Docker Deployment Guide

System Level Guides Current

8/4/2022

Table of Contents

Docker Deployment Guide	3
Installing Docker Engine CE (Community Edition) on CentOS Linux 7	6
Installation of Docker on Alpine Linux	10
Pulling Repositories from Bintray	11
High Availability	13
Blue-Green Deployment Model	21

Docker Deployment Guide

Genesys products are built and deployed with modern development principles and technologies such as Microservices, Docker, DevOps, and Automation. Product development follows the *cloud first* approach using the tools, principles, and methodologies that are applied for cloud services that includes full deployment and upgrade automation, full regression, and a blue-green upgrade model.

This is a generic guide that can be used by the product teams for Docker deployment. The product teams need to follow the instructions provided in this guide as mentioned in each of the chapters. The gist of the instructions is summarized below.

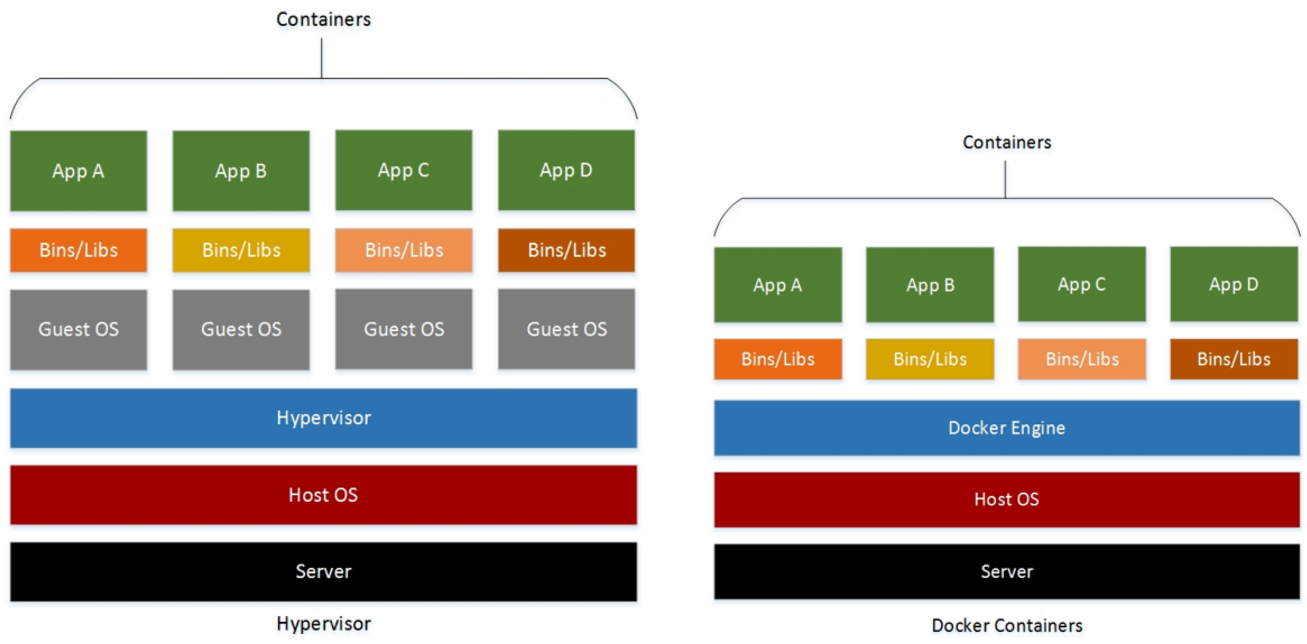
1. Install the Docker Engine CE. For details, refer to Installation of Docker Engine CE.
2. After installing the Docker Engine CE, set up the Docker Engine to use the Bintray Repository provided by Genesys. For details refer to Pulling Repositories from Bintray.
3. Deploy Docker containers in High Availability (HA) models. For details, refer to High Availability.

> **Important**
> The product team decides the HA approach and the orchestration platform to be used.

4. Implement the Blue-Green deployment model. For details, refer to Blue-Green Deployment Model. This topic explains about Blue-Green deployment with Kubernetes.

Introduction to Docker

Docker containers share the OS and kernel of the host system. Therefore, they are easily scalable and lightweight when compared to virtual machines. Each Genesys component is represented as an individual microservice. Each microservice is executed in a Docker container using N+1 horizontal scaling model principles.



Benefits of Product Dockerization

Continuous Deployment and Testing

Docker containers are configured to maintain all configurations and dependencies internally. Therefore, you can use the same container from development to production after ensuring there are no discrepancies or manual intervention.

Environment Standardization and Version Control

Docker containers help developers with easy version control and collaboration by storing the container images in a registry.

Isolation

Docker ensures that each application uses only those resources (CPU, memory, and disk space) that are assigned to it.

Security

Docker ensures that applications running on containers are completely segregated and isolated from each other.

Other Capabilities

Docker containers, when implemented on top of any container orchestration platform (like Kubernetes), will generate the following capabilities:

- Auto-scaling of containers based on demand

- Fault tolerance/self-healing
- High Availability
- Blue-Green deployment of individual containers

Installing Docker Engine CE (Community Edition) on CentOS Linux 7

Prerequisites

OS Requirements

To install Docker CE, the maintained version of CentOS Linux 7 is mandatory. Archived versions are not supported or tested. You must enable the **centos-extras** repository.

> **Important**
> By default, the **centos-extras** repository is enabled. If it is disabled, you must re-enable it.

Uninstall Earlier Versions

Uninstall the earlier versions of Docker/Docker-engine along with associated dependencies using the following commands:

```
$ sudo yum remove docker \

  docker-common\

  docker-selinux \

  docker-engine
```

Though YUM reports that the packages are not installed, the contents available in **/var/lib/docker/** are saved, including images, containers, volumes and networks. The Docker CE package is now changed to **Docker-ce**. Previously, the package name was **Docker**.

Install Docker CE

You can install Docker CE using one of the following methods:

- Install using the repository (most common method)

- Install manually using RPM package
- Install using automated scripts

Install Using the Repository

1. Set up the repository.

   Install required packages. **yum-utils** provides the yum-config-manager utility. The devicemapper storage driver requires the packages **device-mapper-persistent-data** and **lvm2** for installation.

   ```
   $ sudo yum install -y yum-utils \

     device-mapper-persistent-data \

     lvm2
   ```

   Set up the stable repository by using the following command. You need a stable repository, even if you want to install builds from the edge or test repositories.

   ```
   $ sudo yum-config-manager \

     --add-repo \

      https://download.docker.com/linux/centos/docker-ce.repo
   ```

2. Install Docker CE.

   1. Install the latest version of Docker CE.

      ```
      $ sudo yum install docker-ce
      ```

      If you are installing a package from a recently added repository for the first time, you will be prompted to accept the GPG key, and the key's fingerprint will be displayed. Check if the fingerprint is correct. If the fingerprint is correct, accept the key. The fingerprint should match 060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35.

      Docker is now installed, but not started. The Docker group is created. However, no users are added to the group.

      > **Important**
      >
      > If you receive an error message stating that Parallel gzip (pigz) or container-selinux is not installed, you must install them. The following example provides commands on how to install pigz and container-selinux from the repository. These commands are used for installing pigz version 2.3.4-1.e17 and container-selinux version 2.21-1.el7, which are the latest versions. Please visit the repository URLS http://dl.fedoraproject.org/pub/epel/7/x86_64/Packages/p/ for pigz, and http://mirror.centos.org/centos/7/extras/x86_64/Packages/ for container-selinux to view the latest versions of both the packages.

      Command to install pigz

      ```
      yum install http://dl.fedoraproject.org/pub/epel/7/x86_64/Packages/p/
      pigz-2.3.4-1.el7.x86_64.rpm
      ```

      Command to install container-selinux

      ```
      yum install http://mirror.centos.org/centos/7/extras/x86_64/Packages/container-
      ```

```
selinux-2.21-1.el7.noarch.rpm
```

Re-run the Docker installation after installing pigz and container-selinux.

2. Install a specific version of Docker CE.

   ```
   yum list docker-ce --showduplicates | sort -r
   ```

   ```
   docker-ce.x86_64  17.09.ce-1.el7.centos  docker-ce-stable
   ```

   The contents of the list depend on the repositories that are enabled. These contents will be specific to your version of CentOS (indicated by the .el7 suffix on the version in the preceding example). Select a specific version to install. The second column is the version string. You can use the entire version string. You must include at least to the first hyphen. The third column is the repository name, which indicates which repository the package is from and by extension its stability level. To install a specific version, append the version string to the package name and separate them by a hyphen (-).

   > **Important**
   >
   > The version string is the package name including the version up to the first hyphen. In the preceding example, the fully qualified package name is **docker-ce-17.06.1.ce**.

