DEPLOY CI/CD PIPELINE WITH POLICY CONFIGURATION

Step by Step Guide

By Navveen Balani

Follow/Connect - https://www.linkedin.com/in/naveenbalani/

DEPLOY CI/CD WITH POLICY CONFIGURATION

INTRODUCTION

Continuous Integration and Continuous Delivery (CI/CD) is a practice or a process that consists of a set of tools and tasks to enable you to automate the process of building, testing, and deploying your applications in a repeatable manner across environments. CI/CD allows you to roll out releases of your application rapidly. At the same time, it is being developed and maintained by teams that work in tandem to deliver reliable software across environments.

In this paper, we will go through the details of how to build an automated CI/CD process with Policy configuration using a step-by-step approach.

In a traditional Information Technology (IT) landscape, you will often find two distinct teams operating independently: development and operations. The lack of coordination or collaboration among them led to a new practice called DevOps. DevOps is a culture,

methodology, and tools that bridge the gap between development and operations teams. It has become an essential ingredient of modern-day application development environment and process. The modern-day software development teams are a bunch of agile DevOps engineers.

When you consider modernizing your application platform, you think of building your software as containers and releasing them through a CI/CD process. A DevOps team can have resources functioning as developers and operators. While looking through the lens of DevOps, a developer develops, tests, and builds the application into containers as deployable units and operators take the deployment artifacts, apply configurations and security policies and perform deployments to the target runtime.

The security function can be clubbed with operations or carved into a separate dedicated role responsible for ensuring the security and compliance of the application and the deployment environment.

CI/CD PROCESS WORKFLOW WITH POLICY CONFIGURATION

Let's look at the workflow for CI/CD process.

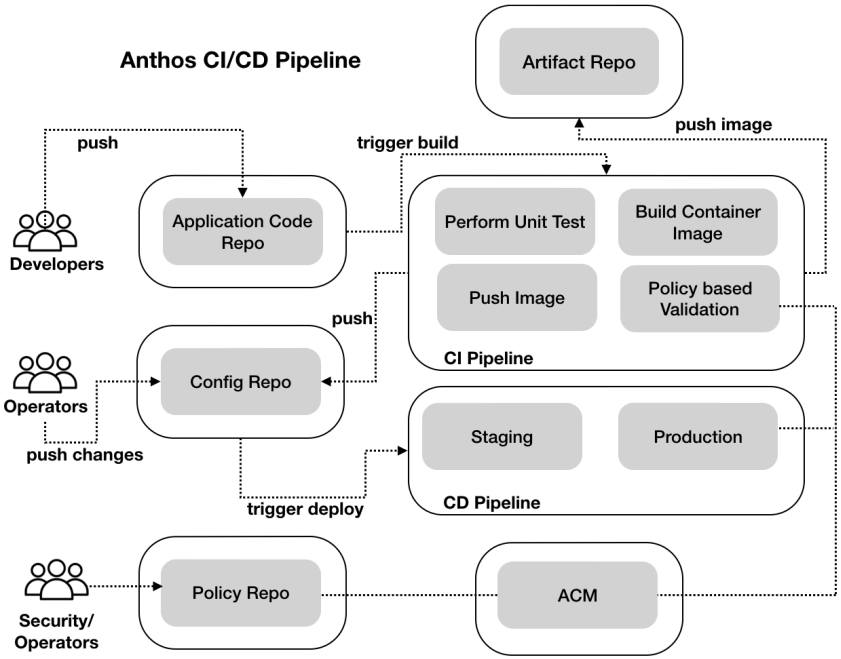


Figure 1 - CI/CD Workflow Processs

We follow a GitOps model to implement CI/CD processes. All the application code, configuration files like Kubernetes manifests, security policy manifests, and customization templates are stored in a Git source repository. The infrastructure setup is also configured as code or declarative constructs and stored in the source repository. The Git repository is a single source of truth that provides implicit built-in auditability for your application code, configurations, and infrastructure code and a basis to propagate changes and drive deployment operations, governance, and observability. This concept of Git serving as an operating model to support the software delivery lifecycle is called GitOps.

At a high level, the process consists of the following -

- An application is broken down into a set of modules. Developers work on the respective modules and check their code in the Git source repository.

- The CI tools like Cloud Build or Jenkins are configured to listen to any changes in the Git source repository. The changes could be a code commit to a particular branch, tagging a release, etc. It then tests the code, builds the container image, and stores it in Google Container Registry or Artifact Registry.

- Once the container image is pushed to the container registry, the config files are updated with the new

image URL. The operations team also uses Git to store all the configurations like Kubernetes manifests, security policy manifests, and infrastructure code like terraform scripts. The declarative code is usually templatized using tools and best practices which are then applied for deployment across environments.

- The template files provide placeholders for injecting property values at runtime (like replicas : 2 for development and 5 for production). Tools like Kustomize or Helm can create templates for your manifests without changing the original base manifest (YAML) files. Anthos provides integration with various open-source packaging and customization tools implemented as part of best practices.

- The changes from earlier steps are reviewed, approved, and then merged into the respective branch in that environment. Once the code is merged, the CD tools like Cloud Build pick up the changes from that branch and apply the configuration for deployment to the respective GKE environment.

- The operations or security team stores the governance or security policy configurations in Git. The ACM component of Anthos can be used to enforce these policies to multiple GKE clusters across environments.

- Once the application is up and running, you can leverage ASM to gain more visibility into your services and benchmark its performances by defining Service

Level Objective (SLO) in alignment with your application requirements. We had gone through the SLO concepts in the ASM chapter. For more details on SLO, refer to the Google SRE handbook at https://sre.google/books/.

The above process briefly outlines the workflow required to execute your CI/CD process across environments. In the next section, you will realize the workflow by setting up and running the end-to-end CI/CD process and using Anthos Anthos Config Management for Policy management.

