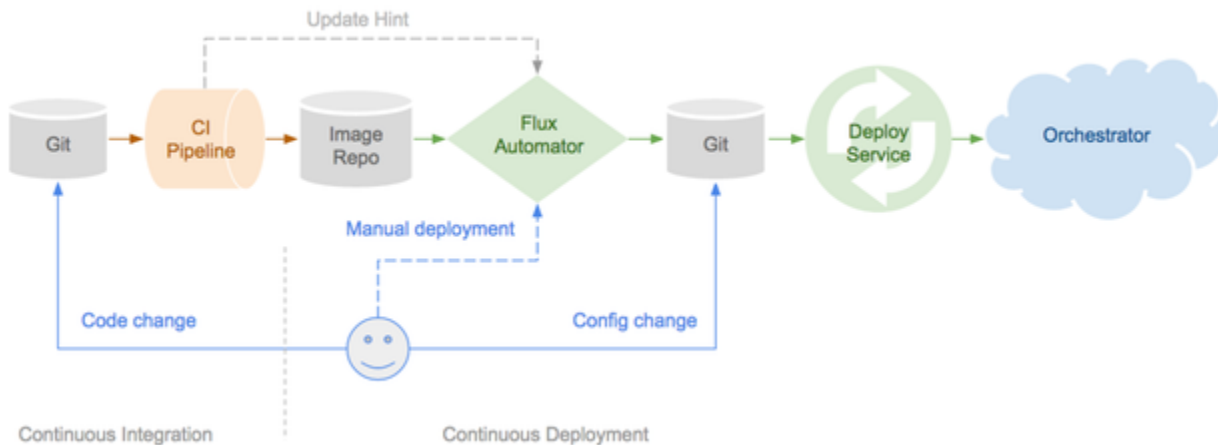


How To GitOps Implementation

What is GitOps?

- GitOps is the art and science of using Git pull requests to manage infrastructure provisioning and software deployment.
- GitOps is a way to do Kubernetes cluster management and application delivery.
- GitOps uses Git as a single source of truth for infrastructures like declarative infrastructure and the applications. On Git, set delivery pipelines, make pull requests, accelerate and simplify both the application deployments and the operations task to Kubernetes.
- A GitOps pipeline enables developers to speed up development and to safely and securely make changes to complex applications.



- **Code** – We can write our application code and push it to Git.
- **Build (Pipeline as Code)** – Build pipeline code(yaml) in Git to build code using CI tools.
- **Provision (Infrastructure as Code)** – Write a terraform file or Ansible playbook in Git to provision a Kubernetes cluster.
- **Deploy** – Write a deployment manifest for Kubernetes or on the basis of the application.

GIT

- Distributed Version Control System
- Manage Source Code
- Any Text file tracked
- Collaboration

Ops

- Configure, build, test, deploy
- CI/CD
- Monitor, fix
- Upgrades
- Rollbacks
- Automation

GitOps

- Ops practices using Git
- Use Git as a source of truth
- Drive operations through git repo
- Get version control, history, peer review and rollback
- Continuous Delivery- Automates pipelines
- Webhooks: push to trigger build/test
- PR's & merge branch to deploy

Why Use GitOps?

- You get a single tool and interface for controlling your infrastructure. This obviates the need to use a different tool for controlling different types of infrastructure.
- You get version control for all of the changes that you make to your configuration. That's handy for rolling back changes, as well as

auditing purposes.

- You can use diff to detect changes and generate alerts automatically. This means not only that you have a way to monitor for changes constantly, but also that if actual conditions diverge from the way things are supposed to be configured, the problem will be detected easily.
- Because you're using Git pull requests (something with which most software developers are already familiar), you don't have to teach your team a new tool in order to manage infrastructure.

How GitOps Works?

The deploy component divided into two parts: one is a Flux Automator used to observe new builds and update configurations on the new release, and second Flux synchronizer to ensure that orchestrator is in the correct state.