   ```
   $ sudo yum install <FULLY-QUALIFIED-PACKAGE-NAME>
   ```

3. Start Docker.

   ```
   $ sudo systemctl start docker
   ```

4. Run the hello-world image to ensure Docker is installed correctly.

   ```
   sudo docker run hello-world
   ```

   This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Install Manually Using the RPM Package

If you cannot use Docker's repository to install Docker, you can download the .rpm file and install it manually. You will need to download a new file each time you want to upgrade Docker.

1. Go to https://download.docker.com/linux/centos/7/x86_64/stable/Packages/ and download the .rpm file for the Docker version that you want to install.

   > **Important**
   >
   > To install an edge package, change the word *stable* in the above URL to the word *edge*.

2. Install Docker CE, changing the path below to the path where you downloaded the Docker package.

   ```
   $ sudo yum install /path/to/package.rpm
   ```

   Docker is now installed, but not started. The Docker group is created, but no users are added to the group.

3. Start Docker.
   ```
   $ sudo systemctl start docker
   ```

4. Run the hello-world image to ensure Docker is installed correctly.
   ```
   sudo docker run hello-world
   ```

Docker CE is now installed and running. You need to use sudo to run the Docker commands. For more information about the installation of Docker engine, refer to Docker documentation: https://docs.docker.com/engine/installation/linux/docker-ce/centos.

Install Using Automated Scripts

Docker provides scripts at https://get.docker.com/ and https://test.docker.com/ for installing edge and testing versions of Docker CE in development environments, quickly and non-interactively. The source code for these scripts is available at https://github.com/docker/docker-install. However, using these scripts is not recommended for production environments.

Installation of Docker on Alpine Linux

To install Docker on Alpine Linux, follow these steps:

1. To install Docker on Alpine Linux, run `apk add --update docker openrc`.

> **Important**
>
> The Docker package is available in the Community repository. Therefore, if `apk add` fails because of `unsatisfiable constraints` error, you need to edit the **/etc/apk/repositories** file to add (or uncomment) a line. Community repository link: http://dl-cdn.alpinelinux.org/alpine/latest-stable/community.

2. To start the Docker daemon at boot, run `rc-update add docker boot`.
3. To start the Docker daemon manually, run `service docker start`.
4. Execute `service docker status` to ensure the status is **running**.

For more details about installing Docker on Alpine, refer to https://wiki.alpinelinux.org/wiki/Docker#Installation.

Pulling Repositories from Bintray

Warning
Genesys no longer supports Bintray.

Bintray is a fully-fledged repository for your Docker images, and works seamlessly with the Docker client. You must point to your Docker client in Bintray, and issue Docker commands to perform the following operations:

- **Pull** — pull an image from a Docker repository in Bintray
- **Search** — search for images within a Docker repository in Bintray

After the Docker images are stored in Bintray, you can use the Bintray user interface to browse through them. To obtain code snippets that are formulated for a specific username and repository, use the **Set Me Up** button.

Terminology

Some of the terminology used in Docker and Bintray may be the same, but may not have the same meaning.

Examples:

- A Docker “tag” translates into a Bintray “version”
- A Docker “namespace: repository” translates into a Bintray “package”

Configure Docker

Genesys will provide a username and password for Bintray. You can use this username and password to Search and Pull repositories.

Execute Docker Commands in Bintray

You can use Bintray as a Docker registry for Pull and Search operations. The registry URL includes your username (or organization name), and must be in the following format:

```
{subject}-docker-{repo}.bintray.io/[{namespace}/]{docker_repo}[:{tag}]
```

Values:

- **{subject}** — the user or organization name
- **{repo}** — the Bintray repository name
- **{namespace}/{docker_repo}** — {namespace} is the Docker namespace you have applied. This is used as the prefix for the Bintray package name (Default: library)
- **{namespace}:{docker_repo}** — Bintray package
- **{tag}** — the Docker tag (Default: latest)

Pull

The format for Docker Pull command is:

```
docker push/pull {subject}-docker-
{repo}.bintray.io/[{namespace}/]{docker_repo}[:{version}]
```

Example:

```
docker pull genesyssoftware-docker-premisetest.bintray.io/genesys/lca
```

> **Important**
> To push to a repository, you must configure Docker with your Bintray credentials. This ensures that only the repository owner can push to it.

When pulling from a Bintray repository, you must configure Docker with your Bintray credentials if you are pulling from a private repository. Pulling from a public repository can be performed anonymously. Therefore, it does not require a corresponding `~/.dockercfg` entry.

Search

The format for Docker search command is:

```
docker search {subject}-docker-{repo}.bintray.io/{search-string}
```

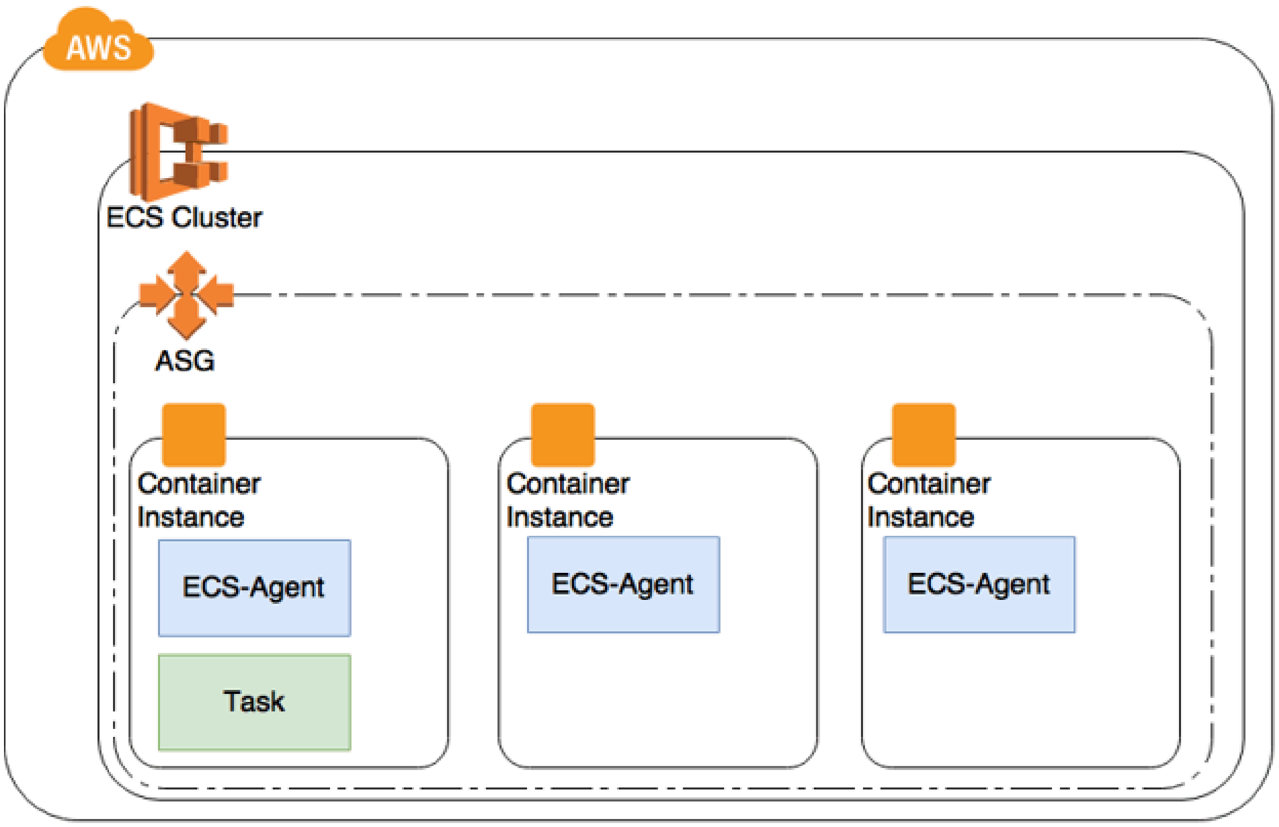
Example:

```
docker search genesyssoftware-docker-premisetest.bintray.io/genesys
```