We will take a two-step approach to demonstrate the CI/CD use case. We will first perform the CI/CD process by manually testing the build and deploying scripts, which will help you understand the flow.

Once all the manual steps are working, the second step will be to automate the flow using triggers and execute the CI/CD process end-to-end.

CI/CD REPOSITORY SETUP

In this section, you will set up source repositories that will form the basis for the CI/CD process. The setup will consist of three source repositories: application, Config, and Policy.

- **Application repository**: This repository contains the application source code, Docker file, and build scripts for creating the container images and performing policy validation. The repository is located at - https://github.com/cloudsolutions-academy/anthos-demo-app

- **Config repository**: This repository contains the Kubernetes manifests template created using tools like Kustomize. The template files are realized into specific configurations based on the environment (like staging or production) and placed in a deployment directory of your choice as part of this repository. The directory name could be as simple as `configs/` or the name could be based on the environment like `staging/` or `production/`. The repository also contains deployment scripts that can apply the final configuration to the GKE clusters as part of the deployment.

- **Policy repository**: The policy repository contains declarative programmable governance policies against which the cluster is validated for compliance. You will use ACM based constraints and constraint templates as part of policy configurations.

For our CI/CD setup, you will use Google Cloud Source Repositories as a source repository, Cloud Container Registry for storing the container images, Cloud Build for continuous integration and deployment and ACM for

creating and validating governance-based policy for cluster compliance.

Prerequisites

The setup assumes you have the following prerequisites in place

- Anthos cluster with ACM and Config Sync enabled
- Git client tool

For setting up Anthos, please refer to Chapter 2 - Anthos Installation.

If not already enabled, go to the command/terminal window and enable the following Google Cloud service APIs for your google cloud project.

```
gcloud services enable
container.googleapis.com
cloudbuild.googleapis.com
sourcerepo.googleapis.com
containeranalysis.googleapis.com
```

Create and configure repositories in Cloud Source Repositories

Configure Git to use your email address and name.

```
git config --global user.email
"YOUR_EMAIL_ADDRESS"
git config --global user.name “YOUR_NAME”
```

Please ensure the above user has the admin role to create and update the repository. For more details on required parts, you can refer to *https://cloud.google.com/source-repositories/docs/configure-access-control*

You will set up two source repositories - Application and Configuration.

- Create a top level directory inside your home directory

```
mkdir anthos-demo
cd anthos-demo
```

- To get started quickly, we have made the complete source code for the application and configuration available in *cloudsolutions-academy* GitHub repository. This will be the same code you will emulate while working with your own repository in Google Cloud. You can execute the following command to get the application source code from our *cloudsolutions-academy* GitHub repository.

```
git clone https://github.com/
cloudsolutions-academy/anthos-demo-app \
```

```
demo-app
```

- Execute the following command to get the configuration source files from our cloudsolutions-academy GitHub repository.

```
git clone https://github.com/
cloudsolutions-academy/anthos-demo-config
\
demo-config
```

- Execute the following command to create the repository on Google Cloud.

```
gcloud source repos create demo-app
```

```
gcloud source repos create demo-config
```

- Configure the above two created repositories as remote.

```
cd ~/anthos-demo/demo-app
```

```
PROJECT_ID=$(gcloud config get-value
project)
```

```
git remote add google \
    https://source.developers.google.com/
p/${PROJECT_ID}/r/demo-app
```

```
cd ~/anthos-demo/demo-config

PROJECT_ID=$(gcloud config get-value
project)

git remote add google-conf \
    https://source.developers.google.com/
p/${PROJECT_ID}/r/demo-config
```

Sample Application

The sample code added to the demo-app repository is a simple Node.js application consisting of three endpoints. `/echo`, `/healthz` and `/fetchWebsite`. The *echo* endpoint prints the message, the *healthz* endpoint checks application health and prints the ok message; if it is working fine, and the *fetchWebsite* endpoint tests inbound and outbound connectivity to/from the GKE cluster by visiting a public website.

To view the code, change the directory to *demo-app* and open the *server.js* file.

```
'use strict';

const express = require('express');
const request = require('request');
const bodyParser = require('body-parser');
const {Buffer} = require('safe-buffer');
```

```
const app = express();

app.set('case sensitive routing', true);
app.use(bodyParser.json());

const PORT = 8080;

app.post('/echo', (req, res) => {
  res
    .status(200)
    .json({message: req.body.message})
    .end();
});


app.get('/healthz', (req, res) => {
console.log(req.connection.remoteAddress);
res
    .status(200)
    .json({message: "ok"})
    .end();
});

app.get('/fetchWebsite', (req, res) => {

  request('https://navveenbalani.dev/',
function (error, response, html) {
  if (!error && response.statusCode == 200) {
    res
```

```
    .status(200)
    .json({message: "ok"})
    .end();
  } else {
    res
    .status(500)
    .json({message: error})
    .end(); }})});

  app.listen(PORT);
console.log(`Running on ${PORT}`);
```

The following shows the docker file for the application. It downloads the base node docker image, builds the application code, and exposes the application on port *8080*.

```
FROM node:12
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "npm", "start" ]
```

In the next section, you will kick-start the process of executing the CI/CD process.

DEVELOPING THE CI/CD PROCESS

In this section, as part of the first step, you will use the manual approach to build and deploy the sample application using the Cloud Build scripts. This will also test the script is performing as expected. Later, as part of the *Setting up CI/CD Process* section, we will automate the CI/CD process through these scripts using triggers.

Creating the Integration Pipeline

You will create the container image for the above application using Cloud Build and store the same in the Container Registry as part of the integration pipeline.

