Unit-3	**Containers and Kubernetes**	**CONTACT HOURS:4 hours**
	Container ecosystem, Kubernetes, Container, orchestration, Kubernetes architecture, Master Node Components, Worker Node Components, Kubernetes Building Blocks, Images, Immutability, Pod, Config Maps & Secrets, Deploying Applications on Kubernetes, Pod Health Checking, Kubectl Commands, Cloud Application Component Architecture, Benefits of using Kubernetes with IBM Containers, About Microservices ,monolithic application, microservice security, api management and gateways, the future of microservices, microservices governance	

COURSE OUTCOMES

On completion of this course, the students shall be able to

COs	**Statements**	**Bloom's Level**
CO5	Cloud implementation and managing Kubernetes and clusters	L5

UNIT-3: CONTAINERS AND KUBERNETES

CONTAINER ECOSYSTEM

Containerisation has been a big technology over the past couple of years.

Developers rank Docker as the leading technology for containerization, and it’s considered a “game-changer” in DevOps.

Many large companies use Docker containers to manage their infrastructures, including Airbnb, Google, IBM, Microsoft, Amazon, and Nokia.

In the few years since its inception, the Docker ecosystem has quickly become the de-facto standard for managing containerized applications.

For those new to the Docker ecosystem, it can be daunting to understand how it works and how to evolve with it.

VIRTUAL MACHINES

Over the last decade we've had the rise of Virtual Machines (VMs) for application deployment and management.

This is essentially one physical computer called the *host* which has an OS and some sort of VM management layer.

Then ontop of the management layer we have a bunch of isolated virtual computers each with it's own OS and networking.

We are free to buy huge machines and split them up into VM's so we can reap as much of the host resources as possible, while maintaining isolation within our applications.

VM's have been the dominant technology for most sys admins / ops teams managing large application stacks.

If you work in a startup or a smaller business, you are probably not even aware of the host machine very often.

Platforms like AWS, Azure, and GCP abstract this from you.

Instead you can simply request a VM with a particular set of specs.

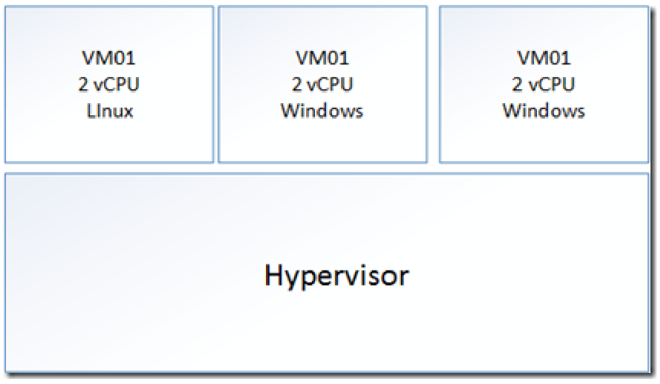
CONTAINERS

So the easiest way to describe a container is how to differs from a virtual machine.

When we setup a virtual machine we need a hypervisor which supports virtualization using certain CPU instructions.

Then we define a set of virtual hardware (CPU, Memory, Harddisk) etc.

Then we need to install a Guest Operating system on the virtual machine or have some form of guest OS running, so the guest OS is not sharing any of the underlaying operating system running on the hypervisor in fact it is unaware of it.



So even if we have a hypervisor running Microsoft Hyper-V, we can still setup virtual machines running Linux or other 32/64-bits virtual machines on the top.

So in fact virtualization is **hardware virtualization.**

Containers effectively virtualize the host operating system (or kernel) and isolate an application’s dependencies from other containers running on the same machine.

Before containers, if you had multiple applications deployed on the same virtual machine (VM), any changes to shared dependencies could cause strange things to happen—so the tendency was to have one application per virtual machine.

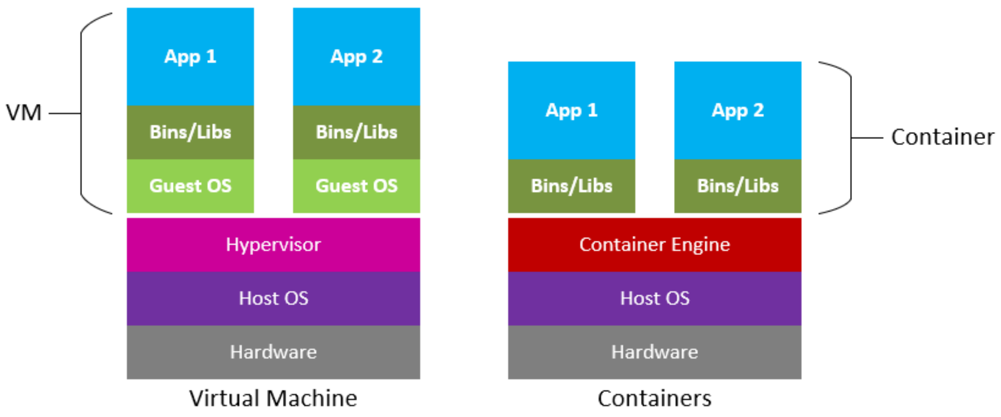
The solution of one application per VM solved the isolation problem for conflicting dependencies, but it wasted a lot of resources (CPU and memory).

This is because a VM runs not only your application but also a full operating system that needs resources too, so less would be available for your application to use.

Containers solve this problem with two pieces: a container engine and a container image, which is a package of an application and its dependencies.

The container engine runs applications in containers isolating it from other applications running on the host machine.

This removes the need to run a separate operating system for each application, allowing for higher resource utilization and lower costs.



CONTAINER ECOSYSTEM

A container ecosystem refers to the collection of tools, technologies, and platforms that enable the creation, deployment, and management of containerized applications.

Containers are lightweight, isolated, and portable units of software that package an application and its dependencies, allowing it to run consistently across different computing environments.

COMPONENTS OF A CONTAINER ECOSYSTEM

A container ecosystem typically consists of the **following** components:

CONTAINERIZATION TECHNOLOGY

Containerization technology, such as Docker, is at the core of a container ecosystem.

It provides the necessary tools and runtime environment to create and manage containers.

Containerization **technology** allows developers to package applications and their dependencies into portable containers, ensuring consistency and **reproducibility** across different environments.

CONTAINER ORCHESTRATION PLATFORMS

Container orchestration platforms, like Kubernetes, are used to manage and automate the deployment, scaling, and monitoring of containers.

These platforms provide a higher level of abstraction, allowing developers to define the desired state of their containerized applications and leaving the platform to handle the underlying infrastructure and resource management.

CONTAINER REGISTRIES

Container registries, such as Docker Hub or Amazon Elastic Container Registry (ECR), serve as **repositories** for storing and distributing container images.

These registries allow developers to share and distribute their containerized applications, making it easier to deploy them on different systems or share them with others.

CONTAINER NETWORKING

Container networking technologies enable communication between containers and external systems.

They provide mechanisms for containerized applications to connect with each other, as well as with other services or resources outside the container ecosystem.

Container networking ensures that containers can **seamlessly** interact with other components of the application architecture.

CONTAINER SECURITY

Container security tools and practices are essential to protect containerized applications from vulnerabilities and threats.

These tools help ensure that containers are built from trusted sources, regularly updated, and isolated from each other to prevent unauthorized access or data breaches.

Container security is a critical aspect of a container ecosystem, especially when deploying containers in production environments.

BENEFITS OF A CONTAINER ECOSYSTEM

A container ecosystem offers several benefits for software **development** and deployment:

- **Portability:** Containers provide a consistent runtime environment, making it easier to deploy applications across different systems and cloud platforms.
- **Scalability:** Container orchestration platforms enable automatic scaling of applications based on demand, ensuring efficient resource utilization.
- **Isolation:** Containers provide isolation between applications, preventing conflicts and dependencies issues.
- **Efficiency:** Containers are lightweight and start quickly, allowing for faster development, testing, and deployment cycles.
- **Reproducibility:** Containers encapsulate the application and its dependencies, ensuring consistent behavior across different environments.

Overall, a container ecosystem simplifies the development, deployment, and management of applications, making it a popular choice for modern software development and deployment workflows.

DOCKER

Docker was first released in 2013 and is responsible for revolutionizing container technology by providing a toolset to easily create container images of applications.

The underlying concept has been around longer than Docker’s technology, but it was not easy to do until Docker came out with its cohesive set of tools to accomplish it.

Docker consists of a few components: a container runtime (called dockerd), a container image builder (BuildKit), and a CLI that is used to work with the builder, containers, and the engine (called docker).

DOCKER IMAGES VS. DOCKER CONTAINERS

A Docker image is a template; a Docker container is a running instance of that template.
To create an image with your application’s source code, you specify a list of commands in a special text file named Dockerfile. The docker builder takes this file and packages it into an image. Once you have an image, you push it to a container registry—a central repository for versioning your images.
When you want to run a Docker image, you need to either build it or pull the image from a registry.
DockerHub is a well-known public registry, but there are also private registries like Azure Container Registry that allow you to keep your application images private.

CONTAINER ORCHESTRATION AND MANAGEMENT OPTIONS

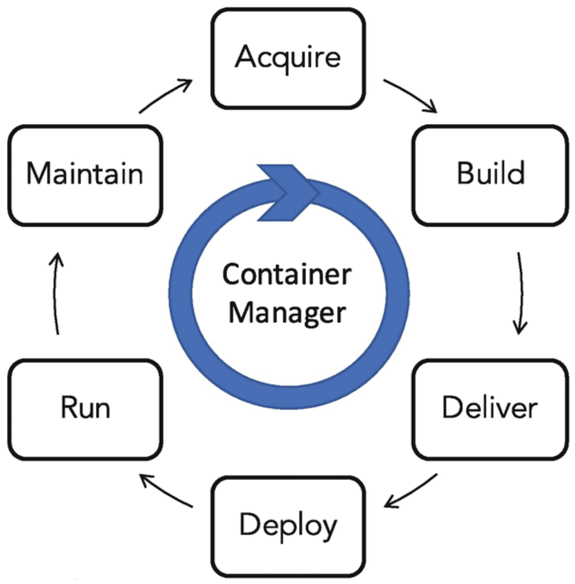
Container Orchestration is used for managing, scheduling, scaling, storage and networking for individual **containers**.

This can be used in any environment where we use the containers.