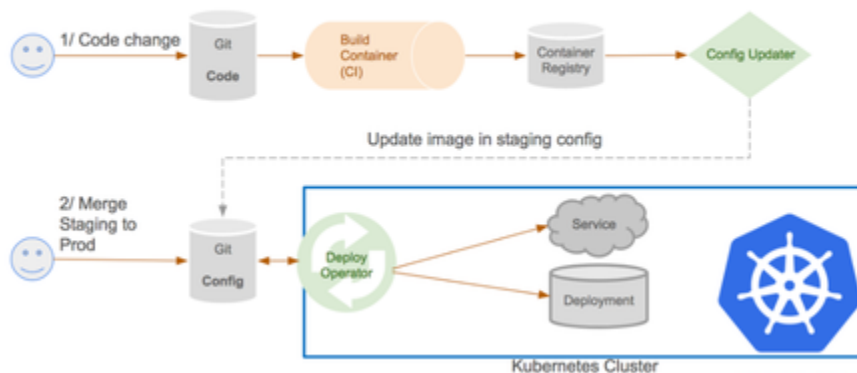
Workflow for the new creation or updating of a new feature –

- First, make a pull request for a new feature pushed to GitHub.
- Go for the code review and merge to Git.
- Then Git automatically merge the triggers, the process of CI and builds the pipeline, runs tests and then builds a new image and deposits to a registry.
- The Deployment Automator checks the image registry, reviews the image, and then it pulls the new image from the registry and updates the YAML file of that project in the config repo.
- The Deployment Synchronizer installed to the cluster and detects that the cluster is out of date. Then It pulls the changes in the manifests from the config repo and then deploys the new feature to production.

How to Adopt GitOps?

- Create a new Repository.
- Create a new directory, open it and perform a "git init" to create a new git repository.
- Checkout Repository.
- Create a working copy of a Local repository by running the command "git clone /path to the directory.
- While using a remote server, enter Username and Password.
- Create Required Files as per the requirements of the Application.
- Then by using "git add " add those files on the Repository at Login.
- After Adding files do the commit by using this command "git commit -m "Commit message."
- Changes done on local working, now copy to send those changes to the remote repository, execute "git push origin , approve code review and merge to Git.
- After Code pushed on Git Repository, then the CI pipeline automatically starts and runs tests.
- Then build new Image saved in Registry Like Docker Hub.
- Git automatically merges, triggers the process of CI and builds the pipeline, and runs tests, builds a new image and deposits to a registry.
- The Deployment Automator checks the image registry, reviews the image, and then it pulls the new image from the registry and updates the YAML file of that project in the configuration repo.
- The Deployment Synchronizer installed to the cluster and detects that the cluster is out of date. It pulls the changes in the manifests from the configuration repository and then deploys the new feature to production.

Example GitOps Pipeline



The project: understanding of the whole CI/CD flow

Source code

Let's just consider the following files:

- *app.py* exposes a single HTTP endpoint and returns a string

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

- *requirements.txt* defines the dependency, *Flask* library, needed by *app.py*

```
Flask==1.0.2
```

- *Dockerfile* use to build an image out of the source code

```
FROM python:3-alpine
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD python /app/app.py
```

Making sure it works

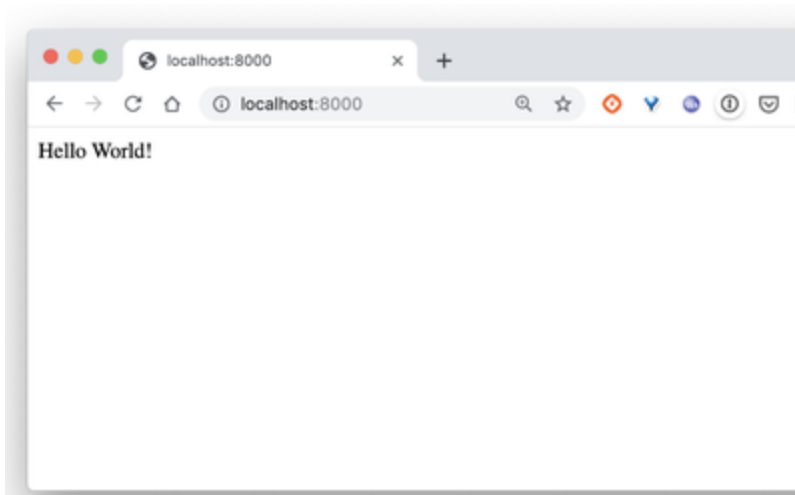
We start by creating a first Docker image for our application

```
$ docker image build -t hello:1.0 .
```

Once the image is built, we can run a container using it.