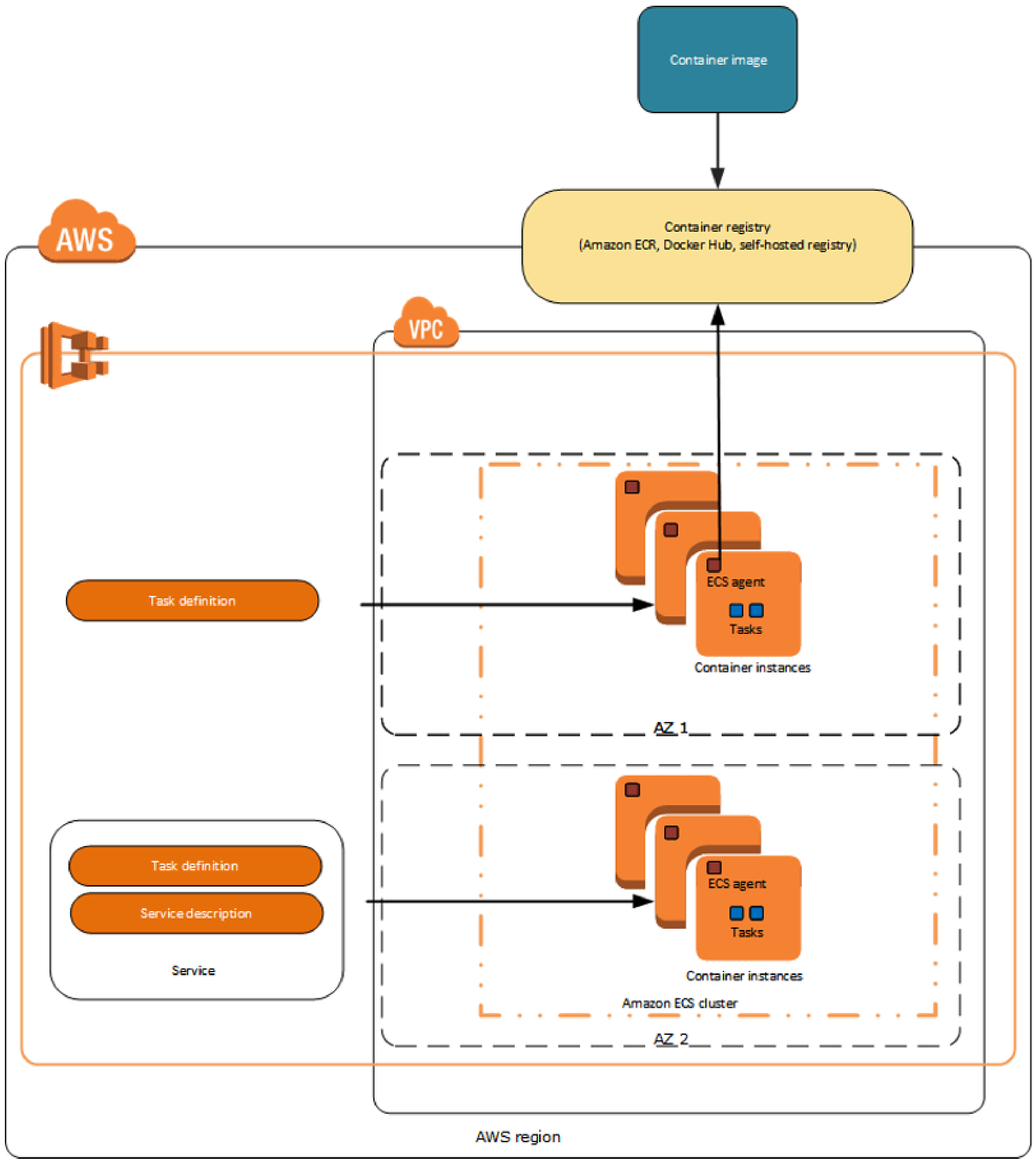
High Availability

High Availability with Amazon Web Service (AWS)

Each Genesys product component is ensured high availability by implementing active-active configuration. If you host the product on public cloud (such as AWS), the container cluster is deployed within Auto Scaling Group (ASG). This ensures that a minimum number of specified containers run to serve the traffic.

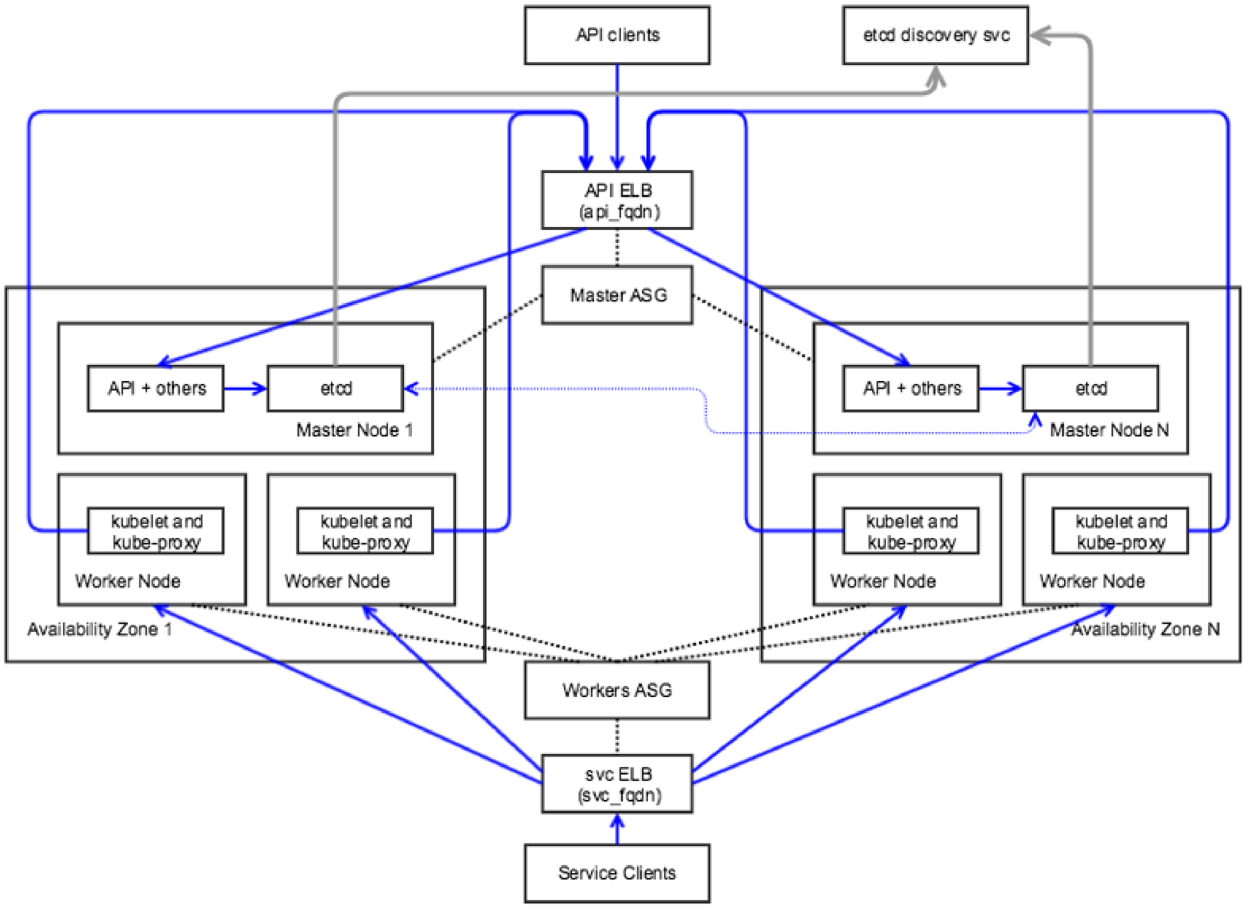


ASG is also deployed across multiple Availability Zones (AZ) to ensure availability in case of zone failure.


Container image
Container registry
(Amazon ECR, Docker Hub, self-hosted registry)
AWS
VPC
Task definition
ECS agent
Tasks
Container instances
AZ 1
Task definition
Service description
Service
ECS agent
Tasks
Container instances
Amazon ECS cluster
AZ 2
AWS region

High Availability with Kubernetes

You can deploy a Genesys product in a private cloud or on premise. During such deployment scenarios, containers are implemented on container orchestration platforms such as Kubernetes.



Kubernetes clusters enable a higher level of abstraction to deploy and manage a group of containers that comprise microservices in a cloud-native application. A Kubernetes cluster provides a single Kubernetes API entry point, cluster-wide resource naming scheme, placement engine and scheduler for pods, service network routing domain, and authentication and authorization model.

