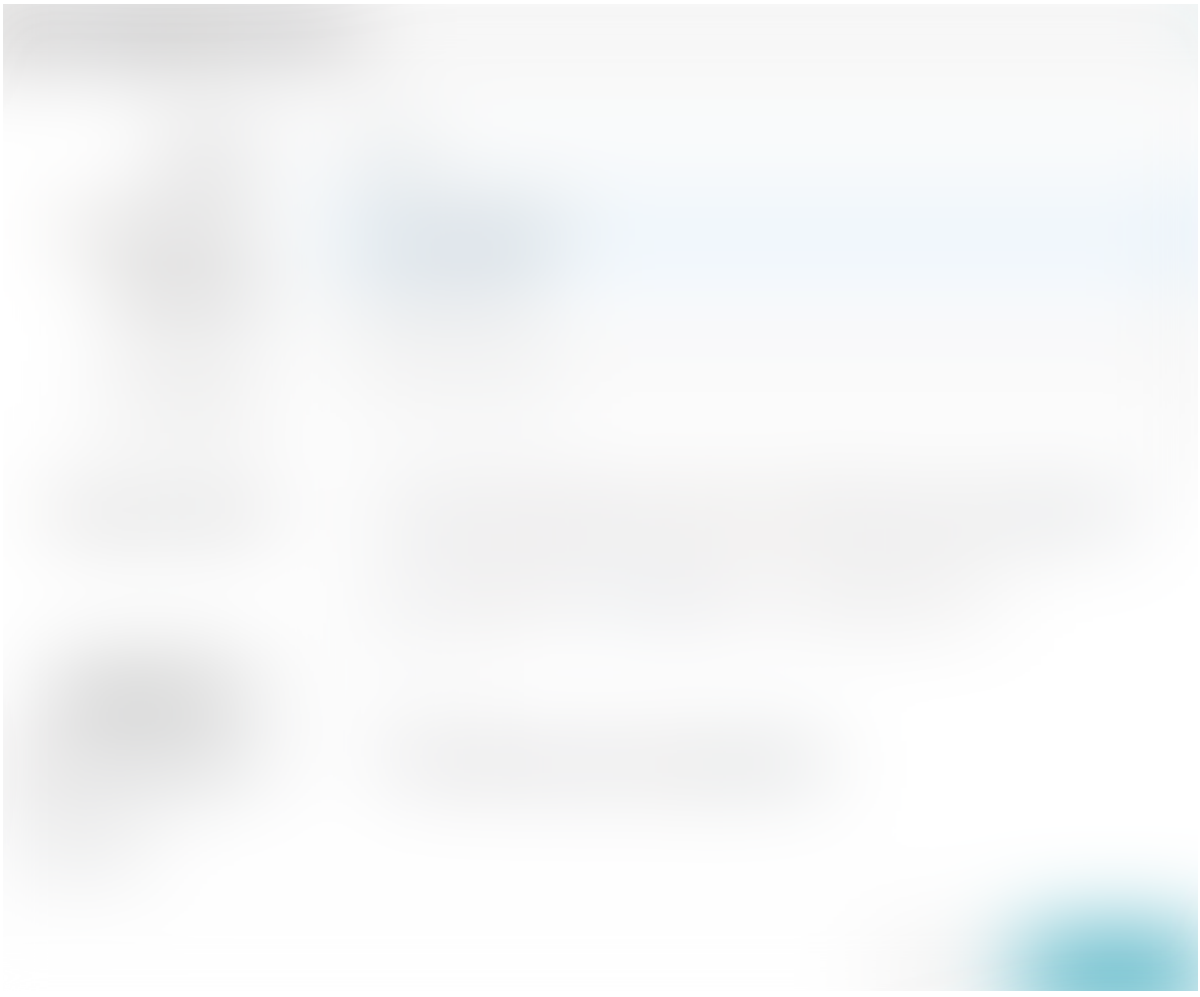
1. In the Spinnaker UI, click **Actions**, then click **Create Application**.

2. In the **New Application** dialog, enter the following fields:

Name: sample

Owner Email: [your email address]

3. Click **Create.**



Create service load balancers

To avoid having to enter the information manually in the UI, use the Kubernetes command-line interface to create load balancers for the services. Alternatively, we can perform this operation in the Spinnaker UI.

On the local machine where the code resides, run the following command from the sample-app root directory:

Create the deployment pipeline

Now we create the continuous delivery pipeline. The pipeline is configured to detect when a Docker image with a tag prefixed with "v" has arrived in your Container Registry.

1. Create a new pipeline named say "**Deploy**".

2. Go to the **Config** page for the pipeline that we just created and click **Pipeline Actions -> Edit as JSON.**

3. Change the directory to the source code directory and update the current pipeline-deploy.json at path **spinnaker/pipeline-deploy.json** according to our needs.