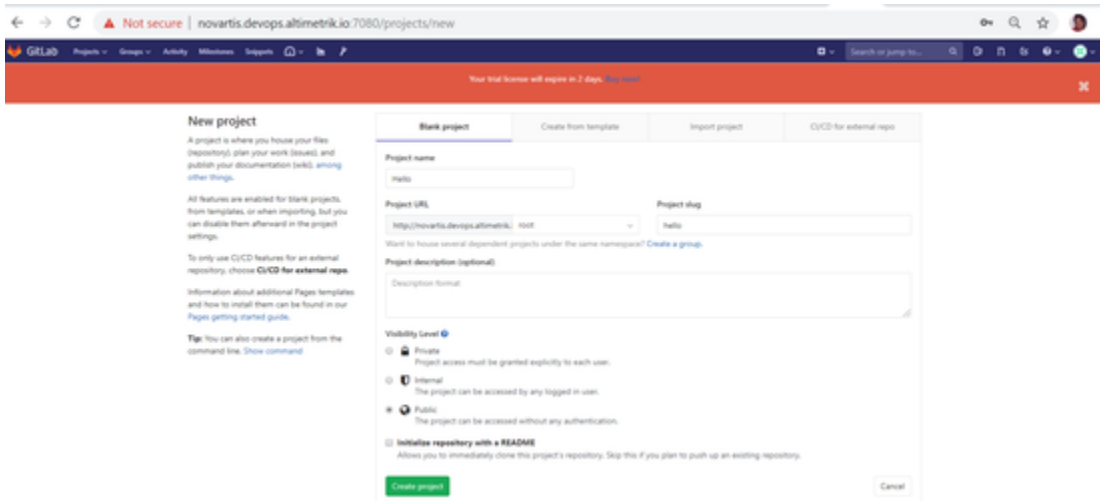
```
$ docker container run -p 8000:8000 hello:1.0
* Serving Flask app "app" (lazy loading)
* Environment: production
WARNING: Do not use the development server in a production
environment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
```

Our server is listening on port 8000 as we can see below.



GitLab project

We will use GitLab to manage this application, so let's create a new project named *hello* :



We can then initialize git for the application folder and push everything into the GitLab project:

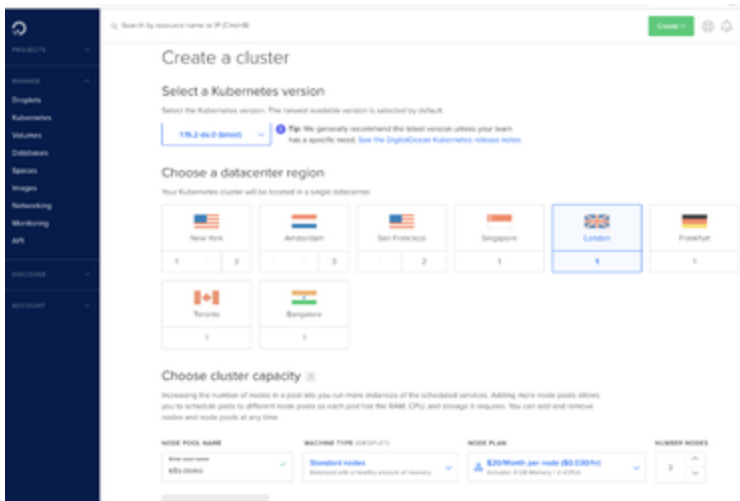
```
$ git init
$ git remote add origin git@gitlab.com:lucj/hello.git
$ git add .
$ git commit -m "Initial commit"
$ git push -u origin master
```

Enters Kubernetes

As we want to deploy our application on a Kubernetes cluster, we will use the Kubernetes integration functionality of GitLab to import the configuration of an external cluster in the project.

Creation of a managed cluster

In this example, we will setup a 3 worker nodes cluster, the manager nodes being managed by us.