Kubernetes can be deployed in the following scenarios:

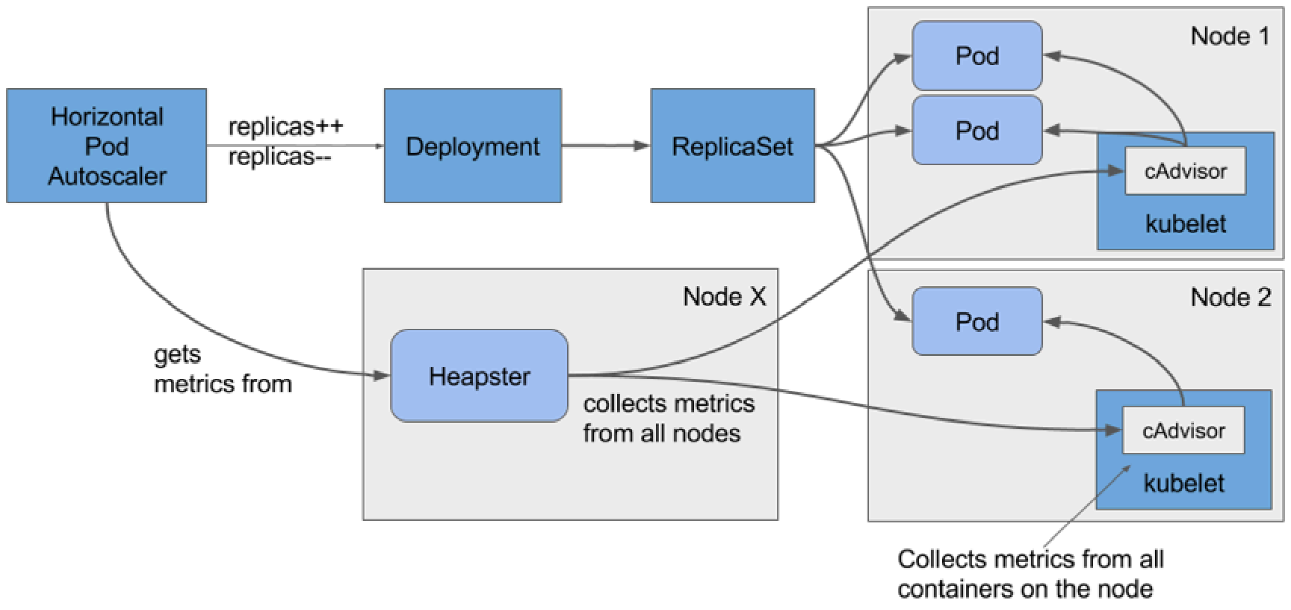
- Across multiple availability zones, but within a single cloud provider — Cloud provider services like storage, load balancers, network routing may interoperate easily.
- In the same geographical region — Network performance will be fast enough to act like a single data center.
- Across multiple geographical regions — High network cost and poor network performance may be prohibitive.
- Across multiple cloud providers with incompatible services and limited interoperability — Inter-cluster

networking will be complex and expensive to set up in an efficient way.

Container Orchestration with Kubernetes

Auto-scaling with Kubernetes

Kubernetes pods can be auto-scaled up or down based on metrics such as CPU and memory utilization. Kubernetes implements Horizontal Pod Autoscaler (HPA) to achieve auto-scaling. HPA collects metrics about pods from Heapster. Based on the auto-scaling rule, HPA either increases or decreases the number of pods through the deployment object created for the Genesys microservice. The following diagram illustrates the auto-scaling architecture with HPA and Heapster.



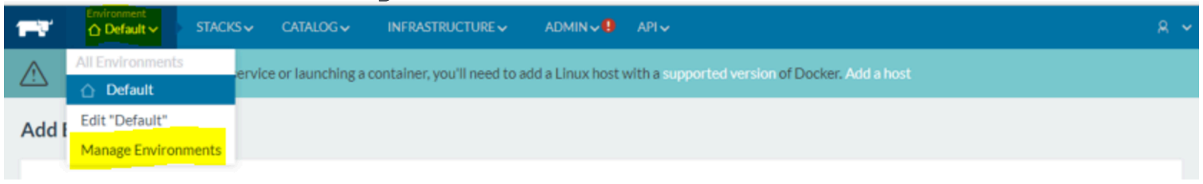
Kubernetes Setup with Rancher

Rancher provides an easier process to set up a Kubernetes cluster either on cloud or on premise. It hides the complex steps involved in setting up the master and worker nodes. Rancher is recommended for a push button setup of Kubernetes cluster for development and QA purposes. However, it is not recommended for production use. If you have any queries regarding Rancher, contact the Genesys Engage Architecture team (**pureengagearchitecture@genesyslab.onmicrosoft.com**).

To set up a single node Kubernetes cluster with Rancher, follow these steps:

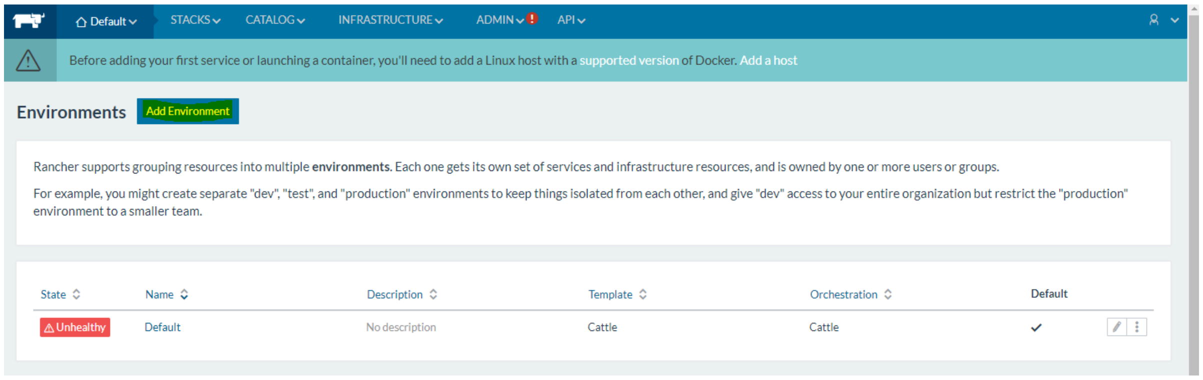
1. Start Rancher Server by executing the following command.
   ```
   sudo docker run -d --restart=unless-stopped -p 8080:8080 rancher/server
   ```
2. Open the Rancher console using **<ip>:8080** on the browser.

3. On the **Default** menu, select **Manage Environments**.



ervice or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host

4. Click **Add Environment**.



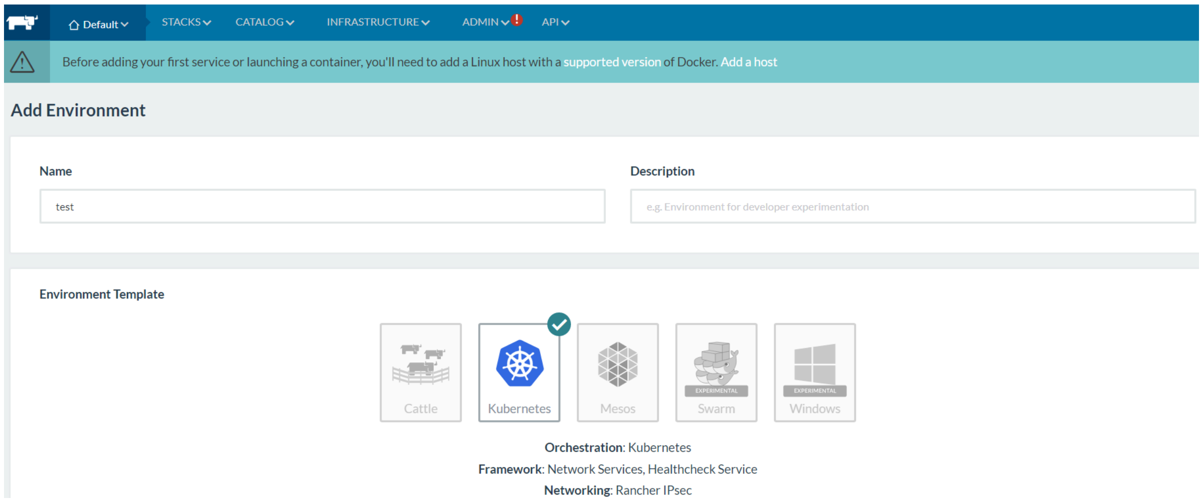
Before adding your first service or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host

Rancher supports grouping resources into multiple **environments**. Each one gets its own set of services and infrastructure resources, and is owned by one or more users or groups.

For example, you might create separate "dev", "test", and "production" environments to keep things isolated from each other, and give "dev" access to your entire organization but restrict the "production" environment to a smaller team.

State	Name	Description	Template	Orchestration	Default
Unhealthy	Default	No description	Cattle	Cattle	✓

5. On the screen displayed, create a new environment by providing a name in the **Name** field. Then, select **Kubernetes** as the orchestration for the new environment.



