Itera Container Cloud

Architecture and guide

INTRODUCTION **2**

HOW TO ACCESS THE ENVIRONMENT **2**

KUBERNETES AND CONTAINER CLOUD ARCHITECTURE **3**

OVERVIEW 3

OPENSTACK CLOUD PROVIDER FOR KUBERNETES 4

LOGGING, MONITORING AND ALERTING 5

CI/CD **6**

JENKINS 7

JENKINS MASTER 7

JENKINS SLAVE 7

GITLAB AND DOCKER REGISTRY 8

HOW TO DEPLOY APPLICATIONS ON CONTAINER CLOUD **9**

USING KUBECTL AND KUBERNETES API DIRECTLY 9

USING YAML FILE 9

RUN THE APPLICATION DIRECTLY 11

USING HELM 12

EXAMPLE : INSTALL GOGS USING HELM 12

PERSISTENT VOLUMES FOR KUBERNETES 13

INTRODUCTION

This guide walks you through the process of accessing your environment and how to install applications on container cloud using different methods such as kubectl/armada and helm.
For accessing your environment you need to have a valid SSH key, the IP of environment, the IP of Grafana, Kibana and Kubernetes Dashboard. As soon as you have service from Itera Technologies, you will receive an email which will have all required credentials and related informations.

HOW TO ACCESS THE ENVIRONMENT

To access the environment just execute the command as shown in below with correct IP of the environment, SSH key of the environment. But before accessing the environment make sure that you have file which has SSH key saved in it. As soon as you save the key you are ready to roll. Now you can access your environment by following command.

```
ssh -i cloud.key user@KUBERNETES_IP_address
```

So now you have full access to your environment which has kubernetes installed in it. If you would like check whether pods and nodes are working fine then just execute given command.

To check the pods status : `kubectl get pods --all-namespaces -o wide`
To check the nodes status: `kubectl get nodes`

Please don't touch or don't try to modify pods in kube-system. If you do then service will not work according to expectations. As soon as you know that all the nodes as well as pods are running then just move towards understanding the

basic aspects of container and cloud architecture

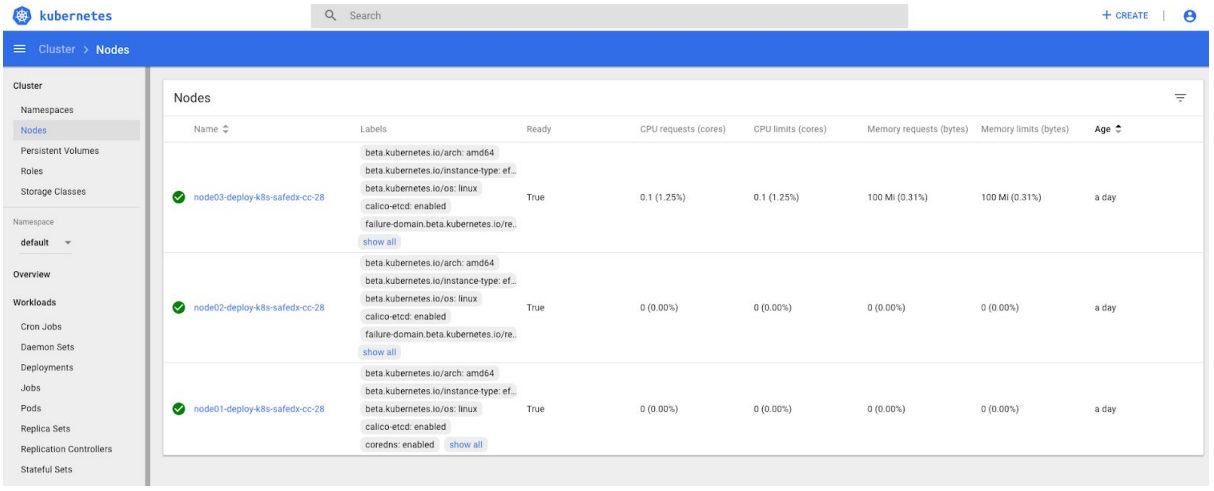
KUBERNETES AND CONTAINER CLOUD ARCHITECTURE

OVERVIEW

Main goal of this architecture is to provide flexible infrastructure for container services. This goal is achieved by using Kubernetes as cornerstone. OpenStack Cinder for providing persistent storage. And Project Calico as robust networking solution.
This architecture also includes CI/CD build around Jenkins tooling. Supported by GitLab and a Docker registry as storages of code and containers.
Also this architecture is created to be portable and presents opportunities for re-using it's components buy higher level applications.