Container Orchestration helps to deploy the same application across different environments without needing to re-design or re-configure it.

Also, the **microservices** in containers make it easier to orchestrate services, including **docker storage**, **security** and **networking**.

CONTAINER ORCHESTRATION



Container orchestration basically focuses on managing containers and their dynamic environments.

It is all about managing the **lifecycles of containers** and it also helps **DevOps** teams who integrate containers in **CI/CD** workflow.

The Software team uses container orchestration engine for controlling/managing and automating tasks mentioned below:

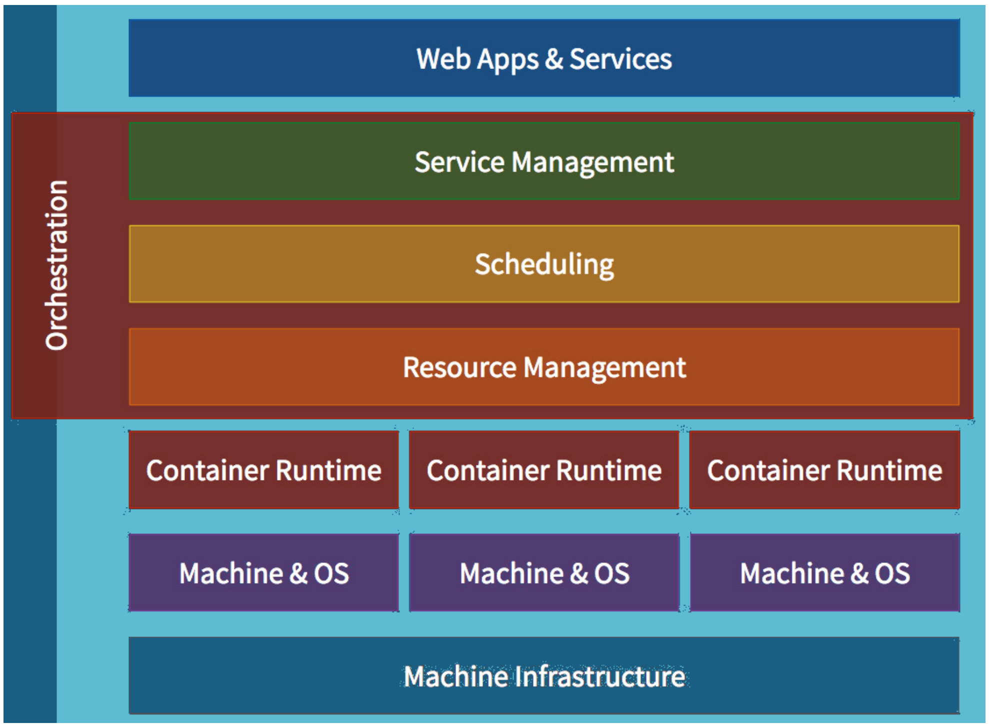
- Containers provisioning and deployment
- Availability of containers
- Scaling up or down according to spread application load evenly across the host
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Allocation of resources between containers
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts

Check Out : Our kubernetes training.

A Kubernetes certification makes your resume look good and stand out from the competition.

As companies rely more and more on Kubernetes, your expertise will be an immediate asset.

WORKING OF CONTAINER ORCHESTRATION



In any container orchestration tool, we have to write a configuration file using either **YAML** or **JSON**.

This file will describe the configuration of an application like where to find the **docker images**, how to establish a network between containers, how to **mount storage** volumes and where to store the logs for that container.

The developer team writes and do version control of these configuration files so they can deploy the same applications across different development and testing environments before deploying them to production clusters.

Containers are deployed onto the hosts, usually in Replicated groups.

The container orchestration tool subsequently schedules the deployment, once it's time to deploy a container into the cluster, then it searches for a suitable host to place the container.

The host is found on the basis of some constraints mentioned in the configuration file such as **CPU** or **memory availability**.

We can even place containers according to **labels** or metadata, or according to their proximity in relation to other hosts.

Once the container is up and running on the host, the orchestration tool has to manage the lifecycle of a container according to the specifications we have mentioned in the container's definition file (for example, it's **Dockerfile**).

BENEFITS OF CONTAINER ORCHESTRATION TOOLS

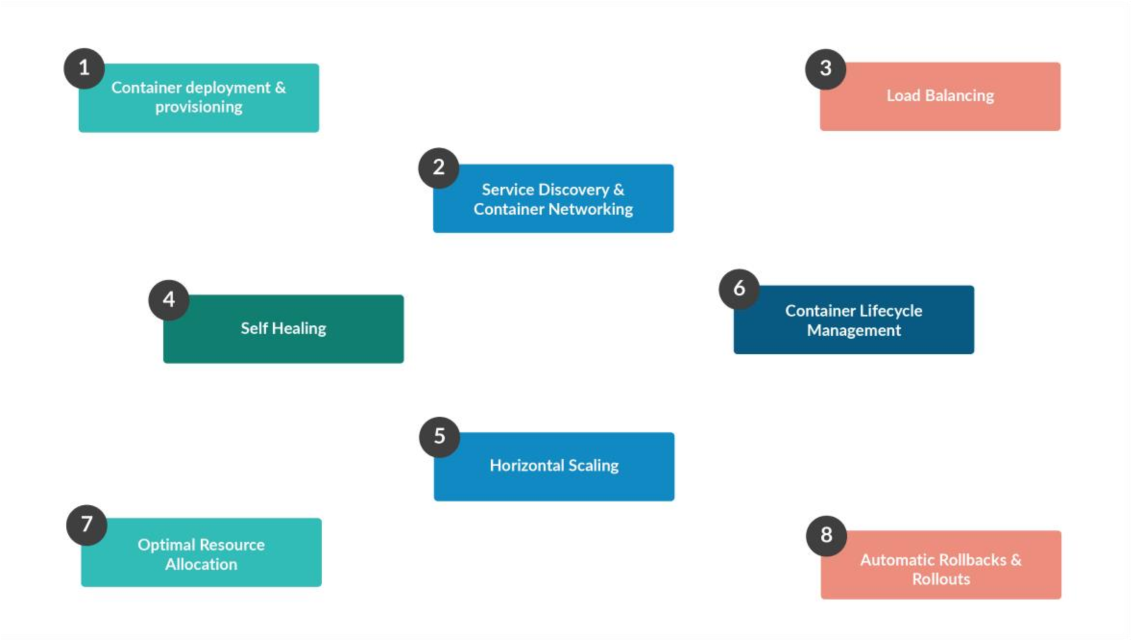
The main benefits of Container Orchestration Tools are that they manage the whole containerization process.

They also provide **portability** and **reproducibility** option for a containerized process which means that we have an opportunity to **move** and **scale** our containerized applications across clouds and data centres according to our need.

We only have to mention in the Configuration file.

Some more benefits of using Container Orchestration engine are:

- Service discovery and container networking
- Improved governance and security controls
- Container health monitoring
- Load balancing of containers evenly among hosts
- Optimal resource allocation
- Container lifecycle management


1 Container deployment & provisioning
3 Load Balancing
2 Service Discovery & Container Networking
4 Self Healing
6 Container Lifecycle Management
5 Horizontal Scaling
7 Optimal Resource Allocation
8 Automatic Rollbacks & Rollouts

KUBERNETES

Kubernetes is an open-source container management platform that unifies a cluster of machines into a single pool of compute resources. With Kubernetes, you organize your applications in groups of containers, which it runs using the Docker engine, taking care of keeping your application running as you request. Kubernetes provides the following:

- **Compute scheduling**—It considers the resource needs of your containers, to find the right place to run them automatically
- **Self-healing**—If a container crashes, a new one will be created to replace it.
- **Horizontal scaling**—By observing CPU or custom metrics, Kubernetes can add and remove instances as needed.
- **Volume management**—It manages the persistent storage used by your applications
- **Service discovery & load balancing**—IP address, DNS, and multiple instances are load-balanced.
- **Automated rollouts & rollbacks**–During updates, the health of your new instances are monitored, and if a failure occurs, it can roll back to the previous version automatically.
- **Secret & configuration management**. It manages application configuration and secrets.

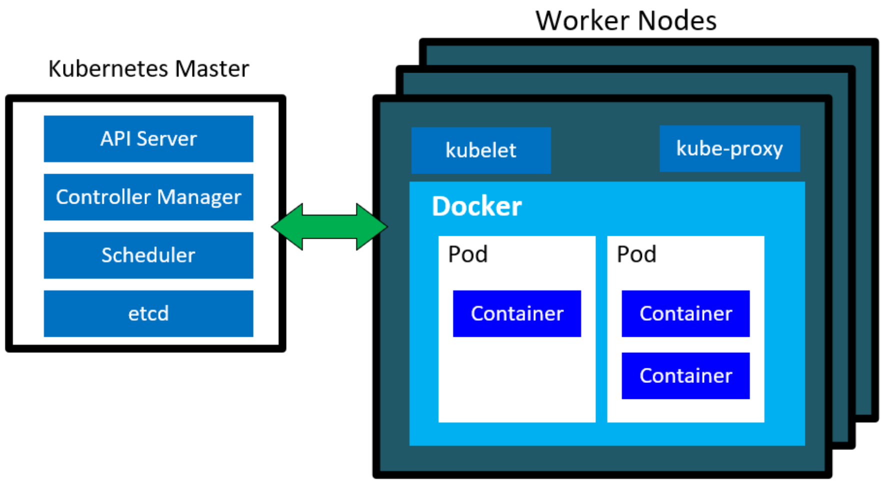
Kubernetes uses a master/slave communication model where there is at least one master and usually several worker nodes. The master (sometimes called the control plane) has three components and a data store:

- **API server**—exposes the Kubernetes API for controlling the cluster
- **Controller manager**—responsible for watching the cluster's objects and resources and ensuring the desired state is consistent
- **Scheduler**—responsible for scheduling compute requests on the cluster

- **etcd**—an open-source distributed key value store used to hold the cluster data

The worker nodes provide the container runtime for your applications and have a few components responsible for communicating with the master and networking on every worker node:

- **Kubelet**—responsible for communicating to the master and ensuring the containers are running on the node
- **Kube-proxy**—enables the cluster to forward traffic to executing containers
- **Docker (container runtime)**—provides the runtime environment for containers



The master and workers are the platform that run your applications. In order to get your applications running on the cluster, you need to interact with the API server and work with the Kubernetes object model.