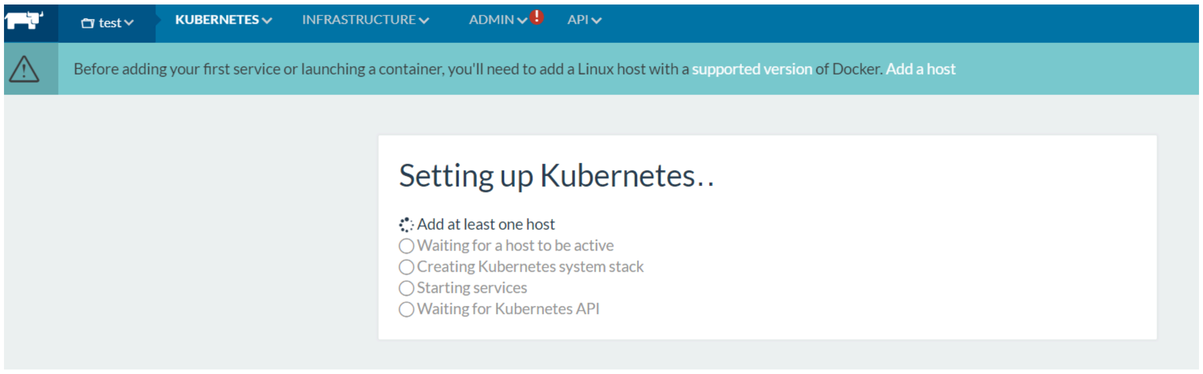
Before adding your first service or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host

Name: test

Description: e.g. Environment for developer experimentation

Orchestration: Kubernetes
Framework: Network Services, Healthcheck Service
Networking: Rancher IPsec

This starts setting up Kubernetes.

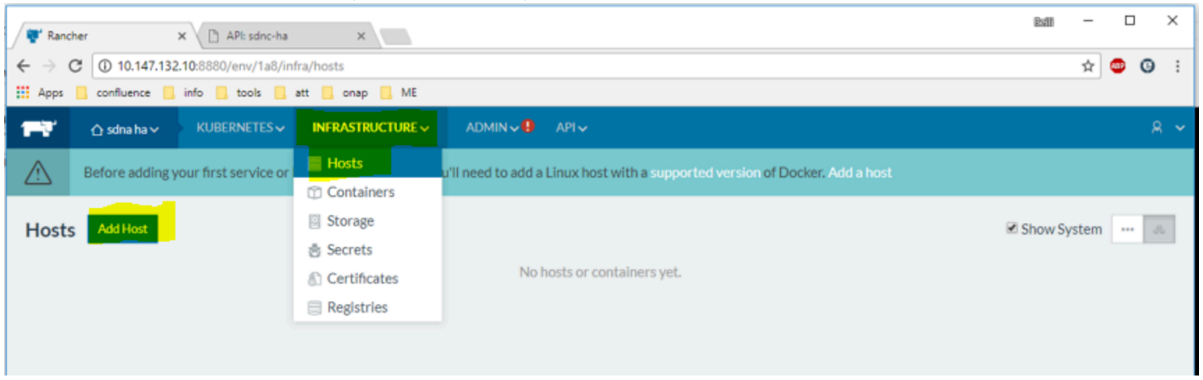


Before adding your first service or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host

Setting up Kubernetes..

Add at least one host
Waiting for a host to be active
Creating Kubernetes system stack
Starting services
Waiting for Kubernetes API

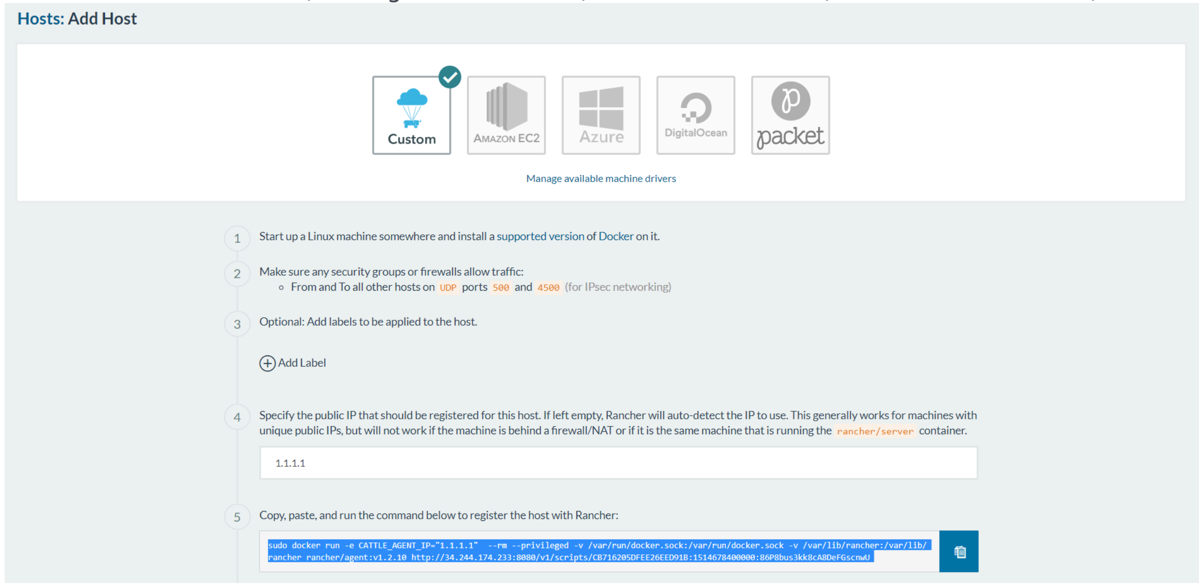
6. Complete the wizard.
7. On the **INFRASTRUCTURE** menu, select **Hosts**, and then click **Add Host**.



Before adding your first service or ...u'll need to add a Linux host with a supported version of Docker. Add a host

No hosts or containers yet.

8. Enter the IP of the same host (for single node cluster) or a different host (for multi-node cluster).



1 Start up a Linux machine somewhere and install a supported version of Docker on it.

2 Make sure any security groups or firewalls allow traffic:
- From and To all other hosts on `UDP` ports `500` and `4500` (for IPsec networking)

3 Optional: Add labels to be applied to the host.

4 Specify the public IP that should be registered for this host. If left empty, Rancher will auto-detect the IP to use. This generally works for machines with unique public IPs, but will not work if the machine is behind a firewall/NAT or if it is the same machine that is running the `rancher/server` container.

1.1.1.1

5 Copy, paste, and run the command below to register the host with Rancher:

```
sudo docker run -e CATTLE_AGENT_IP="1.1.1.1"  --rm --privileged -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/rancher:/var/lib/
rancher rancher/agent:v1.2.10 http://34.244.174.233:8080/v1/scripts/CB716205DFEE26EED91B:1514678400000:86P8bus3kk8cA8DeFGscnwU
```

9. Execute the command generated by the wizard on the node to be added.
10. Watch the status of the Kubernetes cluster getting initialized and the services getting started.
11. Navigate to the cluster environment and select **CLI**. The command line interface that interacts with the

cluster using `Kubectl` command will be accessible from the browser.

12. To interact with the cluster from a different computer, copy the config file that Rancher generates and paste it on the **~/.kube/config** file. Create the config file if it is not available.

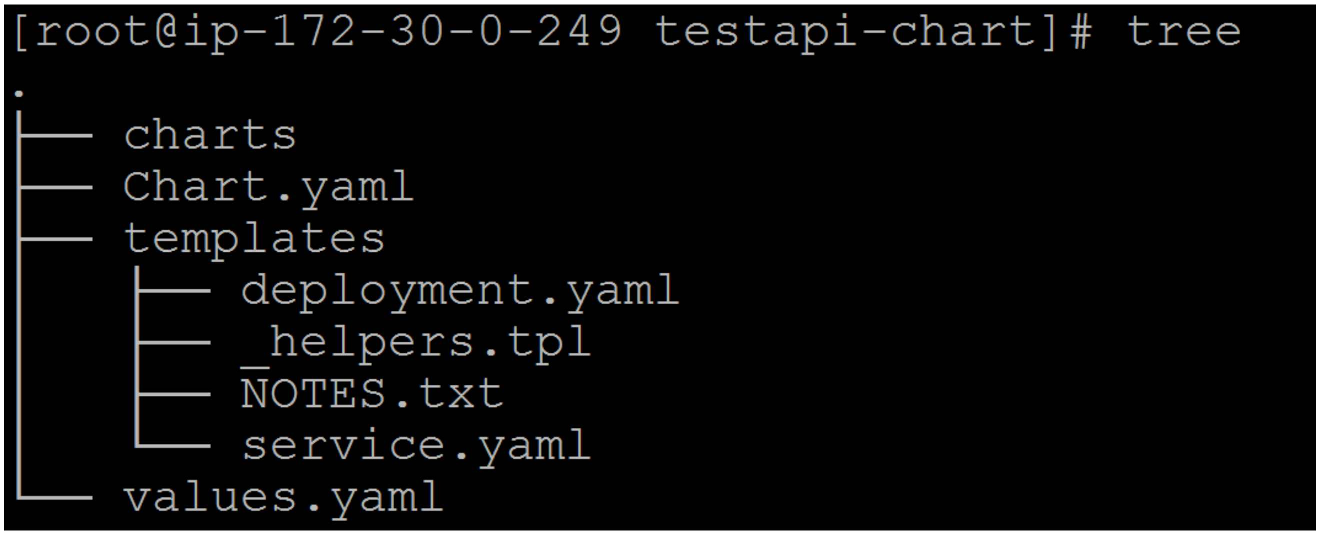
Kubernetes Helm for Package Management

Helm is a package manager for Kubernetes. Helm packages multiple K8s resources into a single logical deployment unit called Chart. Helm is also a deployment manager for Kubernetes. It helps in:

- Performing repeatable deployments
- Managing (reusing and sharing) dependencies
- Managing multiple configurations
- Updating, rolling back, and testing application deployments (releases)

Here are the components of a typical Helm environment:

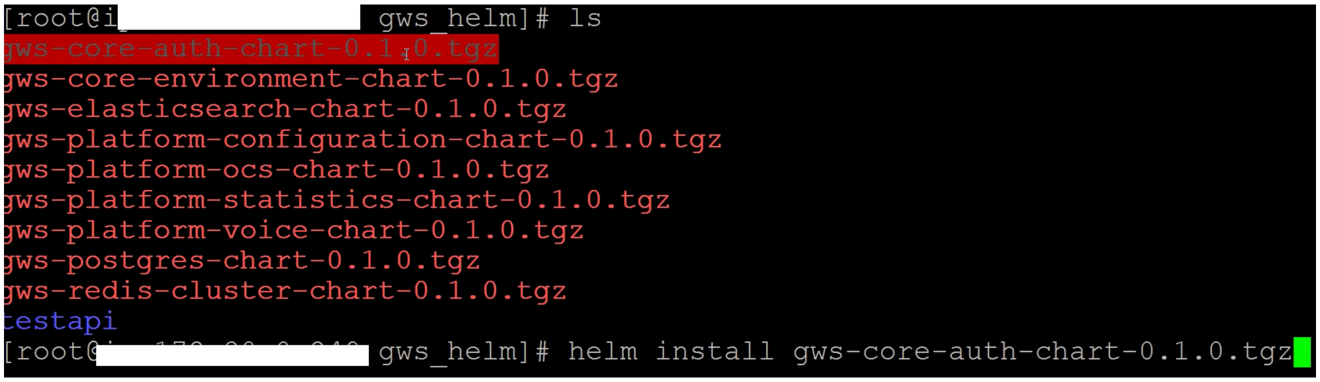
- **Chart** - a package; bundle of Kubernetes resources
- **Release** - a chart instance that is loaded into Kubernetes
- **Repository** - a repository of published charts
- **Template** - a K8s configuration file with Go template



```
[root@ip-172-30-0-249 testapi-chart]# tree
.
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

The Genesys Kubernetes manifests are packaged as Helm charts. The charts are in .tgz format. You can download the charts from the Helm repository that will be created for each project, and then install them. The following example illustrates the steps to be followed while installing a chart from the repository.

1. Download the chart from the **git** repository using the `wget` command.
2. Install the chart using the `helm install` command as shown in the following image.