The Container Cloud is composed of 5 nodes:

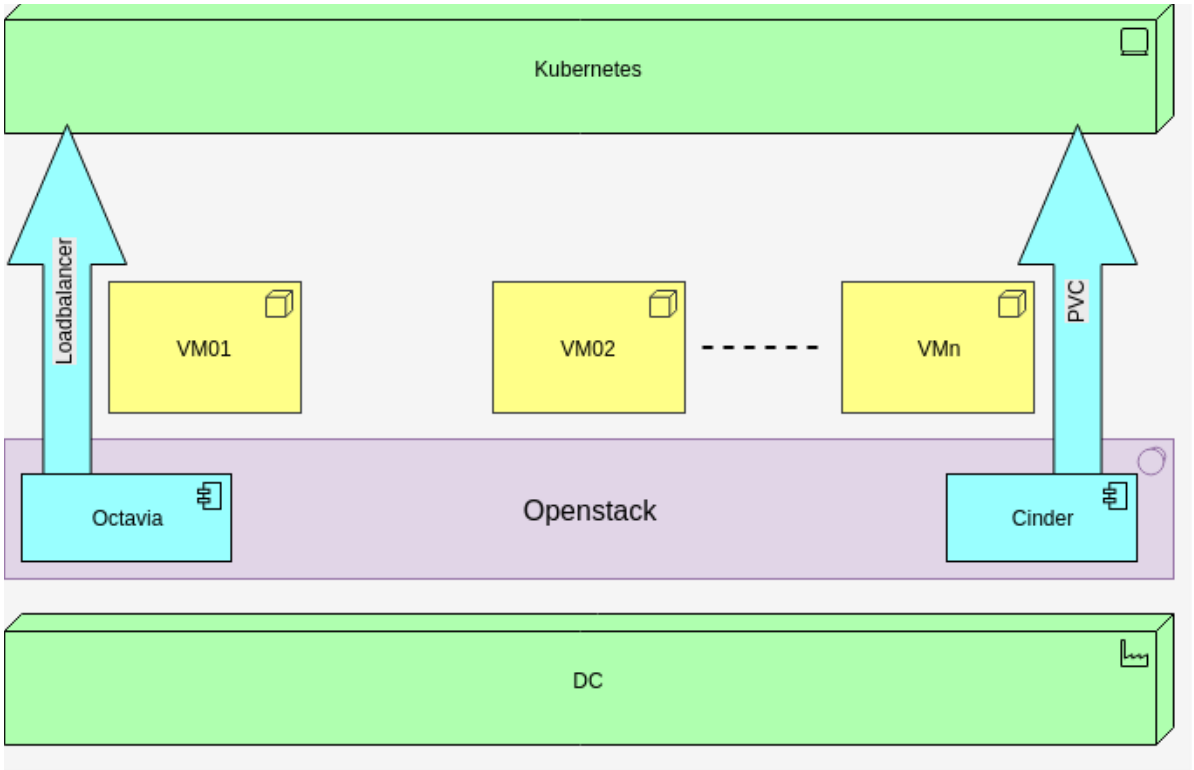
- Kubernetes cluster is installed on 3 of these nodes.
- The 2 other nodes host CI/CD services :
    - Gitlab : Docker registry and Git repositories hosting
    - Jenkins: Pipelines for CI/CD



OPENSTACK CLOUD PROVIDER FOR KUBERNETES

Kubernetes provides automated container orchestration -- management of your machines and services for you -- improving your reliability and reducing the time and resources. In above architecture there are two components of OpenStack which interacts with Kubernetes: Octavia and Cinder which provides loadbalancing and PVC respectively over which virtual machines are created to deploy kubernetes on it. As soon as you have kubernetes you can start deploying applications on it.
Cinder is a persistent block level storage which can be used with openstack compute instances. It manages the creation, attachment and detachment of block devices to the cloud servers. Cinder provides an API to attach/detach volume to Nova instances. With OpenStack as a cloud provider for Kubernetes, volumes are created directly by Kubernetes and automatically attached to the virtual machines of the Kubernetes cluster.

Octavia is an open source, operator-scale load balancing solution designed to work with OpenStack. Octavia accomplishes its delivery of load balancing services by managing a fleet of virtual machines, containers, or bare metal servers which it spins up on demand. Thank to Octavia, Kubernetes can expose services to the outside world using OpenStack floating IPs.