The following shows the listing of Cloud Build file: *app-build-trigger-ci.yaml* located in the *demo-app* directory.

```
steps:

 #Build the container image.
- name: 'gcr.io/cloud-builders/docker'
  id: Build
  args:
  - 'build'
  - '-t'
  - 'gcr.io/$PROJECT_ID/demoapp:$SHORT_SHA'
```

```
  - '.'

# Push image to gcr.io Container Registry
# The SHORT_SHA variable is automatically
replaced by Cloud Build.
- name: 'gcr.io/cloud-builders/docker'
  id: Push
  args:
  - 'push'
  - 'gcr.io/$PROJECT_ID/demoapp:$SHORT_SHA'
```

The SHORT_SHA is replaced with the commit SHA by the Cloud Build tool during runtime. You can test the above script by executing the following steps.

```
cd ~/anthos-demo/demo-app

COMMIT_ID="$(git rev-parse --short=7
HEAD)"

gcloud builds submit --config app-
build.yaml . --substitutions SHORT_SHA=$
{COMMIT_ID}
```

You should see the image getting built and pushed to the container registry. You will receive a success message, as shown in the output below.

```
Finished Step #1 - "Push"
```

```
PUSH
DONE
6538933f-503a-4c92-ad62-ad6ecdae97a9
2021-12-20T11:14:06+00:00  43S       gs://
anthos-demo-335406_cloudbuild/source/
1639998843.535127-9568f800e88d410a9d7187f636391
8c3.tgz  -       SUCCESS
```

You can verify the image in the Container Registry by navigating to the Google Cloud Console N*avigation menu icon* > *Container Registry -> Images*. The tag as shown in figure 2, should match the latest COMMIT_ID that was passed to the build command.

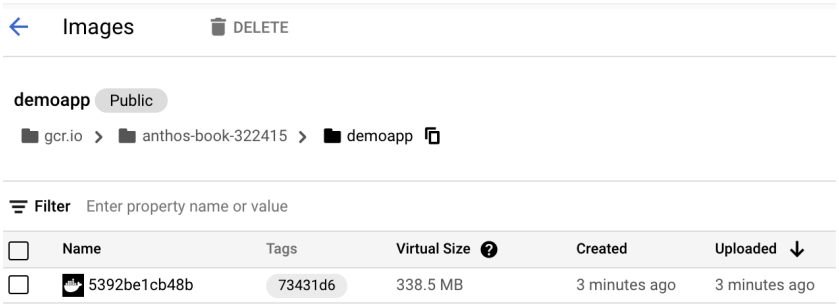


Figure 2 - Image Details

You will now push the code to the google cloud *demo-ap* repository

```
cd ~/anthos-demo/demo-app

git push  --all google
```

The following shows the output of the above command.

```
git push --all google
Enumerating objects: 32, done.
Counting objects: 100% (32/32), done.
Delta compression using up to 8 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (32/32), 6.71 KiB | 6.71
MiB/s, done.
Total 32 (delta 17), reused 32 (delta 17),
pack-reused 0
remote: Resolving deltas: 100% (17/17)
To https://source.developers.google.com/p/
anthos-demo-xx/r/demo-app
 * [new branch]      main -> main
```

You will use the above build script later for the CI pipeline automation.

In the next section, you will create the delivery pipeline to deploy the container image to the Anthos cluster in the respective environment, like staging or production.

Creating the Delivery Pipeline

You will create the delivery pipeline by writing the deployment script using Cloud Build.

The following shows the listing of Cloud Build file: *app-deploy-delivery.yaml* located in the *demo-config* directory.

```
cd ~/anthos-demo/demo-config

nano app-deploy-delivery.yaml

#Continuous Delivery Pipeline Template
for Staging Branch

steps:
# Replace CLOUDSDK_COMPUTE_ZONE and
CLOUDSDK_CONTAINER_CLUSTER with your
# zone and Anthos Cluster name.
- name: 'gcr.io/cloud-builders/kubectl'
  id: Deploy
  args:
  - 'apply'
  - '-f'
  - 'config/staging/kubernetes.yaml'
  env:
  - 'CLOUDSDK_COMPUTE_ZONE=asia-
southeast1-a'
```

```
  - 'CLOUDSDK_CONTAINER_CLUSTER=anthos-
cluster'
```

In the above code, you can replace the zone and cluster name as per your environment.

Now, check in the code in the Google Cloud *demo-config* repository by issuing the following command.

```
cd ~/anthos-demo/demo-config

git checkout -b staging

git add .

git commit -m "Cluster and zone changes
as per environment"

git push --all google-conf
```

You will create a branch called staging to check in the initial changes. The staging branch will be used for initial deployment, and once everything is working, changes can be propagated to another environment like pre-production or production. You will then create the production branch (using git checkout -b production) and push the required changes to the branch. The pipeline for the staging environment can be emulated for the production environment.

The application deployment config is the config/staging/kubernetes.yaml file path mentioned in the app-deploy-delivery.yaml. The kubernetes.yaml file is generated from the kubernetes.yaml.template located in demo-config/manifests folder. The kubernetes.yaml.template file represents a base reference deployment template used to customize the actual configs based on the environment, like staging, production, etc. where it's being deployed.

The templates are used as a best practice base configuration that allows you to bind values to parameters or variables to realize it into a specific environment-based configuration.

Let’s inspect the kubernetes.yaml.template file

```
# Sample Kubernetes Deployment Template
# The COMMIT SHA variable below would be
replaced by the committed build image
# as part of CD process.

apiVersion: apps/v1
kind: Deployment
metadata:
  name: demoapp
  labels:
    app: demoapp
    businessunit: software-demo
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: demoapp
  template:
    metadata:
      labels:
        app: demoapp
        businessunit: software-demo
    spec:
      containers:
      - name: demoapp
        image: gcr.io/GOOGLE_CLOUD_PROJECT/
demoapp:COMMIT_SHA
        ports:
        - containerPort: 8080
---
kind: Service
apiVersion: v1
metadata:
  name: demoapp
spec:
  selector:
    app: demoapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

The customization happens as part of the delivery pipeline where the template - *kubernetes.yaml.template* is realized into *config/staging/kubernetes.yaml* file. In this case, the COMMIT_SHA variable, as shown above is replaced by the container build image (i.e. COMMIT_ID) that we described in the *Creating the Container Image* section. You can use tools like Kustomize to perform various customizations based on your requirement.

You will also see a *businessunit: software-demo* label that indicates the business unit that owns the deployment. We will later make this label a mandatory requirement and validate its presence as part of the ACM Policy configuration and validation.

The following build code carries out the customization steps. The build first checks out the *demo-config* branch, customizes the *kubernetes.yaml.template* using *sed* command and copies the *kubernetes.yaml* to staging directory for deployment as part of staging branch.