```
[root@i[redacted] gws_helm]# ls
gws-core-auth-chart-0.1.0.tgz
gws-core-environment-chart-0.1.0.tgz
gws-elasticsearch-chart-0.1.0.tgz
gws-platform-configuration-chart-0.1.0.tgz
gws-platform-ocs-chart-0.1.0.tgz
gws-platform-statistics-chart-0.1.0.tgz
gws-platform-voice-chart-0.1.0.tgz
gws-postgres-chart-0.1.0.tgz
gws-redis-cluster-chart-0.1.0.tgz
testapi
[root@[redacted] gws_helm]# helm install gws-core-auth-chart-0.1.0.tgz
```

For detailed information about Helm installation, refer to https://github.com/kubernetes/helm/.

Blue-Green Deployment Model

Overview

Typically, deploying a new release replaces the current one. You stop the previous release, and then replace it with the new release. The problem with this approach is the downtime that occurs from the moment the previous release is stopped till the new one is fully operational. The Blue-green process removes the deployment downtime and also reduces the risk that the deployment might introduce.

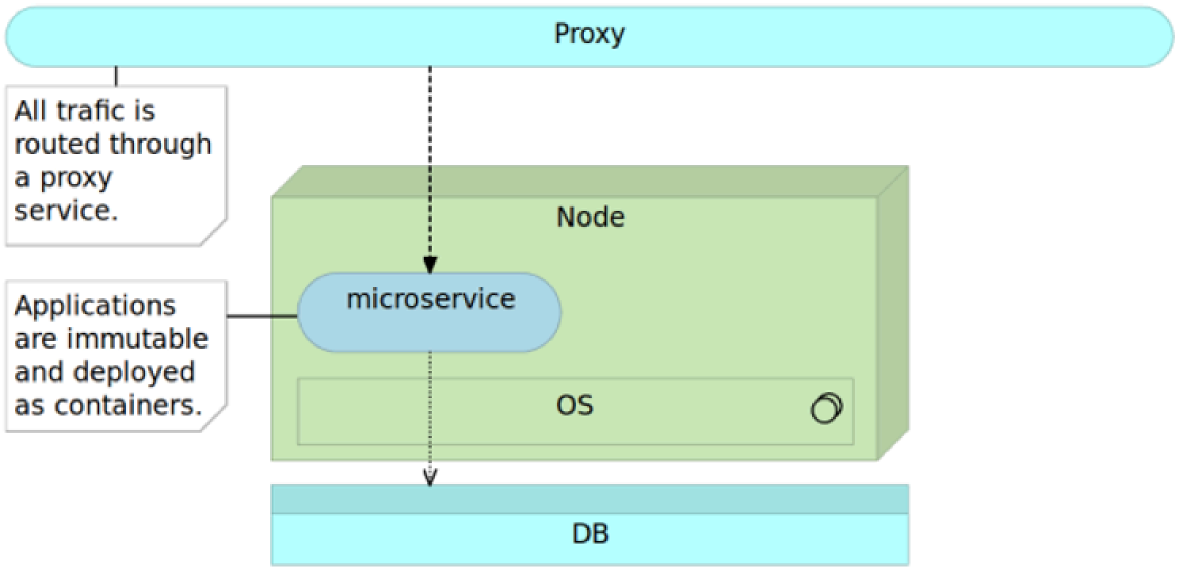
The Blue-Green Deployment Process

The Blue-Green deployment procedure, when applied to microservices packed as containers, is as follows.

Important

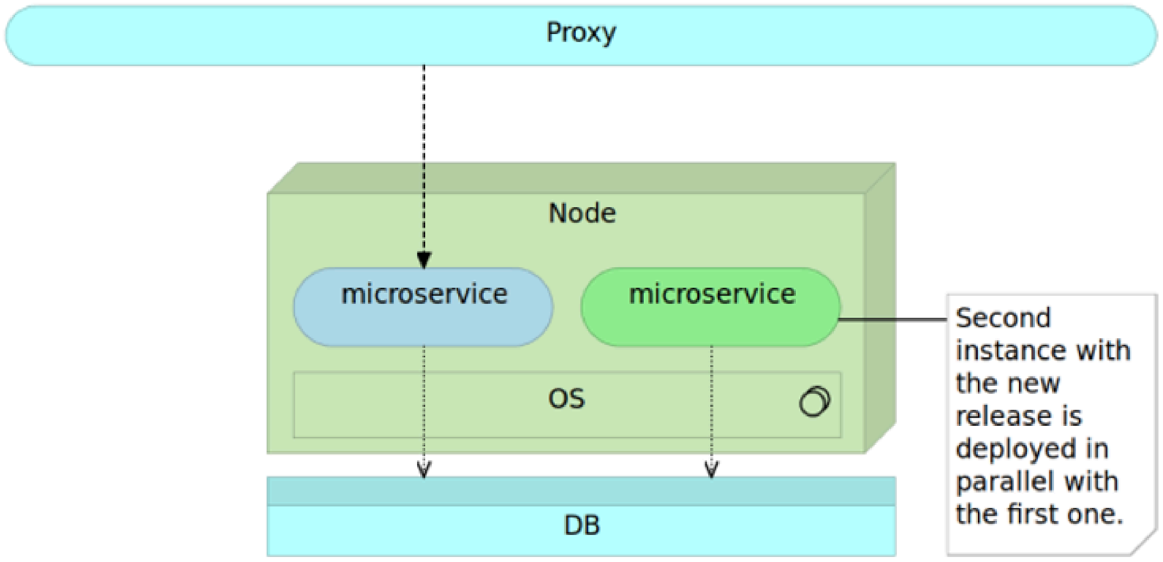
This example is limited only to microservices and not to the database.

1. When the current release (for example, blue) is running on the server, route all traffic to that release through a proxy service. Microservices are immutable and deployed as containers.

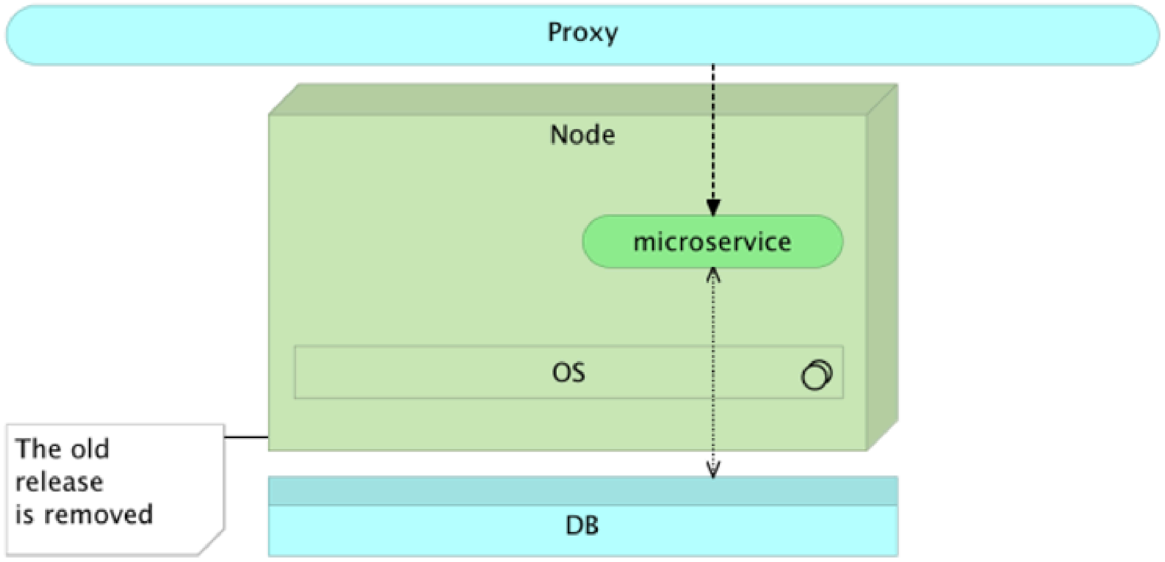

Proxy
All trafic is routed through a proxy service.
Node
Applications are immutable and deployed as containers.
microservice
OS
DB

2. When a new release (for example, green) is ready to be deployed, run it in parallel with the current release. This way, you can test the new release without affecting the users since all the traffic

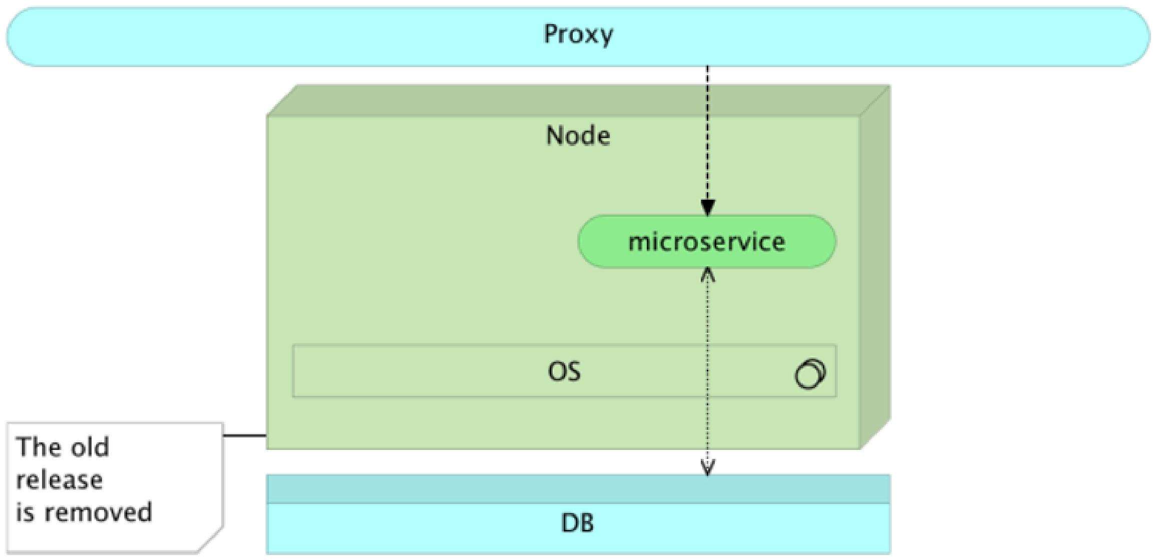
continues to be sent to the current release.


Proxy
Node
microservice
microservice
OS
DB
Second instance with the new release is deployed in parallel with the first one.

3. Once the new release works as expected, change the proxy service configuration to redirect the traffic to the new release. Most of the proxy services will allow the existing requests to complete the execution using the previous proxy configuration to ensure there is no interruption.

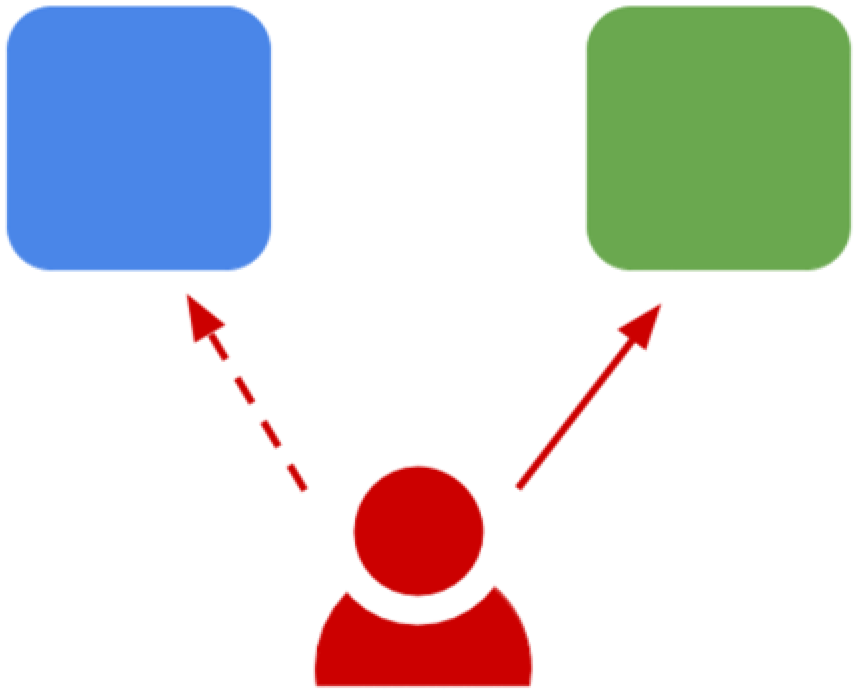


4. When all the requests sent to the previous release receive responses, you can remove or stop the previous version of a service. When you stop the previous version of a service from running, a rollback in case of a failure of the new release will be the instantaneous action so that you can back up the previous release.



Sample Blue-Green Deployment on Kubernetes

When blue-green deployments are performed, a new copy of the application (green) is deployed along with the existing version (blue). The ingress/router to the app is updated to switch to the new version (green). You must wait for the previous (blue) version to complete the requests sent to it. However, for the most part, traffic to the app changes to the new version, instantly.



Kubernetes does not contain built-in support for blue-green deployments. Currently, the best way to support deployments is to create new deployment, and then update the service for the application to point to the new deployment. This section contains a sample blue-green deployment implemented on a Kubernetes cluster.

The Blue Deployment

A Kubernetes deployment specifies a group of instances of an application. At the back end, it creates a replicaset that is responsible for keeping the specified number of instances up and running.

You can create your blue deployment by saving the following `yaml` to a `blue.yaml` file.