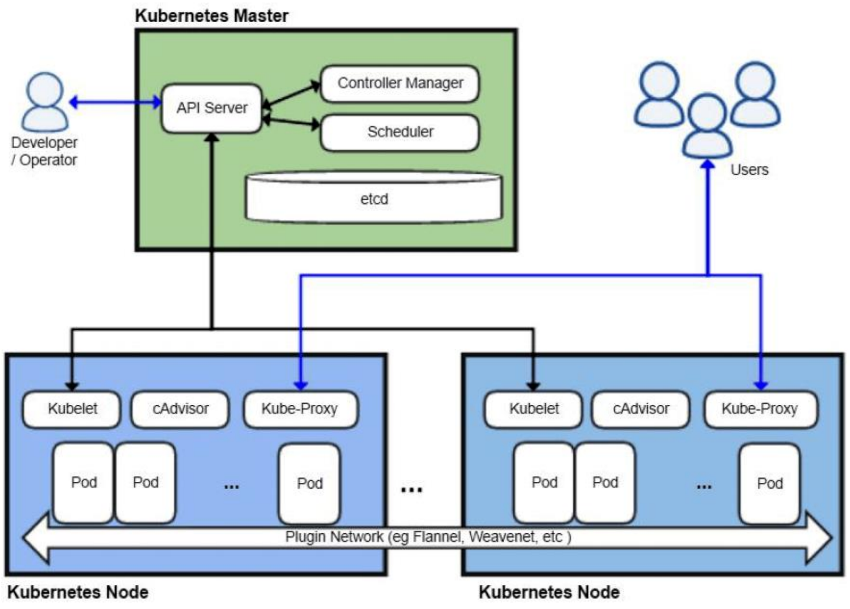
KUBERNETES OBJECTS

To run an application on Kubernetes, you need to communicate with the API server using the object model. The objects are usually expressed in .yaml or .json format; kubectl is the command-line interface used to interact with the API. The most common objects are:

- **Pod**—a group of one or more containers and metadata
- **Service**—works with a set of pods and directs traffic to them
- **Deployment**—ensures the desired state and scale are maintained

KUBERNETES ARCHITECTURE

Kubernetes provides a flexible architecture for discovering services within a cluster while maintaining loose connections. A Kubernetes cluster comprises a set of control planes and compute nodes. The primary role of the control plane is to oversee the entire cluster, expose the API, and manage the scheduling of compute nodes according to the desired settings. The compute nodes run container runtimes such as Docker, alongside a communication agent called kubelet, which interacts with the control plane. These nodes can consist of physical servers, virtual machines (VMs) in on-premises or cloud environments.



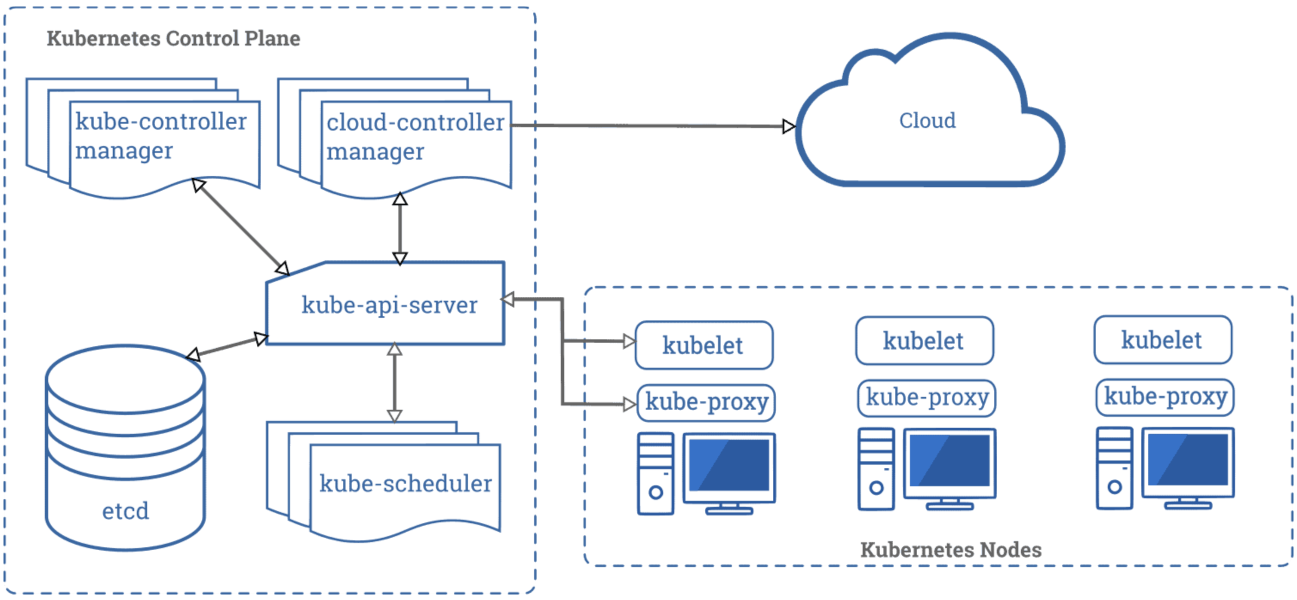
1) In the Kubernetes architecture diagram above you can see, there is one master and multiple nodes.

2) The Master node communicates with Worker nodes using Kube API-server to kubelet communication.

3) In the Worker node, there can be one or more pods and pods can contain one or more containers.

4) Containers can be deployed using the image also can be deployed externally by the user.

KUBERNETES ARCHITECTURE COMPONENTS



KUBERNETES MASTER NODE

In Kubernetes (k8s), a master node is the **control plane component** responsible for **managing the cluster**. It coordinates and schedules tasks, maintains cluster state, and monitors node health. It includes components like API server, scheduler, and controller manager, ensuring overall cluster functionality and orchestration of containerized applications. Master Node Components:

1) Kube API server handles administrative tasks on the master node. Users send REST commands in YAML/JSON to the API server, which processes and executes them. The Kube API server acts as the front end of the Kubernetes control plane.

2) etcd, a distributed key-value store, maintains the cluster state and configuration details like subnets and config maps in Kubernetes' database. It's where Kubernetes stores its information.

3) Kube-scheduler assigns tasks to worker nodes and manages new requests from the API Server, ensuring they are directed to healthy nodes.

4) Kube Controller Manager task is to retrieve the desired state from the API Server. If the desired state does not match the current state of the object, corrective steps are taken by the control loop to align the current state with the desired state.

There are different types of control manager in Kubernetes architecture:

- **Node Manager:** It oversees nodes, creating new ones in case of unavailability or destruction.
- **Replication Controller:** It ensures the desired container count is maintained within the replication group.
- **Endpoints Controller:** This controller populates the endpoints object, connecting Services & Pods.

KUBERNETES WORKER NODE

Worker nodes in a cluster are machines or servers running applications, controlled by the Kubernetes master. Multiple nodes connect to the master. On each node, multiple pods and containers operate.

COMPONENTS OF WORKER NODES:

1) Kubelet, an agent on each node, communicates with the master. It ensures pod containers' health, executing tasks like deploying or destroying containers, reporting back to the Master.

2) Kube-proxy enables worker node communication, managing network rules. It ensures rules are set for containers to communicate across nodes.

3) A **Kubernetes pod** is a set of containers on a single host, sharing storage and network. It includes specifications for container execution, enabling easy inter-container communication.

4) Container Runtime, responsible for container execution, supports multiple runtimes: Docker, containers.

KUBERNETES - SETUP

It is important to set up the Virtual Datacenter (vDC) before setting up Kubernetes. This can be considered as a set of machines where they can communicate with each other via the network. For hands-on approach, you can set up vDC on **PROFITBRICKS** if you do not have a physical or cloud infrastructure set up.

Once the IaaS setup on any cloud is complete, you need to configure the **Master** and the **Node**.

Note − The setup is shown for Ubuntu machines. The same can be set up on other Linux machines as well.

Prerequisites

Installing Docker − Docker is required on all the instances of Kubernetes. Following are the steps to install the Docker.

Step 1 − Log on to the machine with the root user account.

Step 2 − Update the package information. Make sure that the apt package is working.

Step 3 − Run the following commands.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

Step 4 − Add the new GPG key.

```
$ sudo apt-key adv \
   --keyserver hkp://ha.pool.sks-keyservers.net:80 \
   --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty
main" | sudo tee
/etc/apt/sources.list.d/docker.list
```

Step 5 − Update the API package image.

```
$ sudo apt-get update
```

Once all the above tasks are complete, you can start with the actual installation of the Docker engine. However, before this you need to verify that the kernel version you are using is correct.

Install Docker Engine

Run the following commands to install the Docker engine.

Step 1 − Logon to the machine.

Step 2 − Update the package index.

```
$ sudo apt-get update
```

Step 3 − Install the Docker Engine using the following command.

```
$ sudo apt-get install docker-engine
```

Step 4 − Start the Docker daemon.

```
$ sudo apt-get install docker-engine
```

Step 5 − To very if the Docker is installed, use the following command.

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

Install etcd 2.0

This needs to be installed on Kubernetes Master Machine. In order to install it, run the following commands.

```
$ curl -L
https://github.com/coreos/etcd/releases/download/v2.0.0/etcd
-v2.0.0-linux-amd64.tar.gz -o etcd-v2.0.0-linux-amd64.tar.gz ->1
$ tar xzvf etcd-v2.0.0-linux-amd64.tar.gz ------>2
$ cd etcd-v2.0.0-linux-amd64 ------------>3
$ mkdir /opt/bin ------------->4
$ cp etcd* /opt/bin ----------->5
```

In the above set of command −

- First, we download the **etcd**. Save this with specified name.
- Then, we have to un-tar the tar package.
- We make a dir. inside the /opt named bin.

- Copy the extracted file to the target location.

Now we are ready to build Kubernetes. We need to install Kubernetes on all the machines on the cluster.