LOGGING, MONITORING AND ALERTING

The Logging, Monitoring, and Alerting toolchain i.e (LMA), is the operational health and response monitoring solution of ITERA for the Kubernetes cluster.

The LMA stacks includes 2 dashboards:

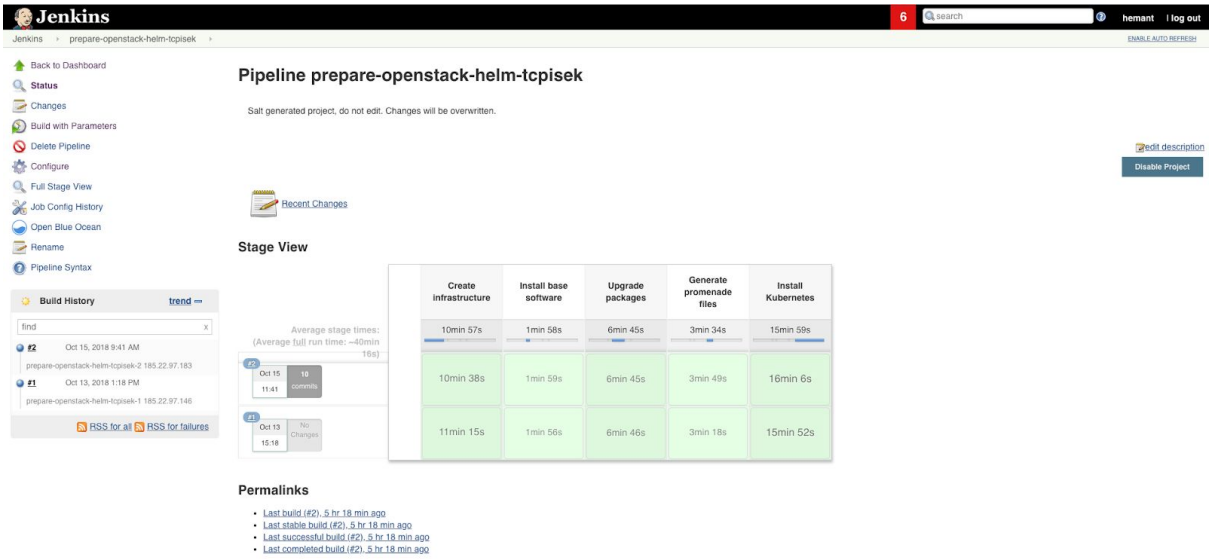
- Grafana allows to monitor your Kubernetes cluster performance. It includes 4 dashboards, Cluster, Node, Pod/Container and Deployment. The metrics collected are high-level cluster and node stats as well as lower level pod and container stats. Use the high-level metrics to alert on and the low-level metrics to troubleshoot.
- Kibana which is the UI companion of Elasticsearch, simplifying visualization and querying. Kibana also has its own dashboard similar like grafana. Kibana core ships with the classics: histograms, line graphs, pie charts, sunbursts, and more. Plus, you can use Vega grammar to design your own visualizations. Kibana developer tools offer powerful ways to help developers interact with the Elastic Stack. With Console, you can bypass using curl from the terminal and tinker with your Elasticsearch data directly

CI/CD

The CI/CD are the pillars of DevOps which improve collaboration between operation and development terms and enable the delivery of high quality software for continued success. A Docker registry is there in gitlab and it can be used for pipelines, kubernetes and many more automation process.

With CI, developers frequently integrate their code into a common repository. Rather than building features in isolation and submitting each of them at the end of the cycle, they continuously build software work products several times on any given day. Every time the code is inputted, the system starts the compilation process, runs unit tests and other quality-related checks as needed. CI relies heavily on test suites and an automated test execution. When done correctly, it enables developers to perform frequent and iterative builds, and deal with bugs early in the lifecycle.

CD executes a progressive set of test suites against every build and alerts the development team in case of a failure, which then rectifies it. In situations where there are no issues, CD conducts tests in a sequential manner. The end result is a build that is deployable and verifiable in an actual production environment. CD executes a progressive set of test suites against every build and alerts the development team in case of a failure, which then rectifies it. In situations where there are no issues, CD conducts tests in a sequential manner. The end result is a build that is deployable and verifiable in an actual production environment.



JENKINS

Jenkins master is running in Docker Swarm on the CI/CD server. Jenkins uses a Master-Slave architecture to manage distributed builds. In this architecture, Master and Slave communicate through TCP/IP protocol.