```
apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: nginx-1.10

spec:

  replicas: 3

  template:
   metadata:
    labels:
     name: nginx
     version: "1.10"
   spec:
    containers:
     - name: nginx
       image: nginx:1.10
       ports:
        - name: http
          containerPort: 80
```

Create the deployment using the `kubectl` command:

```
$ kubectl apply -f blue.yaml
```

After the deployment, you can provide a way to access the instances of the deployment by creating a Service. Services are decoupled from deployments, which means that you do not explicitly point a service at a deployment. Instead, you specify a label selector that is used to list the pods that make up the service. When using deployments, this is typically set up so that it matches the pods for a deployment.

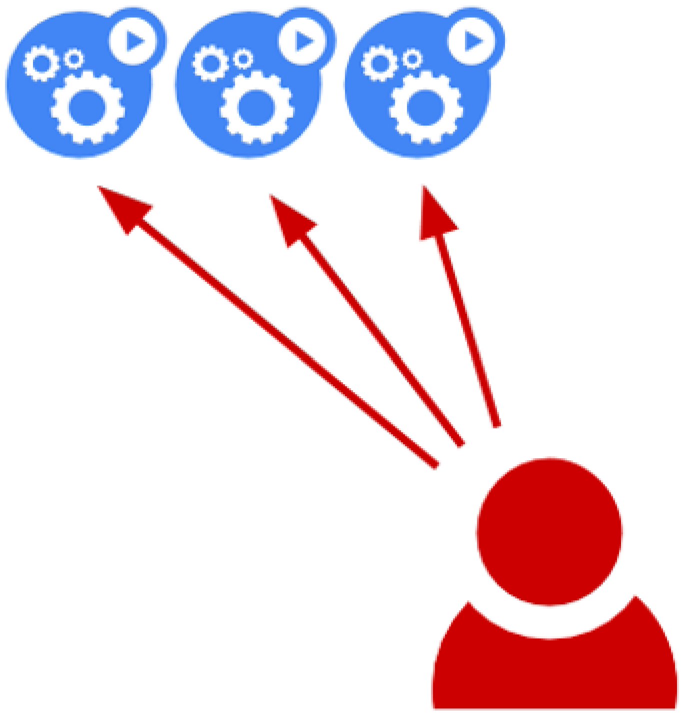
In this case, you have two labels, `name=nginx` and `version=1.10`. You will set them as the label selector for the following service. Save this to `service.yaml`.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels
   name: nginx
```

```
specs:
  ports
   - name: http
     Port: 80
     targetPort: 80
  selector
   name: nginx
   version: "1.10"
  type: LoadBalancer
```

Creating the service creates a load balancer that is accessible outside the cluster.

```
$ kubectl apply -f service.yaml
```



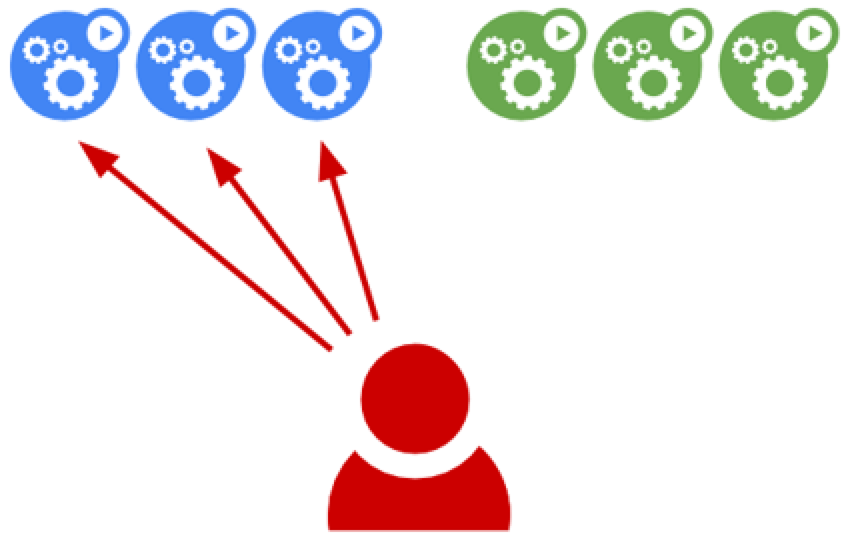
The Green Deployment

For the green deployment, you will perform a new deployment in parallel with the "blue" deployment. The following service is saved as `green.yaml`.

```
apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: nginx-1.11

spec:

  replicas: 3

  template:

    metadata:

     labels:

      name: nginx

      version: "1.11"

    spec:

     containers:

      - name: nginx

      image: nginx:1.11

      ports:

       - name: http

        containerPort: 80
```

```
$ kubectl apply -f green.yaml
```

Now there are two deployments, but the service still points to the "blue" one.



The Cut-Over

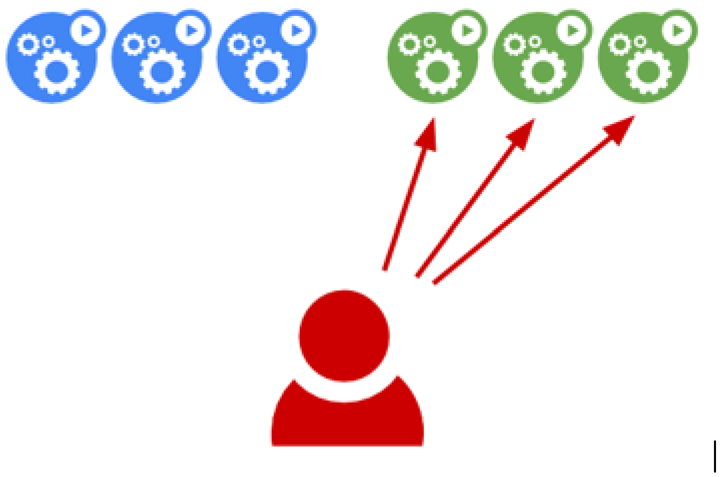
To cut over to the "green" deployment, you must update the selector for the service. Edit `service.yaml` and then change the selector version to "1.11". This matches the pods on the "green" deployment.

```
apiVersion: v1

kind: Service

metadata:

  name: nginx

  labels:

   name: nginx

spec:

  ports:

   - name: http

    port: 80

    pargetPort: 80

  selector:
  name: nginx
  version: "1.11"
  type: LoadBalancer
```

The following **apply** command will update the existing nginx service.

```
$ kubectl apply -f service.yaml
```

Now the service appears as follows:



Updating the selector for the service is applied immediately. Therefore, you can see that the new version of nginx will be serving the traffic.

```
$ EXTERNAL_IP=$(kubectl get svc nginx -o
jsonpath="{.status.loadBalancer.ingress[*].ip}")
$ curl -s http://$EXTERNAL_IP/version | grep nginx
```

Terminating and Deleting Blue Deployment

The Blue deployment will stop receiving further requests because the service points to the Green deployment. However, it is necessary that the Blue deployment is terminated gracefully after it terminates the current serving connections.

This can be done using one of the following methods depending on the scenario:

- Terminate with grace-period
- Use preStop hook

Terminate with grace-period can be done by specifying a grace period while deleting the deployment:

```
kubectl delete deployment blue --grace-period=120
```

The grace-period setting can also be specified at the pod spec level as follows:

```
apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: test

spec:

  replicas: 1

  template:

   spec:

    containers:

     - name: test

      image: ...

  terminationGracePeriodSeconds: 60
```

The **preStop hook** is configured at the container level and allows execution of a custom command before SIGTERM (signal sent to a running process to end the process) is sent. The termination grace period countdown starts before invoking the preStop hook and not after the SIGTERM signal is sent.

The following example, shows how to configure a preStop command:

```
apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: nginx/tt>

spec:
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    lifecycle:
     preStop:
      exec:
       # SIGTERM triggers a quick exit; gracefully terminate
instead
       command: ["/usr/sbin/nginx","-s","quit"]
```

Automating

You can automate the blue/green deployment to some extent with scripting. The following script uses the name of the service, version you want to deploy, and path to the green deployment's **yaml** file. Then, it runs through a full blue/green deployment process using **kubectl** to send a raw JSON as the output from the API and parsing it with the **jq** script. It waits for the green deployment to become ready by inspecting **status.conditions** on the deployment object before updating the service definition.

```
#!/bin/bash

# bg-deploy.sh <servicename> <version> <green-deployment.yaml> # Deployment name
should be <service>-<version>

DEPLOYMENTNAME=$1-$2 SERVICE=$1 VERSION=$2 DEPLOYMENTFILE=$3

kubectl apply -f $DEPLOYMENTFILE
```

```
# Wait until the Deployment is ready by checking the MinimumReplicasAvailable
condition. READY=$(kubectl get deploy $DEPLOYMENTNAME -o json | jq
'.status.conditions[] | select(.reason == "MinimumReplicasAvailable") | .status' | tr
-d '"') while [[ "$READY" != "True" ]]; do

     READY=$(kubectl get deploy $DEPLOYMENTNAME -o json | jq
   '.status.conditions[] | select(.reason ==
   "MinimumReplicasAvailable") | .status' | tr -d '"')

     sleep 5

done

# Update the service selector with the new version kubectl patch svc $SERVICE -p
"{\"spec\":{\"selector\": {\"name\": \"${SERVICE}\", \"version\": \"${VERSION}\"}}}"

echo "Done."
```