It takes a couple of minutes for the infrastructure to be provisioned and the cluster created. Once it's done we need to retrieve the kubeconfig file so our `kubectl` client can communicate with the cluster's API Server. We will use the `doctl` command line and save this configuration in `k8s-demo.cfg` file:

```
$ doctl k8s cluster cfg show k8s-demo > k8s-demo.cfg
```

We then configure `kubectl` to communicate with our cluster setting the KUBECONFIG environment variable:

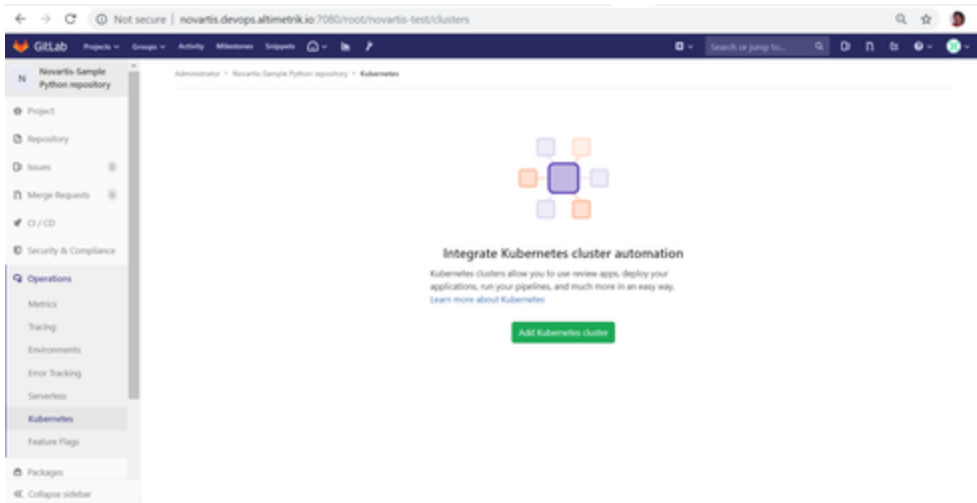
```
$ export KUBECONFIG=$PWD/k8s-demo.cfg
```

Let's check the state of our cluster:

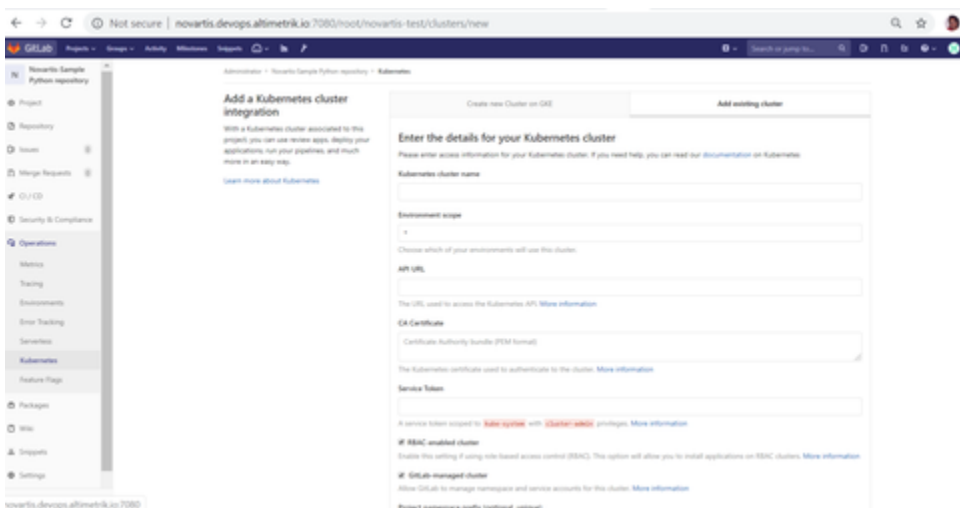
```
$ kubectl get nodes
NAME                STATUS    ROLES    AGE   VERSION
k8s-demo-rlf5       Ready     <none>    2m10s v1.15.2
k8s-demo-rlfh       Ready     <none>    2m40s v1.15.2
k8s-demo-rlfk       Ready     <none>    2m33s v1.15.2
```

Integration with the GitLab project:-

From the GitLab's web interface, it's very easy to integrate an external Kubernetes cluster to a project. We just need to enter the *Operations > Kubernetes* and then click on *Add Kubernetes cluster*:



We then need to select the *Add existing cluster* tab. From there we need to fill a couple of fields. The firsts of them can easily be retrieved from the configuration file:



- Cluster name
- URL of the API Server
- Cluster's CA certificate

To provide the cluster CA certificate to GitLab, we need to decode the one specified in the configuration (as it's encoded in base64).

```
$ kubectl config view --raw \
-o=jsonpath='{.clusters[0].cluster.certificate-authority-data}' \
| base64 --decode
```

- Service token

The process to get an identification token involves several steps. We first need to create a `ServiceAccount` and provide it with the `cluster-admin` role. This can be done with the following command:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitlab-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: gitlab-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: gitlab-admin
  namespace: kube-system
EOF
```

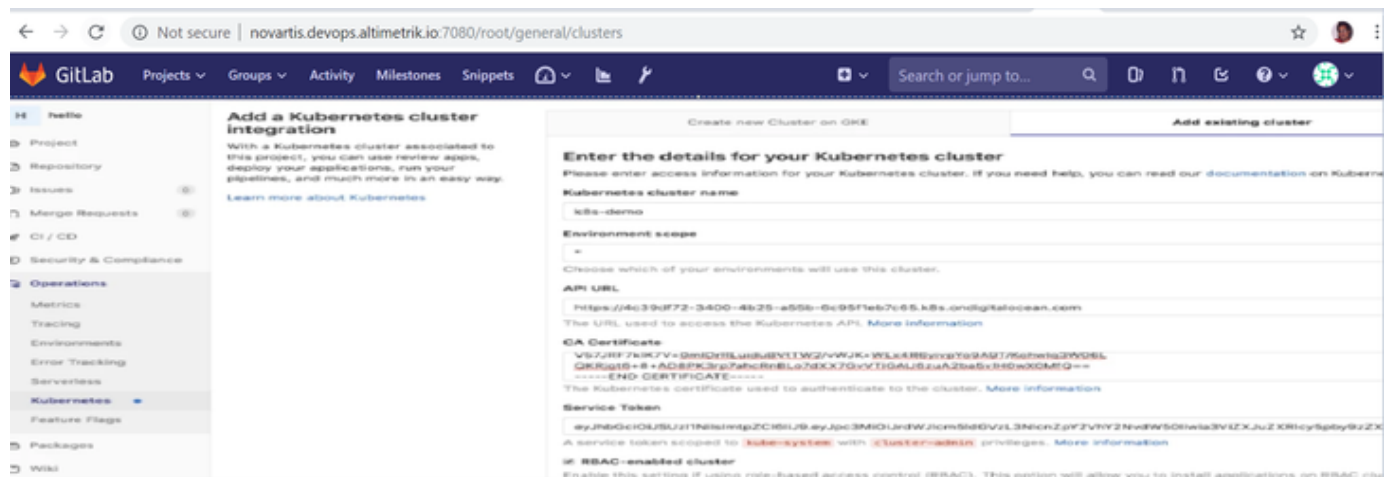
Once the ServiceAccount is created, we retrieve the associated *Secret*:

```
$ SECRET=$(kubectl -n kube-system get secret | grep gitlab-admin |
awk '{print $1}')
```

add extract its JWT token, the one we need to enter in the *Service Token* field in the GitLab interface:

```
$ TOKEN=$(kubectl -n kube-system get secret $SECRET -o
jsonpath='{.data.token}' | base64 --decode) && echo $TOKEN
```

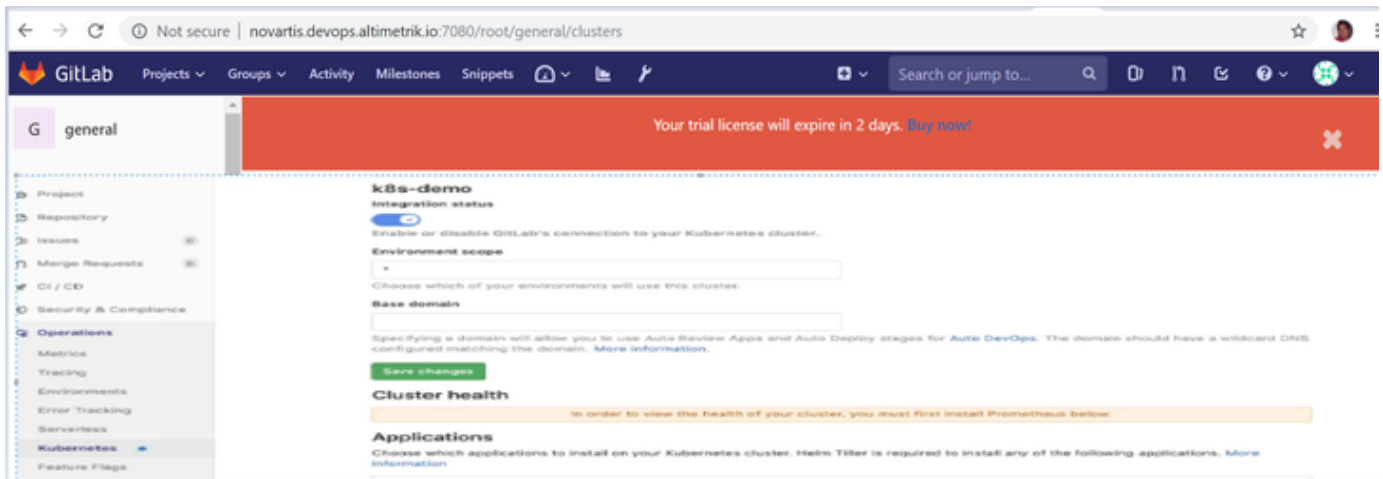
Before validating the cluster integration, we uncheck the *GitLab-managed-cluster* checkbox as we will manage our own namespaces.



The screenshot shows the GitLab web interface at the URL `novartis.devops.altimetrik.io:7080/root/general/clusters`. The left sidebar shows the 'Operations' menu with 'Kubernetes' selected. The main content area is titled 'Add a Kubernetes cluster integration' and contains a form for creating a new cluster. The form has two tabs: 'Create new Cluster on GKE' (selected) and 'Add existing cluster'. The 'Create new Cluster on GKE' tab is active, showing the 'Enter the details for your Kubernetes cluster' form. The form includes the following fields and options:

- Kubernetes cluster name:** `k8s-demo`
- Environment scope:** `*`
- API URL:** `https://4c39df72-3400-4b25-a55b-6c95f1eb7c65.k8s.onligitalocean.com`
- CA Certificate:** A long base64-encoded string.
- Service Token:** A long base64-encoded string.
- RBAC-enabled cluster:** An unchecked checkbox.

Once the cluster is integrated GitLab allows to install several applications in one click through Helm charts.



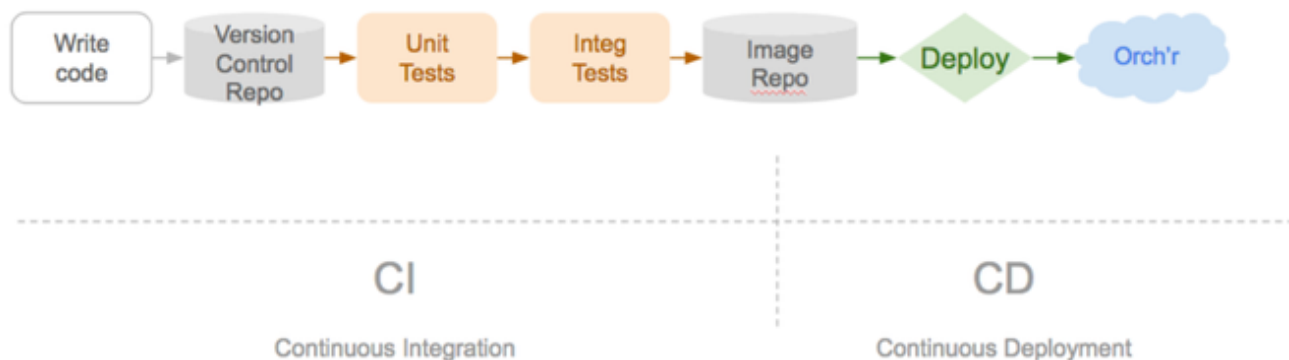
Kubernetes cluster integrated with our GitLab's project

GitOps: Pipelines intro

Our end to end pipeline is wired directly into our clusters and encompasses every step in our build, deploy, manage, monitor lifecycle. We only deploy containers and all our workloads are containerized .

- CI runs tests; output is delivered to a container image repository
- CD systems deploy a container automatically (or on request, i.e. manually)

In diagram form this might look like this:



CD follows the GitOps model of deployment, where desired configuration changes are first pushed to Git, and the cluster state then syncs to the desired state in git. This is a departure from imperative pipelines which do not traditionally use Git repositories to hold application config.

To push new container images into to a cluster managed by CD, the following workflow might be used:

Create the continuous integration pipeline

Build And Publish A New Container Image