$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git

$ cd kubernetes

$ make release

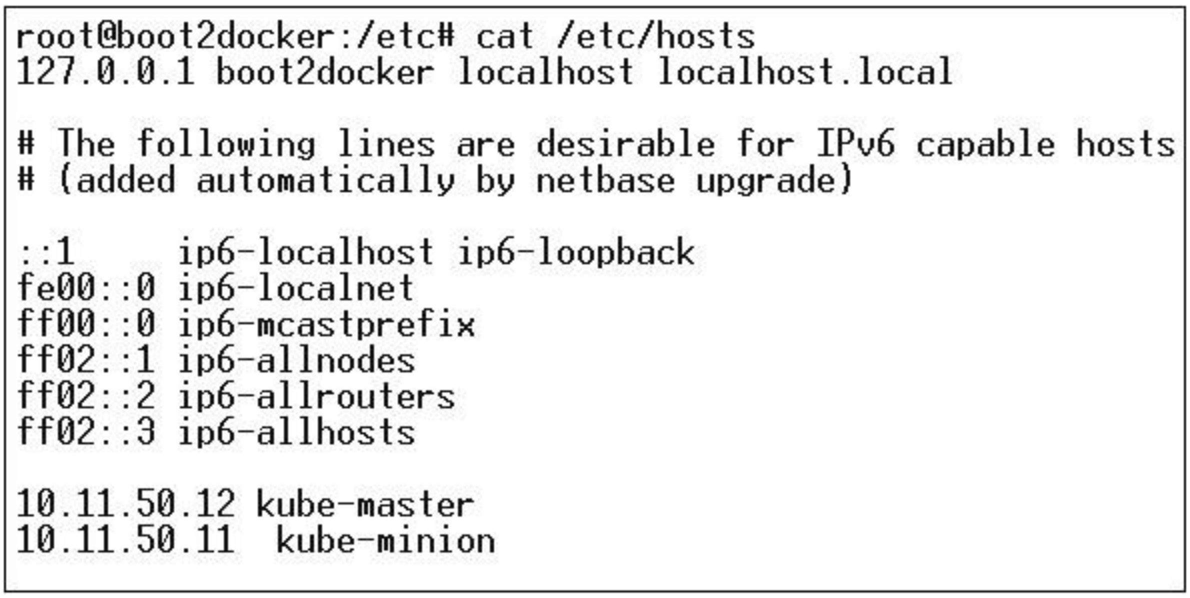
The above command will create a **_output** dir in the root of the kubernetes folder. Next, we can extract the directory into any of the directory of our choice /opt/bin, etc.

Next, comes the networking part wherein we need to actually start with the setup of Kubernetes master and node. In order to do this, we will make an entry in the host file which can be done on the node machine.

$ echo "<IP address of master machine> kube-master

< IP address of Node Machine>" >> /etc/hosts

Following will be the output of the above command.



```
root@boot2docker:/etc# cat /etc/hosts
127.0.0.1 boot2docker localhost localhost.local

# The following lines are desirable for IPv6 capable hosts
# (added automatically by netbase upgrade)

::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts

10.11.50.12 kube-master
10.11.50.11  kube-minion
```

Now, we will start with the actual configuration on Kubernetes Master.

First, we will start copying all the configuration files to their correct location.

```
$ cp <Current dir. location>/kube-apiserver /opt/bin/
$ cp <Current dir. location>/kube-controller-manager /opt/bin/
$ cp <Current dir. location>/kube-kube-scheduler /opt/bin/
$ cp <Current dir. location>/kubecfg /opt/bin/
$ cp <Current dir. location>/kubectl /opt/bin/
$ cp <Current dir. location>/kubernetes /opt/bin/
```

The above command will copy all the configuration files to the required location. Now we will come back to the same directory where we have built the Kubernetes folder.

```
$ cp kubernetes/cluster/ubuntu/init_conf/kube-apiserver.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-controller-manager.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-kube-scheduler.conf /etc/init/


$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-apiserver /etc/init.d/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-controller-manager /etc/init.d/
```

$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-kube-scheduler /etc/init.d/

$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet /etc/default/

$ cp kubernetes/cluster/ubuntu/default_scripts/kube-proxy /etc/default/

$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet /etc/default/

The next step is to update the copied configuration file under /etc. dir.

Configure etcd on master using the following command.

$ ETCD_OPTS = "-listen-client-urls = http://kube-master:4001"

Configure kube-apiserver

For this on the master, we need to edit the **/etc/default/kube-apiserver** file which we copied earlier.

$ KUBE_APISERVER_OPTS = "--address = 0.0.0.0 \

--port = 8080 \

--etcd_servers = <The path that is configured in ETCD_OPTS> \

--portal_net = 11.1.1.0/24 \

--allow_privileged = false \

--kubelet_port = < Port you want to configure> \

--v = 0"

Configure the kube Controller Manager

We need to add the following content in **/etc/default/kube-controller-manager**.

```
$ KUBE_CONTROLLER_MANAGER_OPTS = "--address = 0.0.0.0 \
--master = 127.0.0.1:8080 \
--machines = kube-minion \ -----> #this is the kubernatics node
--v = 0
```

Next, configure the kube scheduler in the corresponding file.

```
$ KUBE_SCHEDULER_OPTS = "--address = 0.0.0.0 \
--master = 127.0.0.1:8080 \
--v = 0"
```

Once all the above tasks are complete, we are good to go ahead by bring up the Kubernetes Master. In order to do this, we will restart the Docker.

```
$ service docker restart
```

Kubernetes Node Configuration

Kubernetes node will run two services the **kubelet and the kube-proxy**.

Before moving ahead, we need to copy the binaries we downloaded to their required folders where we want to configure the kubernetes node.

Use the same method of copying the files that we did for kubernetes master. As it will only run the kubelet and the kube-proxy, we will configure them.

```
$ cp <Path of the extracted file>/kubelet /opt/bin/
$ cp <Path of the extracted file>/kube-proxy /opt/bin/
$ cp <Path of the extracted file>/kubecfg /opt/bin/
$ cp <Path of the extracted file>/kubectl /opt/bin/
$ cp <Path of the extracted file>/kubernetes /opt/bin/
```

Now, we will copy the content to the appropriate dir.

```
$ cp kubernetes/cluster/ubuntu/init_conf/kubelet.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/init_conf/kube-proxy.conf /etc/init/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kubelet /etc/init.d/
$ cp kubernetes/cluster/ubuntu/initd_scripts/kube-proxy /etc/init.d/
$ cp kubernetes/cluster/ubuntu/default_scripts/kubelet /etc/default/
$ cp kubernetes/cluster/ubuntu/default_scripts/kube-proxy /etc/default/
```

We will configure the **kubelet** and **kube-proxy conf** files.

We will configure the **/etc/init/kubelet.conf**.

```
$ KUBELET_OPTS = "--address = 0.0.0.0 \
--port = 10250 \
--hostname_override = kube-minion \
--etcd_servers = http://kube-master:4001 \
--enable_server = true
--v = 0"
/
```

For kube-proxy, we will configure using the following command.

```
$ KUBE_PROXY_OPTS = "--etcd_servers = http://kube-master:4001 \
--v = 0"
/etc/init/kube-proxy.conf
```

Finally, we will restart the Docker service.

```
$ service docker restart
```

Now we are done with the configuration. You can check by running the following commands.

```
$ /opt/bin/kubectl get minions
```

KUBERNETES - IMAGES

Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, we are only supporting Kubernetes to support Docker images. Each container in a pod has its Docker image running inside it.

When we are configuring a pod, the image property in the configuration file has the same syntax as the Docker command does. The configuration file has a field to define the image name, which we are planning to pull from the registry.

Following is the common configuration structure which will pull image from Docker registry and deploy in to Kubernetes container.

```
apiVersion: v1
kind: pod
metadata:
  name: Tesing_for_Image_pull -----------> 1
  spec:
    containers:
      - name: neo4j-server ------------------------> 2
      image: <Name of the Docker image>----------> 3
      imagePullPolicy: Always ------------->4
      command: ["echo", "SUCCESS"] ------------------->
```

In the above code, we have defined −

- **name: Tesing_for_Image_pull** − This name is given to identify and check what is the name of the container that would get created after pulling the images from Docker registry.
- **name: neo4j-server** − This is the name given to the container that we are trying to create. Like we have given neo4j-server.
- **image: <Name of the Docker image>** − This is the name of the image which we are trying to pull from the Docker or internal registry of images. We need to define a complete registry path along with the image name that we are trying to pull.
- **imagePullPolicy** − Always - This image pull policy defines that whenever we run this file to create the container, it will pull the same name again.
- **command: [“echo”, “SUCCESS”]** − With this, when we create the container and if everything goes fine, it will display a message when we will access the container.

In order to pull the image and create a container, we will run the following command.

```
$ kubectl create –f Tesing_for_Image_pull
```

Once we fetch the log, we will get the output as successful.

```
$ kubectl log Tesing_for_Image_pull
```

The above command will produce an output of success or we will get an output as failure.

Note − It is recommended that you try all the commands yourself.

KUBERNETES - POD

A pod is a collection of containers and its storage inside a node of a Kubernetes cluster.

It is possible to create a pod with multiple containers inside it.

For example, keeping a database container and data container in the same pod.

Types of Pod

There are two types of Pods −

- Single container pod
- Multi container pod

Single Container Pod

They can be simply created with the kubctl run command, where you have a defined image on the Docker registry which we will pull while creating a pod.

```
$ kubectl run <name of pod> --image=<name of the image from registry>
```

Example − We will create a pod with a tomcat image which is available on the Docker hub.

```
$ kubectl run tomcat --image = tomcat:8.0
```

This can also be done by creating the **yaml** file and then running the **kubectl create** command.

```
apiVersion: v1
kind: Pod
metadata:
   name: Tomcat
spec:
   containers:
   - name: Tomcat
    image: tomcat: 8.0
    ports:
containerPort: 7500
   imagePullPolicy: Always
```

Once the above **yaml** file is created, we will save the file with the name of **tomcat.yml** and run the create command to run the document.

```
$ kubectl create –f tomcat.yml
```

It will create a pod with the name of tomcat. We can use the describe command along with **kubectl** to describe the pod.

Multi Container Pod

Multi container pods are created using **yaml mail** with the definition of the containers.

```
apiVersion: v1
kind: Pod
```

```
metadata:
   name: Tomcat
spec:
   containers:
   - name: Tomcat
    image: tomcat: 8.0
    ports:
containerPort: 7500
   imagePullPolicy: Always
   -name: Database
   Image: mongoDB
   Ports:
containerPort: 7501
   imagePullPolicy: Always
```

In the above code, we have created one pod with two containers inside it, one for tomcat and the other for MongoDB.