```
# This step clones the demo-config repository
and checks out
# staging branch
- id: Clone demo environment repository
  name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: /bin/sh
  args:
  - '-c'
```

```
  - |
    mkdir hydrated-manifests &&
    gcloud source repos clone demo-config && \
    cd demo-config && \
    git checkout staging && \
    git config user.email $(gcloud auth list
--filter=status:ACTIVE --
format='value(account)')


 #This step modifies the kubernetes template
with #image id commit details
 #and copies the files to hydrated-manifests
for further processing.
- id: Generate manifest
  name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
     sed "s/GOOGLE_CLOUD_PROJECT/$
{PROJECT_ID}/g" demo-config/manifests/
kubernetes.yaml.template | \
     sed "s/COMMIT_SHA/${SHORT_SHA}/g" >
hydrated-manifests/kubernetes.yaml


  # If all works well, this step copies the
kubernetes. yaml to the staging directory.
- id: 'Copy config'
```

```
  name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: '/bin/sh'
  args: ['-c', '\cp -r hydrated-manifests/
kubernetes.yaml demo-config/config/staging/
kubernetes.yaml']

# This step pushes the kubernetes.yaml
manifest to demo-config staging branch
- id: Push manifest to demo-config staging
branch
  name: 'gcr.io/cloud-builders/gcloud'
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
    set -x && \
    cd demo-config && \
    git add config/staging/kubernetes.yaml &&
\
    git commit -m "Commit manifest for gcr.io/
${PROJECT_ID}/demoapp:${SHORT_SHA}" && \
    git push origin staging
```

The next step is to integrate the above code after the container image is deployed and pushed to the container registry. So let's combine the above code into a file named *app-build-trigger-cd.yaml* file. The *app-build-trigger-cd.yaml* file is already provided in *demo-config* directory for your reference.

```
cd ~/anthos-demo/demo-app/

nano app-build-trigger-cd.yaml
```

The following shows the high-level snippet headers of the combined listing in app-build-trigger-cd.yaml file.

```
steps:

#Build the container image.
#code ..

#Push image to gcr.io Container Registry
#code ..

#This step clones the demo-config repository
#and checks out  staging branch
#code ..

#This step modifies the kubernetes template
with #image id commit details and copies the
files to hydrated-manifests for further
processing.

# If all works well, this step copies the
kubernetes.yaml to staging directory.
#code ..

# This step pushes the kubernetes.yaml
manifest to demo-config staging branch
```

```
#code ..
```

Running the delivery pipeline

To run the above code, you must grant the required IAM role to the Cloud Build service account to commit changes to the demo-config repository and deploy to the container.

You will perform the following steps to provide the required role.

```
PROJECT_NUMBER="$(gcloud projects describe $
{PROJECT_ID} --format='get(projectNumber)')"

echo $PROJECT_NUMBER
```

Set the following policies using the following command

```
gcloud projects add-iam-policy-binding $
{PROJECT_NUMBER} \
    --member=serviceAccount:${PROJECT_NUMBER}
@cloudbuild.gserviceaccount.com \
   --role=roles/container.developer --
role=roles/source.writer
```

You should get the *Updated IAM Policy* message as shown in the output below.

```
Updated IAM policy for project [1264542XXX0].
bindings:
- members:
-serviceAccount:1264542XX@cloudbuild.gserviceaccount.com
  role: roles/cloudbuild.builds.builder
….
```

Now, let's run the above code.

```
cd ~/anthos-demo/demo-app/

COMMIT_ID="$(git rev-parse --short=7 HEAD)"

gcloud builds submit --config app-build-trigger-cd.yaml . --substitutions SHORT_SHA=${COMMIT_ID}
```

You should see the success message printed at the end, as shown in the output below and the *kubernetes.yaml* file should be committed to the staging branch in the *demo-config* repository. You can verify the same at */config/staging* location.

```
Finished Step #5 - "Push manifest to demo-config staging branch"
PUSH
DONE
ID  CREATE_TIME DURATION  SOURCE
IMAGES  STATUS
```

```
a6cb6feb-c1de-4288-8e3d-91f764bbb831
2021-12-20T12:54:31+00:00  55S       gs://
anthos-demo-335XX_cloudbuild/source/
1640004868.-22c3ba6a9467409d80069ef221fabc3f.tg
z  -       SUCCESS
```

Now, let's run the delivery pipeline, which should deploy the committed *kubernetes.yaml* in the staging environment of your cluster. The *git pull google staging* command below pulls the updated *kubernetes.yaml* file from the staging branch, which is referenced in *app-deploy-delivery.yaml* file.

```
cd ~/anthos-demo/demo-config/
```

```
git pull google-conf staging
```

The following shows the output of the above command.

```
git pull google-conf staging
remote: Counting objects: 13, done
remote: Finding sources: 100% (5/5)
remote: Total 5 (delta 1), reused 5 (delta 1)
Unpacking objects: 100% (5/5), 815 bytes |
163.00 KiB/s, done.
From https://source.developers.google.com/p/
anthos-demo-335406/r/demo-config
```

```
 * branch            staging    -> FETCH_HEAD
   4714a0..0f8e49a  staging    -> google-conf/
staging
Updating 47140a0..0f8e49a
Fast-forward
 config/staging/kubernetes.yaml | 40 +++++++++
++++++++++++++++++++++++++++++
 1 file changed, 40 insertions(+)
 create mode 100644 config/staging/
kubernetes.yaml
```

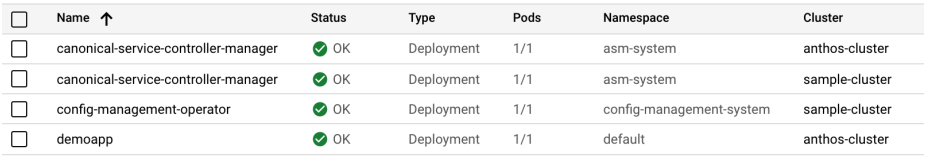
Next, submit the build using the following command.