```
docker build -t mycompany/guestbook:v2.0 .
docker push mycompany/guestbook:v2.0
```



```
git clone https://github.com/mycompany/guestbook-config.git
cd guestbook-config

# kustomize
kustomize edit set imagetag mycompany/guestbook:v2.0

# ksonnet
ks param set guestbook image mycompany/guestbook:v2.0

# plain yaml
kubectl patch --local -f config-deployment.yaml -p '{"spec":{"template":{"spec":{"c

git add . -m "Update guestbook to v2.0"
git push
```

We configure Cloud Build to automatically run a small unit test, build the container image, and then push it to Container Registry. Pushing a new commit to Cloud Source Repositories triggers automatically this pipeline. The `cloudbuild.yaml` file already included in the code is the pipeline's configuration.

cloudbuild.yaml

```
steps:
# This step runs the unit tests on the app
- name: 'python:3.7-slim'
  id: Test
  entrypoint: /bin/sh
  args:
    - -c
    - 'pip install flask && python test_app.py -v'

# This step builds the container image.
- name: 'gcr.io/cloud-builders/docker'
  id: Build
  args:
    - 'build'
    - '-t'
    - 'gcr.io/$PROJECT_ID/hello-cloudbuild:$SHORT_SHA'
    - '.'

# This step pushes the image to Container Registry
# The PROJECT_ID and SHORT_SHA variables are automatically
# replaced by Cloud Build.
- name: 'gcr.io/cloud-builders/docker'
  id: Push
  args:
    - 'push'
    - 'gcr.io/$PROJECT_ID/hello-cloudbuild:$SHORT_SHA'
```

1. Open the **Triggers** page of Cloud Build.

[GO TO TRIGGERS](#)

2. Click **Create trigger**.
3. Select "Cloud Source Repositories" as source and click **Continue**.
4. Select the `hello-cloudbuild-app` repository and click **Continue**.
5. In the "Triggers settings" screen, enter the following parameters:
 - Name: `hello-cloudbuild`

- Branch (regex): `master`
 - Build configuration: **cloudbuild.yaml**
6. Click **Create trigger**.
 7. In Cloud Shell, push the application code to Cloud Source Repositories to trigger the CI pipeline in Cloud Build.

```
cd ~/hello-cloudbuild-app
git push google master
```

Create the continuous delivery pipeline

In Cloud Shell, clone the *hello-cloudbuild-env* repository and create the *production* branch. It is still empty.

```
cd ~
gcloud source repos clone hello-cloudbuild-env
cd ~/hello-cloudbuild-env
git checkout -b production
```

Copy the `cloudbuild-delivery.yaml` file available in the *hello-cloudbuild-app* repository and commit the change.

```
cd ~/hello-cloudbuild-env
cp ~/hello-cloudbuild-app/cloudbuild-delivery.yaml ~/hello-cloudbuild-env/cloudbuild.yaml
git add .
git commit -m "Create cloudbuild.yaml for deployment"
```

cloudbuild-delivery.yaml

```
steps:
# This step deploys the new version of our container image
# in the hello-cloudbuild Kubernetes Engine cluster.
- name: 'gcr.io/cloud-builders/kubectl'
  id: Deploy
  args:
  - 'apply'
  - '-f'
  - 'kubernetes.yaml'
  env:
  - 'CLOUDSDK_COMPUTE_ZONE=us-central1-b'
  - 'CLOUDSDK_CONTAINER_CLUSTER=hello-cloudbuild'

# This step copies the applied manifest to the production branch
# The COMMIT_SHA variable is automatically
# replaced by Cloud Build.
- name: 'gcr.io/cloud-builders/git'
  id: Copy to production branch
  entrypoint: /bin/sh
  args:
  - '-c'
  - |
    set -x && \
    # Configure Git to create commits with Cloud Build's service account
```

Create a *candidate* branch and push both branches for them to be available in Cloud Source Repositories.