ENCAPSULATION OF ONE OR MORE CONTAINERS WITHIN A POD

A pod encapsulates one or more containers and provides a shared execution environment for them. Containers within a pod are co-located and share the same network and storage namespaces. They can communicate with each other using localhost, making it simple for containers within a pod to interact and coordinate their activities.

The encapsulation of containers within a pod allows them to share resources, such as CPU and memory, and simplifies the management and deployment of related containers. Containers within a pod can also mount shared volumes, enabling them to access and share persistent data.

Lifecycle Management and Scaling of Pods

Pods have their own lifecycle within the Kubernetes cluster. The Kubernetes control plane is responsible for managing the creation, termination, and updates of pods based on the desired state defined in the deployment configurations.

Pods can be created, deleted, or updated using declarative configuration files. Kubernetes ensures that the desired number of replicas of a pod is maintained based on the specified configurations. If scaling is required, Kubernetes can horizontally scale the pods by replicating them across multiple nodes.

Communication and Networking within a Pod

Containers within a pod share the same network namespace, allowing them to communicate with each other using localhost.

They can use standard inter-process communication mechanisms, such as TCP/IP or Unix sockets, to exchange data.

Each pod is assigned a unique IP address within the cluster, known as the pod IP address.

Containers within the pod can communicate with each other using this shared IP address.

Additionally, containers within a pod share the same port space, meaning they can communicate over common ports without conflict.

This communication and networking model within a pod enables containers to collaborate and work together as a cohesive unit, making it easier to build and manage complex, multi-container applications within the Kubernetes ecosystem.

IMMUTABLE KUBERNETES PODS

The old saying “the only constant in life is change” is especially true when talking about technology. In software development, concepts such as agile scrum and continuous delivery attempt to embrace change and streamline its delivery. A more drastic approach to making constant operational change more efficient is the use of an immutable architecture.

Knowing that iterative changes and upgrades are essential to the success of applications, this might seem odd. After all, change also affects the underlying infrastructure and its configurations that support your applications. So how would a business benefit from adopting anything described as immutable?

immutable architecture, the advantages it offers, and how it supports the notion of Infrastructure as Code (IaC).

Immutable Architecture

Immutable architecture, also known as immutable infrastructure, is a term that can be a bit misleading. Coined by Chad Fowler, an immutable architecture doesn’t mean that your environment should never change, but rather, once a specific instance (such as a container or virtual machine) is started, its configuration should never change.

Instead of upgrading or re-configuring the underlying infrastructure of that instance (of a container or virtual machine), you should simply replace it entirely with a new instance that has all of your required changes. You may need to replace the instance within minutes, days, or weeks due to a workload change, an architecture change, or simply to keep up with changes going on elsewhere in your environment. Either way, replacing the instance allows for discrete versioning in your application environment. You can think of each version as a configuration snapshot at an exact point in time. Systematic versioning, in turn, lowers the risk of making mistakes during upgrades, offers the ability to test before rolling out, and enables rolling back (to the previous version) if your application encounters a problem. Combining flexibility with precision is the primary goal of an immutable architecture.

The practice of immutable architecture doesn't have to be limited to just the underlying infrastructure of your workloads; you can use this technique to support middleware components (such as a messaging bus, a database, or a data cache) and application software as well. You would simply release application source code as new, immutable, and versioned artifacts. You may version each package in the form of a new Docker image, a new Virtual Machine Image, or a new .jar file (Java code).

Mutable vs. Immutable Architecture

Let's use a real-world example to illustrate the differences between a mutable and an immutable architecture. Say your company has an application web server running on a VM in the cloud. This web server has Nginx (webserver) installed on it and a specific web application version. After some time passes, you decide it’s time to upgrade the version of Nginx or switch to Apache.

Creating Immutable Pods

The time below summarizes the benefits of adopting an immutable architecture **Creating Immutable Pods**

Security is never straight-forward. There is not a single switch we can flip to make our containers immutable. We need to cater for all possible sources of mutability. Fortunately, containers and pods make our job a bit easier, especially if we compare it to in-place deployments that were so common before containerization hit our industry.

In this tutorial, we will focus on the use of the securityContext construct in Kubernetes. We are going to do the following:

- Set the filesystem as readonly
- Ensure the container’s processes run as non-root
- Ensure the container does not have elevated privileges on the host
- Ensure the container cannot request an escalation of privileges

I am going to use the nginx image because we want to learn how to secure a Pod, even though it already exists another image, nginxinc/nginx-unprivileged, that is already set to run in unprivileged mode.

Before we create the actual deployment, I will create a configuration map named nginx-conf with a basic nginx configuration to make sure it runs on port 8080 rather than the default 80 that cannot be opened by a non-root user.

Create a file named default.conf

```
server {
   listen 8080;
   server_name localhost;
   #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
    # proxy the PHP scripts to Apache listening on 127.0.0.1:80
    #
    #location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
    #}

    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    #location ~ \.php$ {
    #    root           html;
    #    fastcgi_pass   127.0.0.1:9000;
    #    fastcgi_index  index.php;
    #    fastcgi_param  SCRIPT_FILENAME  /scripts$fastcgi_script_name;
    #    include        fastcgi_params;
    #}

    # deny access to .htaccess files, if Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #    deny  all;
    #}
```

```
}
```

Then run the following command:

```
k create configmap nginx-conf --from-file default.confCopy
```

Finally we define our deployment manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: default
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
```

```
metadata:
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources: {}
    securityContext: # enforces the security context
      allowPrivilegeEscalation: false
      privileged: false
      readOnlyRootFilesystem: true
      runAsGroup: 101
      runAsUser: 101
    startupProbe: #removes the bash shell
      exec:
        command:
        - rm
        - /bin/bash
      failureThreshold: 3
      initialDelaySeconds: 5
      periodSeconds: 5
      successThreshold: 1
      timeoutSeconds: 1
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts: # necessary volumes to have nginx startup
    - mountPath: /var/cache/nginx
      name: cache
    - mountPath: /var/run
      name: run
    - mountPath: /etc/nginx/conf.d
      name: config
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  terminationGracePeriodSeconds: 30
  volumes:
  - emptyDir: {}
    name: cache
  - emptyDir: {}
    name: run
  - configMap:
      defaultMode: 420
      name: nginx-conf
    name: config
```

CONFIGMAPS

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Caution: ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

Motivation

Use a ConfigMap for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named DATABASE_HOST. Locally, you set that variable to localhost. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

Note: A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

ConfigMap object

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a spec, a ConfigMap has data and binaryData fields. These fields accept key-value pairs as their values. Both the data field and the binaryData are optional. The data field is designed to contain UTF-8 strings while the binaryData field is designed to contain binary data as base64-encoded strings.

The name of a ConfigMap must be a valid DNS subdomain name.

Each key under the data or the binaryData field must consist of alphanumeric characters, -, _ or .. The keys stored in data must not overlap with the keys in the binaryData field.

Starting from v1.19, you can add an immutable field to a ConfigMap definition to create an immutable ConfigMap.

ConfigMaps and Pods

You can write a Pod spec that refers to a ConfigMap and configures the container(s) in that Pod based on the data in the ConfigMap. The Pod and the ConfigMap must be in the same namespace.

Note: The spec of a static Pod cannot refer to a ConfigMap or any other API objects.

Here's an example ConfigMap that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the kubelet uses the data from the ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.

Here's an example Pod that uses values from game-demo to configure a Pod:

configmap/configure-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
    - name: PLAYER_INITIAL_LIVES # Notice that the case
is different here
                   # from the key name in the ConfigMap.
     valueFrom:
      configMapKeyRef:
       name: game-demo       # The ConfigMap this value
comes from.
       key: player_initial_lives # The key to fetch.
    - name: UI_PROPERTIES_FILE_NAME
     valueFrom:
      configMapKeyRef:
       name: game-demo
       key: ui_properties_file_name
   volumeMounts:
   - name: config
    mountPath: "/config"
    readOnly: true
 volumes:
 # You set volumes at the Pod level, then mount them into
containers inside that Pod
 - name: config
  configMap:
   # Provide the name of the ConfigMap you want to mount.
   name: game-demo
```

```
# An array of keys from the ConfigMap to create as files
items:
- key: "game.properties"
  path: "game.properties"
- key: "user-interface.properties"
  path: "user-interface.properties"
```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters is how Pods and other objects consume those values.

For this example, defining a volume and mounting it inside the demo container as /config creates two files, /config/game.properties and /config/user-interface.properties, even though there are four keys in the ConfigMap. This is because the Pod definition specifies an items array in the volumes section. If you omit the items array entirely, every key in the ConfigMap becomes a file with the same name as the key, and you get 4 files.

Using ConfigMaps

ConfigMaps can be mounted as data volumes.

ConfigMaps can also be used by other parts of the system, without being directly exposed to the Pod.

For example, ConfigMaps can hold data that other parts of the system should use for configuration.

The most common way to use ConfigMaps is to configure settings for containers running in a Pod in the same namespace.

You can also use a ConfigMap separately.

For example, you might encounter addons or operators that adjust their behavior based on a ConfigMap.

Using ConfigMaps as files from a Pod

To consume a ConfigMap in a volume in a Pod:

1. Create a ConfigMap or use an existing one. Multiple Pods can reference the same ConfigMap.
2. Modify your Pod definition to add a volume under .spec.volumes[]. Name the volume anything, and have a .spec.volumes[].configMap.name field set to reference your ConfigMap object.
3. Add a .spec.containers[].volumeMounts[] to each container that needs the ConfigMap. Specify .spec.containers[].volumeMounts[].readOnly = true and .spec.containers[].volumeMounts[].mountPath to an unused directory name where you would like the ConfigMap to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the ConfigMap data map becomes the filename under mountPath.