JENKINS MASTER

The Master's job is to handle:

- Scheduling build jobs.
- Dispatching builds to the slaves for the actual execution.
- Monitor the slaves (possibly taking them online and offline as required).
- Recording and presenting the build results.
- A Master instance of Jenkins can also execute build jobs directly.

JENKINS SLAVE

A Slave is a Java executable that runs on a remote machine. Following are the characteristics of Jenkins Slaves:

- It hears requests from the Jenkins Master instance.
- The job of a Slave is to do as they are told to, which involves executing build jobs dispatched by the Master.
- It is possible to configure a job to always run on a particular Slave machine, or a particular type of Slave machine, or simply let Jenkins pick the next available Slave.

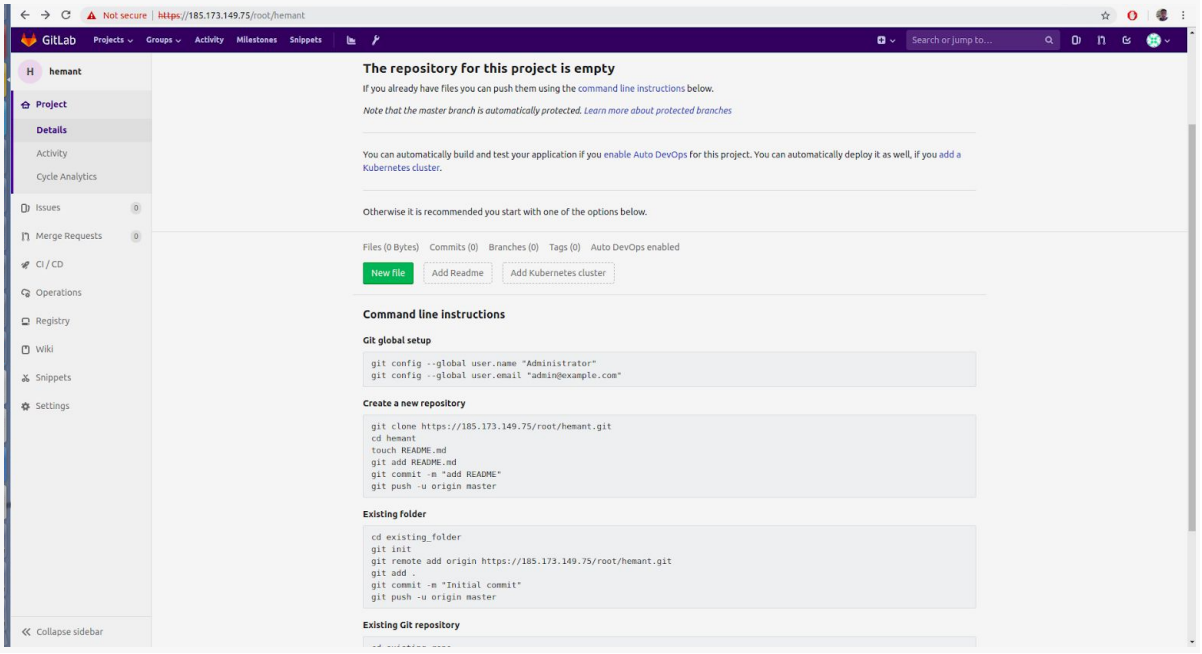
It offers simple way to set up a continuous integration or continuous delivery working environment for almost any kind of coding languages and source code repositories using pipelines as well as other development tools. It can be easily setup and and configure via its web interface. Jenkins integrates almost every tool in CI/CD integration toolchain. It is also possible that jenkins can distribute the work across multiple machine easily. It also helps to build drives, test and deployment across multiple platforms faster : e.g. Docker image build, pipeline for deployment.

GITLAB AND DOCKER REGISTRY

Gitlab is running in a container in Docker Swarm. Gitlab enables teams and team member to collaborate and work on various projects instead of managing multiple threads across disparate tools. It provides single data store, one user interface and one permission model by allowing teams to collaborate with the flexible working environment to focus on software building quickly.

With the help of gitlab it is easy to deploy cloud native applications such as kubernetes. The integration of both Kubernetes and Gitlab makes easy to create and deploy application on the top of Kubernetes.

Gitlab is mainly used to host git repositories. There is also an integrated Docker registry. Users can connect/push/pull images in the Gitlab Docker registry using their Gitlab credentials.