```
git checkout -b candidate
git push origin production
git push origin candidate
```

Grant the Source Repository Writer IAM role to the Cloud Build service account for the *hello-cloudbuild-env* repository.

```
PROJECT_NUMBER="$(gcloud projects describe ${PROJECT_ID} \
  --format='get(projectNumber)')"
cat >/tmp/hello-cloudbuild-env-policy.yaml <<EOF
bindings:
- members:
  - serviceAccount:${PROJECT_NUMBER}@cloudbuild.gserviceaccount.com
  role: roles/source.writer
EOF
gcloud source repos set-iam-policy \
  hello-cloudbuild-env /tmp/hello-cloudbuild-env-policy.yaml
```

Create the trigger for the continuous delivery pipeline

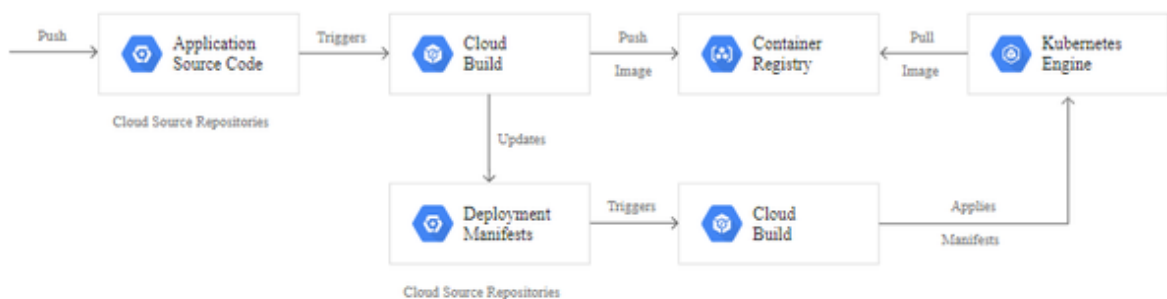
In this section, you configure Cloud Build to be triggered by a push to the *candidate* branch of the *hello-cloudbuild-env* repository.

1. Open the **Triggers** page of Cloud Build.
- GO TO TRIGGERS
2. Click **Add trigger**.
3. Select "Cloud Source Repositories" as source and click **Continue**.
4. Select the *hello-cloudbuild-env* repository and click **Continue**.
5. In the "Triggers settings" screen, enter the following parameters:
 - Name: hello-cloudbuild-deploy
 - Branch (regex): candidate
 - Build configuration: **cloudbuild.yaml**
6. Click **Create trigger**.

'Push' Over 'Pull'

In GitOps you want your whole system to be expressed declaratively. This is where cloud native tools really help:

- **Orchestration:** As you gain momentum and your applications grow, you will find it easier to use declarative platforms like [Kubernetes](#) or [Docker Swarm](#), to automate orchestration and systematise it, instead of doing everything using 'config snowflakes'.
- **Idempotent installation:** Use tools like [kubeadm](#) (or [kops](#) or [kubicorn](#)) that provide idempotent Kubernetes installation which is a prerequisite for GitOps. Or use a cloud service like [GKE](#), [ECS](#), [ACS](#). At Weaveworks, we have been able to use Ansible for idempotent installation.
- **Resources:** As explained in the [previous post](#), we use Terraform to provision resources.



For convenience, the `argocd` CLI can be downloaded directly from the API server. This is useful so that the CLI used in the CI pipeline is always kept in-sync and uses binary that is always compatible with the CD API server.

```
export ARGOCD_SERVER=argocd.mycompany.com
export ARGOCD_AUTH_TOKEN=<JWT token generated from project>
curl -sSL -o /usr/local/bin/argocd https://${ARGOCD_SERVER}/download/argocd-linux-a
argocd app sync guestbook
argocd app wait guestbook
```