This is an example of a Pod that mounts a ConfigMap in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    configMap:
      name: myconfigmap
```

Each ConfigMap you want to use needs to be referred to in .spec.volumes.

If there are multiple containers in the Pod, then each container needs its own volumeMounts block, but only one .spec.volumes is needed per ConfigMap.

Mounted ConfigMaps are updated automatically

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap. The type of the cache is configurable using the configMapAndSecretChangeDetectionStrategy field in the KubeletConfiguration struct. A ConfigMap can be either propagated by watch (default), ttl-based, or by redirecting all requests directly to the API server. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

ConfigMaps consumed as environment variables are not updated automatically and require a pod restart.

Note: A container using a ConfigMap as a subPath volume mount will not receive ConfigMap updates.

Immutable ConfigMaps

FEATURE STATE: Kubernetes v1.21 [stable]

The Kubernetes feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for ConfigMaps marked as immutable.

You can create an immutable ConfigMap by setting the immutable field to true. For example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  ...
data:
  ...
immutable: true
```

Once a ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the data or the binaryData field. You can only delete and recreate the ConfigMap. Because existing Pods maintain a mount point to the deleted ConfigMap, it is recommended to recreate these pods.

Secrets

- A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.
- Such information might otherwise be put in a Pod specification or in a container image.
- Using a Secret means that you don't need to include confidential data in your application code.
- Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods.

- Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing sensitive data to nonvolatile storage.
- Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.

Caution:

- Kubernetes Secrets are, by default, stored unencrypted in the API server's underlying data store (etcd).
- Anyone with API access can retrieve or modify a Secret, and so can anyone with access to etcd.
- Additionally, anyone who is authorized to create a Pod in a namespace can use that access to read any Secret in that namespace; this includes indirect access such as the ability to create a Deployment.
- In order to safely use Secrets, take at least the following steps:
  1. Enable Encryption at Rest for Secrets.
  2. Enable or configure RBAC rules with least-privilege access to Secrets.
  3. Restrict Secret access to specific containers.
  4. Consider using external Secret store providers.
- For more guidelines to manage and improve the security of your Secrets, refer to Good practices for Kubernetes Secrets.

Uses for Secrets

You can use Secrets for purposes such as the following:

- Set environment variables for a container.

- Provide credentials such as SSH keys or passwords to Pods.
- Allow the kubelet to pull container images from private registries.

The Kubernetes control plane also uses Secrets; for example, bootstrap token Secrets are a mechanism to help automate node registration.

Use case: dotfiles in a secret volume

You can make your data "hidden" by defining a key that begins with a dot. This key represents a dotfile or "hidden" file. For example, when the following Secret is mounted into a volume, secret-volume, the volume will contain a single file, called .secret-file, and the dotfile-test-container will have this file present at the path /etc/secret-volume/.secret-file.

Note: Files beginning with dot characters are hidden from the output of ls -l; you must use ls -la to see them when listing directory contents.

secret/dotfile-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
```

```
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: dotfile-secret
  containers:
    - name: dotfile-test-container
      image: registry.k8s.io/busybox
      command:
        - ls
        - "-l"
        - "/etc/secret-volume"
      volumeMounts:
        - name: secret-volume
          readOnly: true
          mountPath: "/etc/secret-volume"
```

Use case: Secret visible to one container in a Pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (for example, over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Alternatives to Secrets

Rather than using a Secret to protect confidential data, you can pick from alternatives.

Here are some of your options:

- If your cloud-native component needs to authenticate to another application that you know is running within the same Kubernetes cluster, you can use a ServiceAccount and its tokens to identify your client.
- There are third-party tools that you can run, either within or outside your cluster, that manage sensitive data. For example, a service that Pods access over HTTPS, that reveals a Secret if the client correctly authenticates (for example, with a ServiceAccount token).
- For authentication, you can implement a custom signer for X.509 certificates, and use CertificateSigningRequests to let that custom signer issue certificates to Pods that need them.

- You can use a device plugin to expose node-local encryption hardware to a specific Pod. For example, you can schedule trusted Pods onto nodes that provide a Trusted Platform Module, configured out-of-band.

You can also combine two or more of those options, including the option to use Secret objects themselves.

For example: implement (or deploy) an operator that fetches short-lived session tokens from an external service, and then creates Secrets based on those short-lived session tokens. Pods running in your cluster can make use of the session tokens, and operator ensures they are valid. This separation means that you can run Pods that are unaware of the exact mechanisms for issuing and refreshing those session tokens.

Types of Secret

When creating a Secret, you can specify its type using the type field of the Secret resource, or certain equivalent kubectl command line flags (if available).

The Secret type is used to facilitate programmatic handling of the Secret data.

Kubernetes provides several built-in types for some common usage scenarios.

These types vary in terms of the validations performed and the constraints Kubernetes imposes on them.

Built-in Type	**Usage**
Opaque	arbitrary user-defined data

Built-in Type	Usage
kubernetes.io/service-account-token	ServiceAccount token
kubernetes.io/dockercfg	serialized ~/.dockercfg file
kubernetes.io/dockerconfigjson	serialized ~/.docker/config.json file
kubernetes.io/basic-auth	credentials for basic authentication
kubernetes.io/ssh-auth	credentials for SSH authentication
kubernetes.io/tls	data for a TLS client or server
bootstrap.kubernetes.io/token	bootstrap token data

You can define and use your own Secret type by assigning a non-empty string as the type value for a Secret object (an empty string is treated as an Opaque type).

Kubernetes doesn't impose any constraints on the type name.

However, if you are using one of the built-in types, you must meet all the requirements defined for that type.

If you are defining a type of Secret that's for public use, follow the convention and structure the Secret type to have your domain name before the name, separated by a /.

For example: cloud-hosting.example.net/cloud-api-credentials.

Opaque Secrets

Opaque is the default Secret type if you don't explicitly specify a type in a Secret manifest.

When you create a Secret using kubectl, you must use the generic subcommand to indicate an Opaque Secret type.

For example, the following command creates an empty Secret of type Opaque:

```
kubectl create secret generic empty-secret
kubectl get secret empty-secret
```

The output looks like:

```
NAME           TYPE     DATA   AGE
empty-secret   Opaque   0      2m6s
```

The DATA column shows the number of data items stored in the Secret.

In this case, 0 means you have created an empty Secret.

ServiceAccount token Secrets

A kubernetes.io/service-account-token type of Secret is used to store a token credential that identifies a ServiceAccount.

This is a legacy mechanism that provides long-lived ServiceAccount credentials to Pods.

In Kubernetes v1.22 and later, the recommended approach is to obtain a short-lived, automatically rotating ServiceAccount token by using the TokenRequest API instead.

You can get these short-lived tokens using the following methods:

- Call the TokenRequest API either directly or by using an API client like kubectl. For example, you can use the kubectl create token command.
- Request a mounted token in a projected volume in your Pod manifest. Kubernetes creates the token and mounts it in the Pod. The token is automatically invalidated when the Pod that it's mounted in is deleted. For details, see Launch a Pod using service account token projection.

Note: You should only create a ServiceAccount token Secret if you can't use the TokenRequest API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you.

For instructions, see Manually create a long-lived API token for a ServiceAccount.

When using this Secret type, you need to ensure that the kubernetes.io/service-account.name annotation is set to an existing ServiceAccount name.

If you are creating both the ServiceAccount and the Secret objects, you should create the ServiceAccount object first.

After the Secret is created, a Kubernetes controller fills in some other fields such as the kubernetes.io/service-account.uid annotation, and the token key in the data field, which is populated with an authentication token.

The following example configuration declares a ServiceAccount token Secret:

secret/serviceaccount-token-secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  extra: YmFyCg==
```

After creating the Secret, wait for Kubernetes to populate the token key in the data field.

See the ServiceAccount documentation for more information on how ServiceAccounts work.

You can also check the automountServiceAccountToken field and the serviceAccountName field of the Pod for information on referencing ServiceAccount credentials from within Pods.

Docker config Secrets

If you are creating a Secret to store credentials for accessing a container image registry, you must use one of the following `type` values for that Secret:

- `kubernetes.io/dockercfg`: store a serialized `~/.dockercfg` which is the legacy format for configuring Docker command line. The Secret `data` field contains a `.dockercfg` key whose value is the content of a base64 encoded `~/.dockercfg` file.
- `kubernetes.io/dockerconfigjson`: store a serialized JSON that follows the same format rules as the `~/.docker/config.json` file, which is a new format

for ~/.dockercfg. The Secret data field must contain a .dockerconfigjson key for which the value is the content of a base64 encoded ~/.docker/config.json file.

Below is an example for a kubernetes.io/dockercfg type of Secret:

secret/dockercfg-secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-dockercfg
type: kubernetes.io/dockercfg
data:
  .dockercfg: |

eyJhdXRocyI6eyJodHRwczovL2V4YW1wbGUvdjEvIjp7ImF1dGgiOiJvcGVuc2V
zYW1lIn19fQo=
```

Note: If you do not want to perform the base64 encoding, you can choose to use the stringData field instead.

When you create Docker config Secrets using a manifest, the API server checks whether the expected key exists in the data field, and it verifies if the value provided can be parsed as a valid JSON.

The API server doesn't validate if the JSON actually is a Docker config file.

You can also use kubectl to create a Secret for accessing a container registry, such as when you don't have a Docker configuration file:

```
kubectl create secret docker-registry secret-tiger-docker \
  --docker-email=tiger@acme.example \
  --docker-username=tiger \
  --docker-password=pass1234 \
  --docker-server=my-registry.example:5000
```

This command creates a Secret of type `kubernetes.io/dockerconfigjson`.

Retrieve the `.data.dockerconfigjson` field from that new Secret and decode the data:

```
kubectl get secret secret-tiger-docker -o jsonpath='{.data.*}' | base64 -d
```

The output is equivalent to the following JSON document (which is also a valid Docker configuration file):

```
{
  "auths": {
    "my-registry.example:5000": {
      "username": "tiger",
      "password": "pass1234",
      "email": "tiger@acme.example",
      "auth": "dGlnZXI6cGFzczEyMzQ="
    }
  }
}
```

Caution:

The `auth` value there is base64 encoded; it is obscured but not secret. Anyone who can read that Secret can learn the registry access bearer token.

It is suggested to use credential providers to dynamically and securely provide pull secrets on-demand.

Basic authentication Secret

The `kubernetes.io/basic-auth` type is provided for storing credentials needed for basic authentication.

When using this Secret type, the data field of the Secret must contain one of the following two keys:

- username: the user name for authentication
- password: the password or token for authentication

Both values for the above two keys are base64 encoded strings.

You can alternatively provide the clear text content using the stringData field in the Secret manifest.

The following manifest is an example of a basic authentication Secret:

secret/basicauth-secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin # required field for kubernetes.io/basic-auth
  password: t0p-Secret # required field for kubernetes.io/basic-auth
```

Note: The stringData field for a Secret does not work well with server-side apply.