```
gcloud builds submit --config app-deploy-
delivery.yaml .
```

The following shows the output of the above command.

```
PUSH
DONE
ID
CREATE_TIME                DURATION  SOURCE
IMAGES  STATUS
6be8f0ba-f350-4e33-9532-7911b25ae7b4
2021-12-20T12:58:45+00:00  17S       gs://
anthos-demo-335406_cloudbuild/source/
1640005122.825047-19f0be88d8f64d9bbe0a0f1cd5ae7
c51.tgz  -       SUCCESS
```

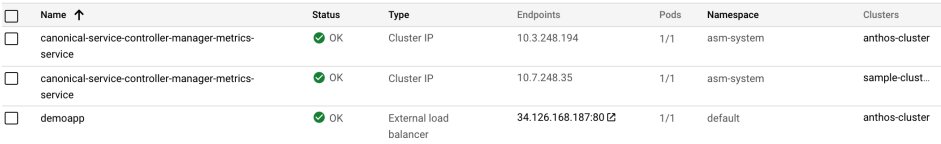
Let's verify if the application was deployed and service is created. Log in to *Google Cloud Console -> Kubernetes Engine -> Workloads* and you should see the *demoapp* listed as shown in figure 3.



Name	Status	Type	Pods	Namespace	Cluster
canonical-service-controller-manager	OK	Deployment	1/1	asm-system	anthos-cluster
canonical-service-controller-manager	OK	Deployment	1/1	asm-system	sample-cluster
config-management-operator	OK	Deployment	1/1	config-management-system	sample-cluster
demoapp	OK	Deployment	1/1	default	anthos-cluster

Figure 3 - Workloads showing demoapp

Click on *Services & Ingress,* and you should see the *demoapp* service running, as shown in figure 4. Copy the IP address from the Endpoint column.



Name	Status	Type	Endpoints	Pods	Namespace	Clusters
canonical-service-controller-manager-metrics-service	OK	Cluster IP	10.3.248.194	1/1	asm-system	anthos-cluster
canonical-service-controller-manager-metrics-service	OK	Cluster IP	10.7.248.35	1/1	asm-system	sample-clust...
demoapp	OK	External load balancer	34.126.168.187:80	1/1	default	anthos-cluster

Figure 4 - Services showing demoapp

Open the browser and navigate to URL *http://<your-endpoint-ip-addres>/healthz* or *http://<your-endpoint-ip-*

addres>/fetchWebsite. You should see the below message being printed in the browser, denoting that the service was executed successfully.

```
{“message”:”ok”}
```

Validate the Policy

As a next step, you will define a governance-based policy to validate your configuration before it is deployed to the cluster as part of the CI pipeline. The policy rule will check and enforce the use of *businessunit* label field in the deployment config.

The definition of policy constraints is stored in the ACM repository at - https://github.com/cloudsolutions-academy/anthos-demo-acm.git/. The cluster directory contains two files - *requiredlabels.yaml* and *deployment-must-have-businessunit.yaml*.

The *requiredlabels.yaml* file below is a constraint template defining our policy that expects a string array of labels as an input. If the input label is not present, it will block the creation of the deployment configuration.

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
```

```
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
      validation:
        # Schema definition for the
'parameters' field
        openAPIV3Schema:
          properties:
            labels:
              type: array
              items: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        violation[{"msg": msg, "details":
{"missing_labels": missing}}] {
          provided := {label |
input.review.object.metadata.labels[label]}
          required := {label | label :=
input.parameters.labels[_]}
          missing := required - provided
          count(missing) > 0
          msg := sprintf("The following
label(s) are required: %v", [missing])
```

```
        }
```

The above policy is then enforced as a constraint in the *deployment-must-have-businessunit.yaml* file. The below constraint applies a policy that mandates the use of labels *app* and *businessunit* in the deployment configuration.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: deployment-must-have-business-unit
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
  parameters:
    labels: ["app", "businessunit"]
    message: "Deployment objects should have
an ‘businessunit' label indicating which
business unit owns it"
```

Now, let's integrate and validate the deployment against the above policy as part of the CI build process as shown below.

```
  # This step fetches the policies from the
Anthos Config Management repository and
consolidates every resource in a single file.
- id: 'Download policies'
  name: 'gcr.io/kpt-dev/kpt'
  entrypoint: '/bin/sh'
  args: ['-c', 'kpt pkg get https://
github.com/cloudsolutions-academy/anthos-demo-
acm.git/cluster@main constraints
                  && kpt fn source
constraints/ hydrated-manifests/ > hydrated-
manifests/kpt-manifests.yaml']

  # This step validates that all resources
comply with all policies.
- id: 'Validate against policies'
  name: 'gcr.io/config-management-release/
policy-controller-validate'
  args: ['--input', 'hydrated-manifests/kpt-
manifests.yaml']
```

Once the container image is pushed, it will validate the configuration against the policy before committing the *kubernetes.yaml* file in the staging branch.

You will use *kpt* tool to fetch all the constraints from our ACM repository at https://github.com/cloudsolutions-academy/anthos-demo-acm.git. and consolidate the Kubernetes configurations and the constraints in a single

file named *kpt-manifests.yaml*. The *kpt-manifests.yaml* file is then validated by the policy controller to ensure all the constraints are met. The deployment will fail when there is a violation of policy.

The updated CI pipeline with the above changes is made available at app-build-trigger-acm-ci.yaml (in demo-app repository).

The following shows the high-level snippet headers of combined listing for app-build-trigger-acm-ci.yaml.