4. Now in the JSON editor just copy the whole file **spinnaker/updated-pipeline-deploy.json.**

5. Click on **Update Pipeline** and we should have an updated pipeline config now.

6. In the Spinnaker UI, click **Pipelines** on the top navigation bar.

7. Click **Configure** in the Deploy pipeline.

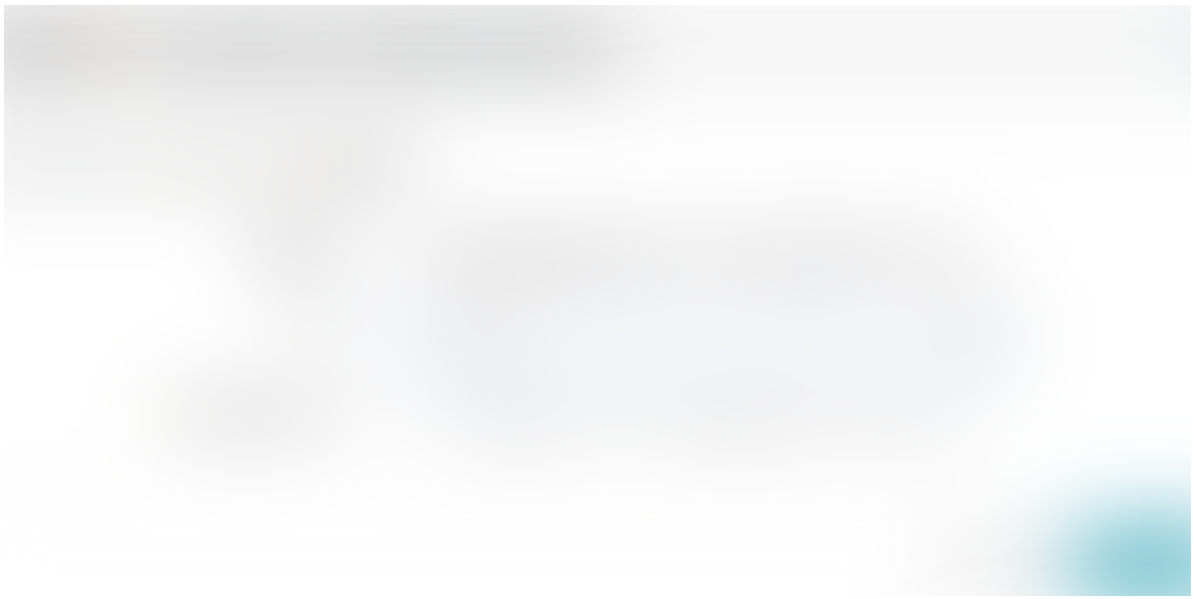
8. The continuous delivery pipeline configuration appears in the UI:

Running the pipeline manually

The configuration we just created contains a trigger to start the pipeline when a new Git tag containing the prefix **"v"** is pushed. Now we test the pipeline by running it manually.

1. Return to the Pipelines page by clicking **Pipelines**.
2. Click **Start Manual Execution**.

3. Select the v1.0.0 tag from the **Tag** drop-down list, then click **Run**.



4. After the pipeline starts, click **Details** to see more information about the build's progress. This section shows the status of the deployment pipeline and its steps. Steps in

blue are currently running, green ones have completed successfully, and red ones have failed. Click a stage to see details about it.

5. After 3 to 5 minutes the integration test phase completes and the pipeline requires manual approval to continue the deployment.

6. Hover over the yellow **"person"** icon and click **Continue**.

7. Your rollout continues to the production frontend and backend deployments. It completes after a few minutes.

8. To view the app, click **Load Balancers** in the top right of the Spinnaker UI.

9. Scroll down the list of load balancers and click **Default**, under **sample-frontend-prod**.

10. Scroll down the details pane on the right and copy application's IP address by clicking the clipboard button on the **Ingress** IP.

11. Paste the address into the browser to view the production version of the application.

12. We have now manually triggered the pipeline to build, test, and deploy your application.

Triggering the pipeline automatically via code changes

Now let's test the pipeline end to end by making a code change, pushing a Git tag, and watching the pipeline run in response. By pushing a Git tag that starts with **"v"**, we trigger **Container Builder** to build a new Docker image and push it to **Container Registry**. Spinnaker detects that the new image tag begins with "v" and triggers a pipeline to deploy the image to canaries, run tests, and roll out the same image to all pods in the deployment.

1. Change the color of the app from orange to blue:

2. Tag your change and push it to the source code repository:

3. See the new build appear in the Container Builder Build History.

4. Click **Pipelines** to watch the pipeline start to deploy the image.

5. Observe the canary deployments. When the deployment is paused, waiting to roll out to production, start refreshing the tab that contains our application. Nine of our backends are running the previous version of your application, while only one backend is running the canary. **Now we should see the new, blue version of our application appear about every tenth time we refresh**.

6. After testing completes, return to the **Spinnaker** tab and approve the deployment.

7. When the pipeline completes, application looks like the following screenshot. Note that the colour has changed to blue because of code change, and that the **Version** field now reads v1.0.1.

8. We have now successfully rolled out your application to your entire production environment!!!!!!

9. Optionally, we can roll back this change by reverting the previous commit. Rolling back adds a new tag (v1.0.2), and pushes the tag back through the same pipeline we used to deploy v1.0.1:

Conclusion

Now that you know how to get Spinnaker up and running in a development environment, start using it already. In this blog, we have done everything from installing a K8s cluster on GCP to deploying an End to End Pipeline just like that in a production environment. Hope you found it helpful. Do let us know in case you have any queries or suggestions in the comments below.

References

https://cloud.google.com/solutions/continuous-delivery-spinnaker-kubernetes-engine

****

*This post was originally published on **Velotio Blog.***

***Velotio Technologies** is a software engineering firm, with core expertise in Data Science, Machine Learning and DevOps. Our modus operandi is working with the latest transformative tech to turbocharge customer success.*

*Interested in learning more about us? We would love to connect with you on our**Website**, **LinkedIn** or **Twitter**.*

****

Docker Spinnaker Kubernetes