The basic authentication Secret type is provided only for convenience.

You can create an Opaque type for credentials used for basic authentication.

However, using the defined and public Secret type (kubernetes.io/basic-auth) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect.

The Kubernetes API verifies that the required keys are set for a Secret of this type.

SSH authentication Secrets

The builtin type kubernetes.io/ssh-auth is provided for storing data used in SSH authentication.

When using this Secret type, you will have to specify a ssh-privatekey key-value pair in the data (or stringData) field as the SSH credential to use.

The following manifest is an example of a Secret used for SSH public/private key authentication:

secret/ssh-auth-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
    UG91cmluZzYlRW1vdGljb24lU2N1YmE=
```

The SSH authentication Secret type is provided only for convenience.

You can create an Opaque type for credentials used for SSH authentication.

However, using the defined and public Secret type (kubernetes.io/ssh-auth) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect.

The Kubernetes API verifies that the required keys are set for a Secret of this type.

Caution: SSH private keys do not establish trusted communication between an SSH client and host server on their own.

A secondary means of establishing trust is needed to mitigate "man in the middle" attacks, such as a known_hosts file added to a ConfigMap.

TLS Secrets

The kubernetes.io/tls Secret type is for storing a certificate and its associated key that are typically used for TLS.

One common use for TLS Secrets is to configure encryption in transit for an Ingress, but you can also use it with other resources or directly in your workload.

When using this type of Secret, the tls.key and the tls.crt key must be provided in the data (or stringData) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

As an alternative to using stringData, you can use the data field to provide the base64 encoded certificate and private key.

The following YAML contains an example config for a TLS Secret:

secret/tls-auth-secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # values are base64 encoded, which obscures them but does NOT provide
  # any useful level of confidentiality
  tls.crt: |

    LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNVakNDQWJzQ0FnMytNQTBHQ1NxR1NJYjNE

    UUVCQlFVQU1JR2JNUXN3Q1FZRFZRUUdFd0pLVURFT01Bd0cKQTFVRUNCTUZWRzlyZVc4eEVEQU9C

    Z05WQkFjVEIwTm9kVzh0YTNVeEVUQVBCZ05WQkFvVENFWnlZVzVyVkVSQpNUmd3RmdZRFZRUUxF

    dzlYWldKRFpYSjBJRk4xY0hCdmNuUXhHREFXQmdOVkJBTVREMFp5
    5WVc1ck5FUkVJRmRsCllpQkRR
```

*VEVqTUNFR0NTcUdTSWIzRFFFSkFSWVVjM1Z3Y0c5eWRFQm1jbU
Z1YXpSa1pDNWpiMjB3SGhjTk1U*

*TXcKTVRFeE1EUTFNVE01V2hjTk1UZ3dNVEV3TURRMU1UTTVXak
JMTVFzd0NRWURWUVFHREFKS1VE*

*RVBNQTBHQTFVRQpDQXdHWEZSdmEzbHZNUkV3RHdZRFZRUUt
EQWhHY21GdWF6UkVSREVZTUJZR0Ex*

*VUVBd3dQZDNkM0xtVjRZVzF3CmJHVXVZMjl0TUlHYU1BMEdDU3
FHU0liM0RRRUJBUVVBQTRHSUFE*

*Q0JoQUo5WThFaUhmeHhNL25PbjJTbkkxWHgKRHdPdEJEVDFKRj
BReTliMVlKanV2YjdjaTEwZjVN*

*Vm1UQllqMUZTVWZNOU1vejJDVVFZdW4yRFljV29IcFA4ZQpqSG1
BUFVrNVd5cDJRN1ArMjh1bklI*

*QkphVGZlQ09PekZSUFY2MEdTWWUzNmFScG04L3dVVm16eGFLO
GtCOWVaCmhPN3F1TjdtSWQxL2pW*

*cTNKODhDQXdFQUFUQU5CZ2txaGtpRzl3MEJBUVVGQUFPQmdR
QU1meTQzeE15OHh3QTUKVjF2T2NS*

*OEtyNWNaSXdtbFhCUU8xeFEzazlxSGtyNFlUY1JxTVQ5WjVKTm1rW
HYxK2VSaGcwTi9WMW5NUTRZ*

*RgpnWXcxbnlESnBnOTduZUV4VzQyeXVlMFlHSDYyV1hYUUhyOVNV
REgrRlowVnQvRGZsdklVTWRj*

```
UUFEZjM4aU9zCjlQbG1kb3YrcE0vNCs5a1h5aDhSUEkzZXZ6OS9NQ
T09Ci0tLS0tRU5EIENFUlRJ
   RklDQVRFLS0tLS0K
 # In this example, the key data is not a real PEM-encoded private key
 tls.key: |
   RXhhbXBsZSBkYXRhIGZvciB0aGUgVExTIGNydCBmaWVsZA==
```

The TLS Secret type is provided only for convenience.

You can create an Opaque type for credentials used for TLS authentication.

However, using the defined and public Secret type (kubernetes.io/ssh-auth) helps ensure the consistency of Secret format in your project.

The API server verifies if the required keys are set for a Secret of this type.

To create a TLS Secret using kubectl, use the tls subcommand:

```
kubectl create secret tls my-tls-secret \
  --cert=path/to/cert/file \
  --key=path/to/key/file
```

The public/private key pair must exist before hand. The public key certificate for --cert must be .PEM encoded and must match the given private key for --key.

Bootstrap token Secrets

The bootstrap.kubernetes.io/token Secret type is for tokens used during the node bootstrap process.

It stores tokens used to sign well-known ConfigMaps.

A bootstrap token Secret is usually created in the kube-system namespace and named in the form bootstrap-token-<token-id> where <token-id> is a 6 character string of the token ID.

As a Kubernetes manifest, a bootstrap token Secret might look like the following:

secret/bootstrap-token-secret-base64.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: bootstrap-token-5emitj
  namespace: kube-system
type: bootstrap.kubernetes.io/token
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZWFkbTpkZWZhdWx0LW5vZGUtdG9rZW4=
  expiration: MjAyMC0wOS0xM1QwNDozOToxMFo=
  token-id: NWVtaXRq
  token-secret: a3E0Z2lodnN6emduMXAwcg==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
```

A bootstrap token Secret has the following keys specified under data:

- token-id: A random 6 character string as the token identifier. Required.
- token-secret: A random 16 character string as the actual token Secret. Required.
- description: A human-readable string that describes what the token is used for. Optional.
- expiration: An absolute UTC time using RFC3339 specifying when the token should be expired. Optional.
- usage-bootstrap-<usage>: A boolean flag indicating additional usage for the bootstrap token.
- auth-extra-groups: A comma-separated list of group names that will be authenticated as in addition to the system:bootstrappers group.

You can alternatively provide the values in the stringData field of the Secret without base64 encoding them:

secret/bootstrap-token-secret-literal.yaml

```
apiVersion: v1
kind: Secret
metadata:
  # Note how the Secret is named
  name: bootstrap-token-5emitj
  # A bootstrap token Secret usually resides in the kube-system namespace
  namespace: kube-system
type: bootstrap.kubernetes.io/token
```

```
stringData:
  auth-extra-groups: "system:bootstrappers:kubeadm:default-node-token"
  expiration: "2020-09-13T04:39:10Z"
  # This token ID is used in the name
  token-id: "5emitj"
  token-secret: "kq4gihvszzgn1p0r"
  # This token can be used for authentication
  usage-bootstrap-authentication: "true"
  # and it can be used for signing
  usage-bootstrap-signing: "true"
```

Note: The stringData field for a Secret does not work well with server-side apply.

Working with Secrets

Creating a Secret

There are several options to create a Secret:

- Use kubectl
- Use a configuration file
- Use the Kustomize tool

Constraints on Secret names and data

The name of a Secret object must be a valid DNS subdomain name.

You can specify the data and/or the stringData field when creating a configuration file for a Secret.

The data and the stringData fields are optional.

The values for all keys in the data field have to be base64-encoded strings.

If the conversion to base64 string is not desirable, you can choose to specify the stringData field instead, which accepts arbitrary strings as values.

The keys of data and stringData must consist of alphanumeric characters, -, _ or .. All key-value pairs in the stringData field are internally merged into the data field.

If a key appears in both the data and the stringData field, the value specified in the stringData field takes precedence.

Size limit

Individual Secrets are limited to 1MiB in size.

This is to discourage creation of very large Secrets that could exhaust the API server and kubelet memory.

However, creation of many smaller Secrets could also exhaust memory.

You can use a resource quota to limit the number of Secrets (or other resources) in a namespace.

Editing a Secret

You can edit an existing Secret unless it is immutable. To edit a Secret, use one of the following methods:

- Use kubectl
- Use a configuration file

You can also edit the data in a Secret using the Kustomize tool.

However, this method creates a new Secret object with the edited data.

Depending on how you created the Secret, as well as how the Secret is used in your Pods, updates to existing Secret objects are propagated automatically to Pods that use the data.

Using a Secret

Secrets can be mounted as data volumes or exposed as environment variables to be used by a container in a Pod.

Secrets can also be used by other parts of the system, without being directly exposed to the Pod.

For example, Secrets can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type Secret.

Therefore, a Secret needs to be created before any Pods that depend on it.

If the Secret cannot be fetched (perhaps because it does not exist, or due to a temporary lack of connection to the API server) the kubelet periodically retries running that Pod.

The kubelet also reports an Event for that Pod, including details of the problem fetching the Secret.

Optional Secrets

When you reference a Secret in a Pod, you can mark the Secret as *optional*, such as in the following example.

If an optional Secret doesn't exist, Kubernetes ignores it.

secret/optional-secret.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      optional: true
```

By default, Secrets are required. None of a Pod's containers will start until all non-optional Secrets are available.

If a Pod references a specific key in a non-optional Secret and that Secret does exist, but is missing the named key, the Pod fails during startup.

Using Secrets as files from a Pod

If you want to access data from a Secret in a Pod, one way to do that is to have Kubernetes make the value of that Secret be available as a file inside the filesystem of one or more of the Pod's containers.