```
steps:
 #Build the container image.
 # code …

 # Push image to gcr.io Container Registry
 # code …

 #This step clones the demo-config repository
 # and checks out staging branch
 # code …


 #This step modifies the kubernetes template
 #with #image id commit details and copies the
 #files to hydrated-manifests for further
 #processing.
 # code …
```

```
# This step fetches the policies from the
Anthos # Config Management repository and
consolidates  # every resource in a single
file.
# code ..

# This step validates that all resources
comply # with all policies.
# code ..

# If all works well, this step copies the
# kubernetes.yaml to staging directory.
# code …

# This step pushes the kubernetes.yaml
manifest # to demo-config staging branch
# code ..
```

To run the above build, execute the following steps -

```
cd ~/anthos-demo/demo-app/

COMMIT_ID="test-$(git rev-parse --short=7
HEAD)"

gcloud builds submit --config app-build-
trigger-acm-ci.yaml . --substitutions
SHORT_SHA=${COMMIT_ID}
```

You should receive a successful build message as shown in the output below.

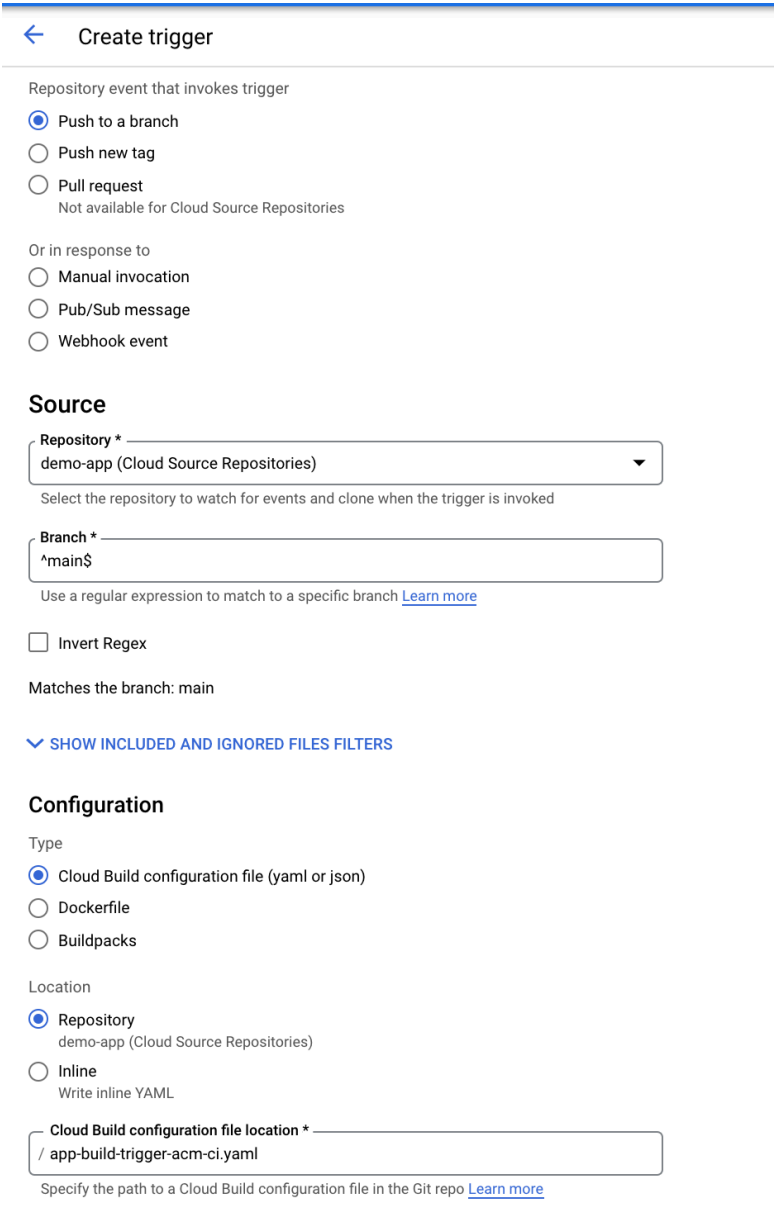
```
Finished Step #7 - "Push manifest to demo-
config staging branch"
PUSH
DONE
ID
CREATE_TIME                DURATION  SOURCE
IMAGES  STATUS
67732e8a-0ffb-4362-9c50-4c141c7803c7
2021-12-20T13:44:30+00:00  1M5S      gs://
anthos-demo-x-cloudbuild/source/
1640007867.806441-xx8348bd4410dbb2dd55c.tgz  -
SUCCESS
```

So far, you have manually executed the CI/CD build process. The next section will demonstrate how to set up an automated CI/CD process.

SETTING UP CI/CD PROCESS

To automate the CI/CD process, you will set up triggers.