HOW TO DEPLOY APPLICATIONS ON CONTAINER CLOUD

USING KUBECTL AND KUBERNETES API DIRECTLY

Before deploying or running the application on container cloud you need to have kubernetes cluster deployed on the cloud environment. It is also necessary to configure kubectl command-line tool to communicate with kubernetes cluster. But you will already have kubernetes deployed on your cloud environment. So you don't need to be worry for that. As soon as you have your environment ready you can start application deployment process. There are two ways to deploy application on kubernetes cluster:

USING YAML FILE

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.7.9.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Now to create deployment based on YAML file execute following commands in your terminal.

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

To get the information about deployed application execute following command

```
kubectl describe deployment nginx-deployment
```

By executing the above command you will get similar output as given below. The **describe** command allows you to interrogate different kubernetes resources such as pods, deployments, and services at a deeper level.

```
Name:     nginx-deployment
Namespace:    default
CreationTimestamp:  Tue, 30 Aug 2016 18:11:37 -0700
Labels:     app=nginx
Annotations:    deployment.kubernetes.io/revision=1
Selector:   app=nginx
Replicas:   2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds:  0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx:1.7.9
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:   <none>
NewReplicaSet:    nginx-deployment-1771418926 (2/2 replicas created)
No events.
```

To get the status of deployment pods and to delete the deployment from kubernetes cluster execute following commands,

```
kubectl get pods -l app=nginx
```

```
kubectl delete deployment nginx-deployment
```

RUN THE APPLICATION DIRECTLY

The second method of deploying application is very simple and a deployment is a logical reference to pods and their configurations.

Create deployment by using kubectl create command.

```
kubectl create deployment nginx --image=nginx
```

And that's it you deployed an application on kubernetes cluster. To describe and check the status of deployment use commands from previous method 5.1.
Make the deployment accessible via internet.

```
kubectl create service nodeport nginx --tcp=80:80
```

This creates a public facing service on the host for the deployment. Because this is a NodePort service, Kubernetes will assign this service a port on the host machine in the 30000-32767 port range.
It is also necessary to expose the deployment on the internet. Execute following command to expose the deployment.

```
kubectl expose deployment/my-nginx
```

So you are pretty much done with the deployment. Now, it's time to see on which port the service is running. To get the current services,

```
 kubectl get svc
```

You will get the following result by executing the svc command and will get on which port the service is running.

```
nginx        NodePort    10.98.24.29   <none>        80:32555/TCP   52s
```

To see whether the deployment is running on the port just go to the IP of your cloud environment and use exposed port after the IP and there you go you will be redirected to your deployment page. There another way to see whether the deployment is exposed or not just by using curl command

```
curl ip:32555  # ip - ip of cloud environment
```

USING HELM

Helm helps you manage Kubernetes applications. Helm Charts helps you define, install, and upgrade even the most complex Kubernetes application.
Charts are easy to create, version, share, and publish.
Use Helm to:

- Find and use popular software packaged as Helm charts to run in Kubernetes
- Share your own applications as Helm charts
- Create reproducible builds of your Kubernetes applications
- Intelligently manage your Kubernetes manifest files
- Manage releases of Helm packages

The example is given blow about how to install application on kubernetes using helm:

EXAMPLE : INSTALL GOGS USING HELM

Gogs is a painless self-hosted Git service : https://gogs.io/

An helm chart to install gogs exists and will be used to install Gogs in the Kubernetes cluster. The helm chart is not in the stable repository, it is instead located in the incubator.

1. Enable the incubator repository

```
helm repo add incubator
https://kubernetes-charts-incubator.storage.googleapis.com/
```

```
"incubator" has been added to your repositories
```

2. Create a file gogs.yaml containing the following content :

```
persistence:
  storageClass: 'general'
  size: 50Gi
postgresql:
  persistence:
    storageClass: 'general'
    size: 10Gi
```

The storageClass general corresponds to Ceph RBD. The allocated volume for Gogs will be of 50 Gi.

3. Install Gogs with helm

```
helm install --name gogs-signals -f gogs.yaml incubator/gogs
--namespace=testing
```

4. Check the result

```
kubectl get svc -n testing | grep gogs
```

```
gogs-signals-gogs        NodePort   10.96.196.198 <none>
80:31465/TCP,22:32448/TCP
gogs-signals-postgresql  ClusterIP  10.96.118.4   <none>  5432/TCP
```

Gogs was correctly installed on the top of the Kubernetes cluster. It is now possible to access the webui with the port 31465.

PERSISTENT VOLUMES FOR KUBERNETES

The storage class standard was created in the Kubernetes cluster. It will create volumes from Cinder for Kubernetes. The volumes will be automatically created and attached to the virtual machines on the top of which Kubernetes is installed.

Example of PVC for Kubernetes using the storage class standard:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```