Go to Google *Cloud Console -> Cloud Build - Triggers*. You will create two triggers, one for *app-build-trigger-acm-ci.yaml* (the CI part) and one for *app-deploy-delivery.yaml* (the CD part)



- Click on *Create Trigger* button

- Specify the name as *App-CI*

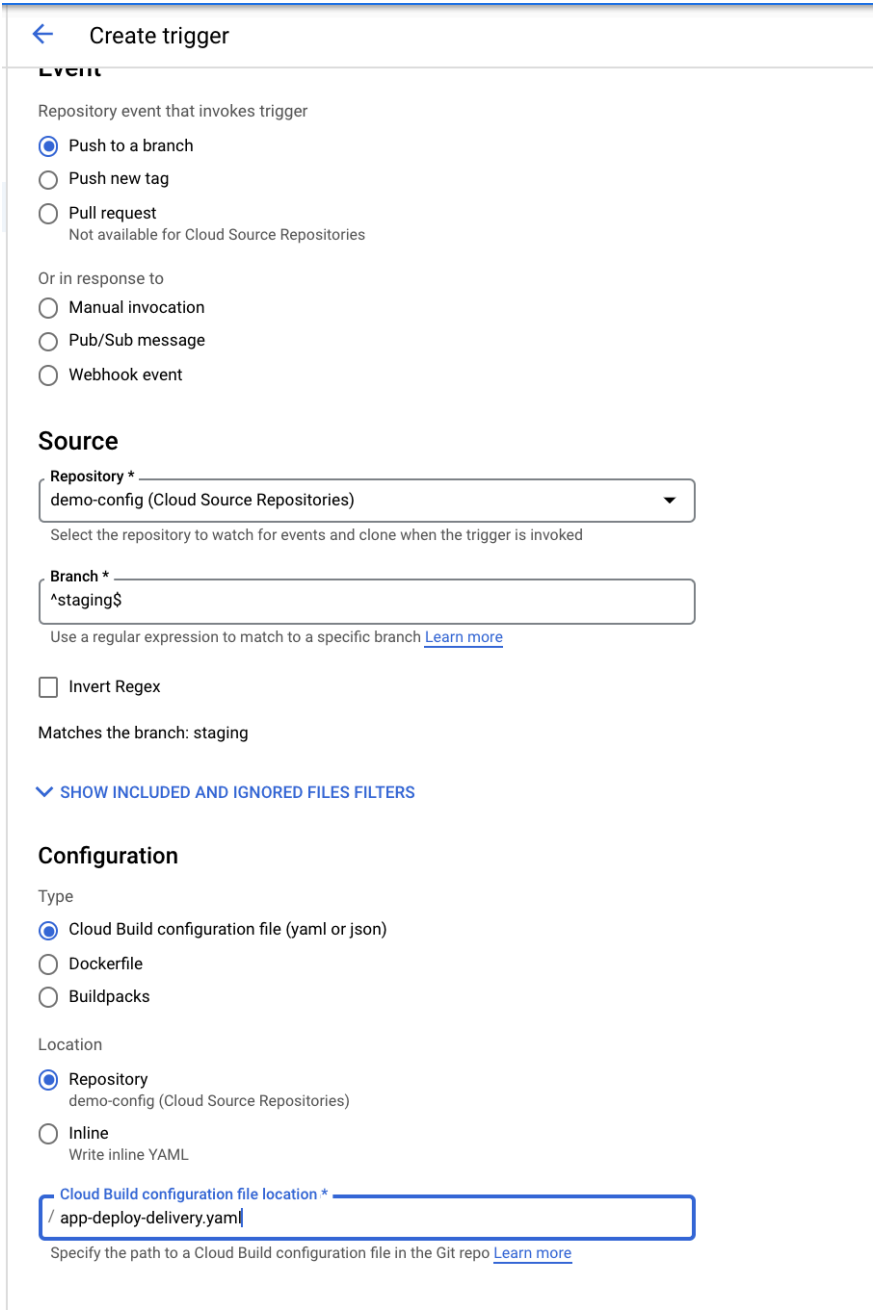
- In the *Event*, keep the default *Push to Branch* option selected
- In the *Source*, specify the *demo-app* repository
- For the branch, specify main branch as wildcard pattern *^main$*
- Keep the configuration as Cloud Build file
- In *Location*, specify the build file *app-build-trigger-acm-ci.yaml*
- Click *Create* button

The figure 5 show the above configuration details.

Figure 5 - Create Trigger for Continuous integration

With the above trigger configured, any commit to the application's main branch will trigger the CI execution pipeline. The pipeline will commit the *kubernetes.yaml* to the staging branch on successful execution.

The next step is to create a similar trigger on the staging



branch, which will execute the *app-deploy-delivery.yaml*

file.

- Click on *Create Trigger* button.
- Specify the name as *App-CD*
- In the *Event*, keep the default *Push to Branch* option selected
- In the *Source*, specify the *demo-config* repository
- For the branch, specify main branch as wildcard pattern `^staging$`
- Keep the configuration as Cloud Build file
- In *Location*, specify the build file *app-deploy-delivery.yaml*
- Click *Create* button

The figure 6 show the above configuration details

Figure 6 - Create Trigger for Continuous delivery

You can now test the integrated CI/CD pipeline. Let's make some changes to *server.js* code.

```
cd ~/anthos-demo/demo-app
```

```
nano server.js
```

Replace the *ok* print message highlighted in bold with the *service is running* message as shown below.

```
app.get('/healthz', (req, res) => {
console.log(req.connection.remoteAddress);
res
    .status(200)
    .json({message: "ok"})
    .end();
});
```

```
by
```

```
app.get('/healthz', (req, res) => {
console.log(req.connection.remoteAddress);
res
    .status(200)
    .json({message: “server is running"})
    .end();
});
```

Save the file.

Commit the changes by executing the following commands.

```
git add .
```

```
git commit –m “Health service message changed”
```

```
git push --all google
```

Go to *Google Cloud Console -> Cloud Build -> History,* and you should see the *App-CI* build being triggered as shown in figure 7.

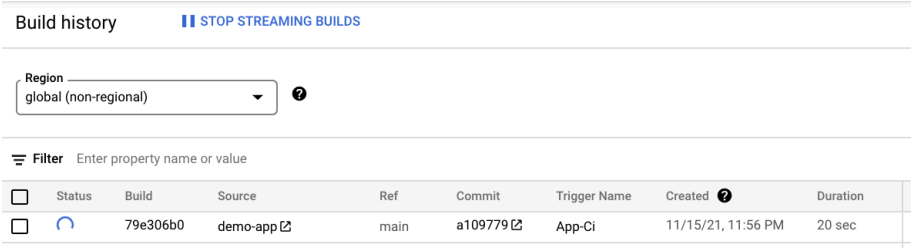


Figure 7 - Cloud Build History

Once the *App-CI* build is successfully executed, as shown in figure 8.8 (see the green ticks), the *App-CD* pipeline will be triggered, and the latest build will be deployed to the cluster in the staging environment.

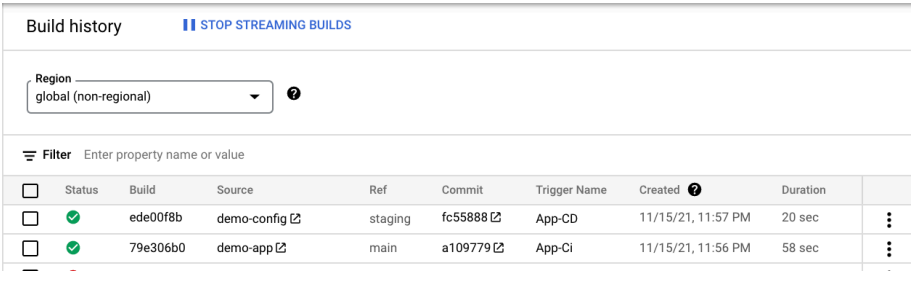


Figure 8.8 - Cloud Build Submission Results

Open the URL *http://<your-endpoint-ip-addres>/healthz* in the browser, and you should see the below latest message being printed.

```
{"message":"server is running”}
```

This completes the setup and execution of end to end CI/CD process for Anthos.

Now let's test the policy violation use case. You will remove the required *businessunit* label from the *kubernetes,template.yaml* file, and in doing so, the build should fail as it violates our policy.

Follow the steps to modify the *kubernetes,template.yaml* file.

```
cd ~/anthos-demo/demo-config/
```

```
nano manifests/kubernetes.yaml.template
```

Just replace the label key *businessunit* with *businessunit1* in the file. Save the file.

Commit the changes by running the command

```
git add .
```

```
git commit –m “Negative policy test”
```

```
git push ––all google–conf
```

Go back to *Google Cloud Console. Go to Cloud Build -> Triggers* and click on *Run* on the *App-CI* row to run the build manually.

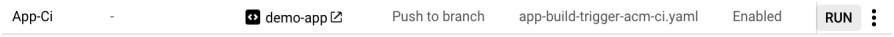
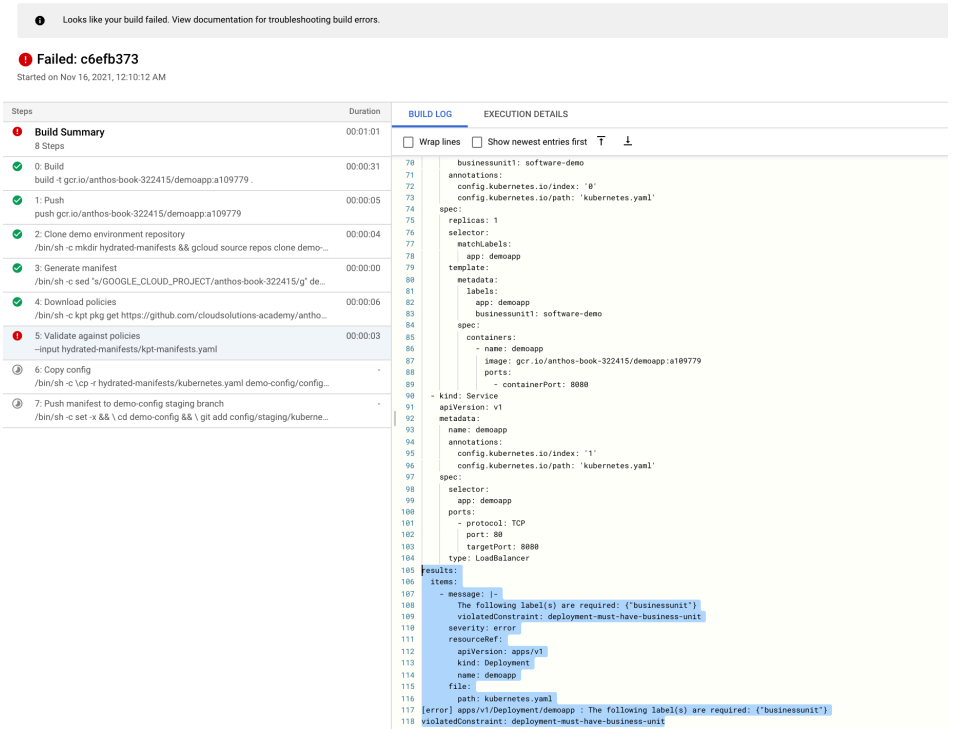


Figure 8 - Manually Run the Build

Go to History to see the build status. You will see the build failed at Step 5 - *Validate against policies* with the message as shown in figure 9.

Figure 9 - Validate Policy Failed

```
“Step #5 – "Validate against policies":
[error] apps/v1/Deployment/demoapp : The
```



`following label(s) are required: {“businessunit”}"`

`Step #5 - "Validate against policies": violatedConstraint: deployment-must-have-business-unit`

This test verifies that our policy execution is working fine, the CI process fails due to policy violation, and changes are not committed to the staging branch.

SUMMARY

In this paper, you learned the importance of DevOps and CI/CD as a practice. You learned how to build an end-to-end CI/CD process using a step-by-step approach (from manual to automation).

You can further refine your CI/CD process as per your application release requirements. Things like adding unit tests, performing container image scanning, setting up more governance-based policies, and working with ACM components to enforce these policies and configurations on different clusters across environments.

ABOUT THE AUTHOR

Navveen Balani is a Google Cloud Certified Fellow with over two decades of experience in building products using exponential technology. You can connect/follow him for the latest technology updates at - https://www.linkedin.com/in/naveenbalani/

Website - https://navveenbalani.dev/

Youtube - https://www.youtube.com/c/CloudSolutionsAcademy