For instructions, refer to Distribute credentials securely using Secrets.

When a volume contains data from a Secret, and that Secret is updated, Kubernetes tracks this and updates the data in the volume, using an eventually-consistent approach.

Note: A container using a Secret as a subPath volume mount does not receive automated Secret updates.

The kubelet keeps a cache of the current keys and values for the Secrets that are used in volumes for pods on that node.

You can configure the way that the kubelet detects changes from the cached values.

The configMapAndSecretChangeDetectionStrategy field in the kubelet configuration controls which strategy the kubelet uses. **The default strategy is Watch.**

Updates to Secrets can be either propagated by an API watch mechanism (the default), based on a cache with a defined time-to-live, or polled from the cluster API server on each kubelet synchronisation loop.

As a result, the total delay from the moment when the Secret is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (following the same order listed in the previous paragraph, these are: watch propagation delay, the configured cache TTL, or zero for direct polling).

Using Secrets as environment variables

To use a Secret in an environment variable in a Pod:

1. For each container in your Pod specification, add an environment variable for each Secret key that you want to use to the env[].valueFrom.secretKeyRef field.
2. Modify your image and/or command line so that the program looks for values in the specified environment variables.

For instructions, refer to Define container environment variables using Secret data.

Invalid environment variables

If your environment variable definitions in your Pod specification are considered to be invalid environment variable names, those keys aren't made available to your container. The Pod is allowed to start.

Kubernetes adds an Event with the reason set to InvalidVariableNames and a message that lists the skipped invalid keys. The following example shows a Pod that refers to a Secret named mysecret, where mysecret contains 2 invalid keys: 1badkey and 2alsobad.

```
kubectl get events
```

The output is similar to:

```
LASTSEEN    FIRSTSEEN    COUNT     NAME               KIND
SUBOBJECT                      TYPE     REASON
```

```
0s 0s 1 dapi-test-pod Pod Warning InvalidEnvironmentVariableNames kubelet, 127.0.0.1 Keys [1badkey, 2alsobad] from the EnvFrom secret default/mysecret were skipped since they are considered invalid environment variable names.
```

Container image pull Secrets

If you want to fetch container images from a private repository, you need a way for the kubelet on each node to authenticate to that repository. You can configure *image pull Secrets* to make this possible. These Secrets are configured at the Pod level.

Using imagePullSecrets

The imagePullSecrets field is a list of references to Secrets in the same namespace. You can use an imagePullSecrets to pass a Secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod. See the PodSpec API for more information about the imagePullSecrets field.

Manually specifying an imagePullSecret

You can learn how to specify imagePullSecrets from the container images documentation.

Arranging for imagePullSecrets to be automatically attached

You can manually create imagePullSecrets, and reference these from a ServiceAccount. Any Pods created with that ServiceAccount or created with that ServiceAccount by default, will get their imagePullSecrets field set to that of the service account. See Add ImagePullSecrets to a service account for a detailed explanation of that process.

Using Secrets with static Pods

You cannot use ConfigMaps or Secrets with static Pods.

IMMUTABLE SECRETS

FEATURE STATE: Kubernetes v1.21 [stable]

Kubernetes lets you mark specific Secrets (and ConfigMaps) as *immutable*. Preventing changes to the data of an existing Secret has the following benefits:

- protects you from accidental (or unwanted) updates that could cause applications outages
- (for clusters that extensively use Secrets - at least tens of thousands of unique Secret to Pod mounts), switching to immutable Secrets improves the performance of your cluster by significantly reducing load on kube-apiserver. The kubelet does not need to maintain a [watch] on any Secrets that are marked as immutable.

Marking a Secret as immutable

You can create an immutable Secret by setting the immutable field to true. For example,

```
apiVersion: v1
kind: Secret
metadata: ...
data: ...
immutable: true
```

You can also update any existing mutable Secret to make it immutable.

Note: Once a Secret or ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the data field. You can only delete and recreate the Secret. Existing Pods maintain a mount point to the deleted Secret - it is recommended to recreate these pods.

Information security for Secrets

Although ConfigMap and Secret work similarly, Kubernetes applies some additional protection for Secret objects.

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the Secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

A Secret is only sent to a node if a Pod on that node requires it. For mounting Secrets into Pods, the kubelet stores a copy of the data into a tmpfs so that the confidential data is not written to durable storage. Once the Pod that depends on the Secret is deleted, the kubelet deletes its local copy of the confidential data from the Secret.

There may be several containers in a Pod. By default, containers you define only have access to the default ServiceAccount and its related Secret. You must explicitly define environment variables or map a volume into a container in order to provide access to any other Secret.

There may be Secrets for several Pods on the same node. However, only the Secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the Secrets of another Pod.

DEPLOYING APPLICATIONS ON KUBERNETES

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It is a popular tool for container orchestration and provides a way to manage large numbers of containers as a single unit rather than having to manage each container individually.

Here's a basic overview of how to use Kubernetes:

- **Set up a cluster**: To use Kubernetes, you need to set up a cluster, which is a set of machines that run the Kubernetes control plane and the containers. You can set up a cluster on your own infrastructure or use a cloud provider such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure.
- **Package your application into containers**: To run your application on Kubernetes, you need to package it into one or more containers. A container is a standalone executable package that includes everything needed to run your application, including the code, runtime, system tools, libraries, and settings.
- **Define the desired state of your application using manifests**: Kubernetes uses manifests, which are files that describe the desired state of your application, to manage the deployment and scaling of your containers. The manifests specify the number of

replicas of each container, how they should be updated, and how they should communicate with each other.

- **Push your code to an SCM platform**: Push your application code to an SCM platform such as GitHub.
- **Use a CI/CD tool to automate**: Use a specialised CI/CD platform such as Harness to automate the deployment of your application. Once you set it up, done; you can easily and often deploy your application code in chunks whenever a new code gets pushed to the project repository.
- **Expose the application**: Once you deploy your application, you need to expose the application to the outside world by creating a Service with a type of LoadBalancer or ExternalName. This allows users to access the application through a stable IP address or hostname.
- **Monitor and manage your application**: After your application is deployed, you can use the kubectl tool to monitor the status of your containers, make changes to the desired state, and scale your application up or down.

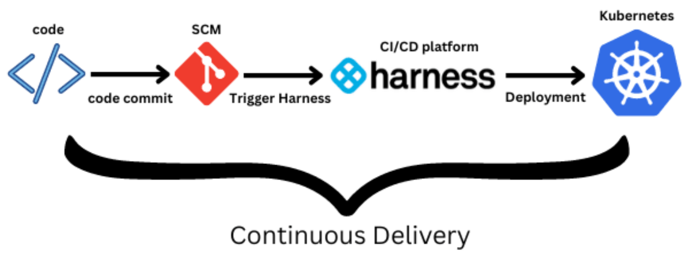
These are the general steps to deploy an application on Kubernetes. Depending on the application's complexity, additional steps may be required, such as configuring storage, network policies, or security. However, this should give you a good starting point for deploying your application on Kubernetes.

Today, we will see how to automate simple application deployment on Kubernetes using Harness.

Prerequisites

- Free Harness cloud account

- Download and install Node.js and npm
- GitHub account, we will be using our sample notes application
- Kubernetes cluster access, you can use Minikube or Kind to create a single-node cluster



We will use our sample application that is already in the GitHub repository. We will use a Kubernetes cluster to deploy our application. Next, we will use a CI/CD platform, Harness, in this tutorial to show how we can automate the software delivery process easily.

Step 1: Test the sample application locally

Go to the application folder with the following command

cd notes-app-cicd

Install dependencies with the following command

npm install

Run the application locally to see if the application works perfectly well

node app.js

Step 2: Containerize the application

You can see the Dockerfile in the sample application repository.

FROM node:14-alpine

WORKDIR /app

```
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm", "start" ]
```

Use the following command to build, tag and push the image to any container registry of your choice. We will push it to Docker Hub in this tutorial.

For Mac M1, use the following command

```
docker buildx build --platform=linux/arm64 --platform=linux/amd64 -t docker.io/$your docker hub user name/$image name:$tag name --push -f ./Dockerfile .
```

For other than Mac M1, use the below commands to build and push the image,

```
docker build -t $your docker hub user name/$image name .
docker push $your docker hub user name/$image name .
```

Step 3: Create or get access to a Kubernetes cluster

Make sure to have access to a Kubernetes cluster from any cloud provider. You can even use Minikube or Kind to create a cluster. In this tutorial, we are going to make use of a Kubernetes cluster from Google Cloud (GCP)

I already have an account on Google Cloud, so creating a cluster will be easy.

Step 4: Make sure the Kubernetes manifest files are neat and clean

You need deployment yaml and service yaml files to deploy and expose your application. Make sure both files are configured properly.

You can see that we have deployment.yaml and service.yaml file already present in the sample application repository.

Below is our deployment.yaml file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: notes-app-deployment
  labels:
    app: notes-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: notes-app
  template:
    metadata:
      labels:
        app: notes-app
    spec:
      containers:
      - name: notes-app-deployment
```

```
        image: pavansa/notes-app
        resources:
          requests:
            cpu: "100m"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 3000
```

Below is our service.yaml file

```
apiVersion: v1
# Indicates this as a service
kind: Service
metadata:
 # Service name
 name: notes-app-deployment
spec:
 selector:
   # Selector for Pods
   app: notes-app
 ports:
   # Port Map
 - port: 80
   targetPort: 3000
   protocol: TCP
 type: LoadBalancer
```

Apply the manifest files with the following commands. Starting with deployment and then service yaml file.

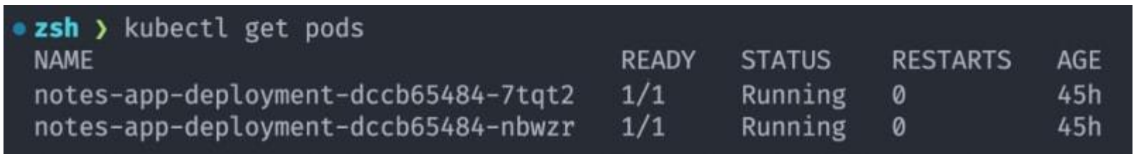
kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

Verify the pods are running properly as expected after applying the kubectl apply commands.

kubectl get pods

You should see the pods and their status.

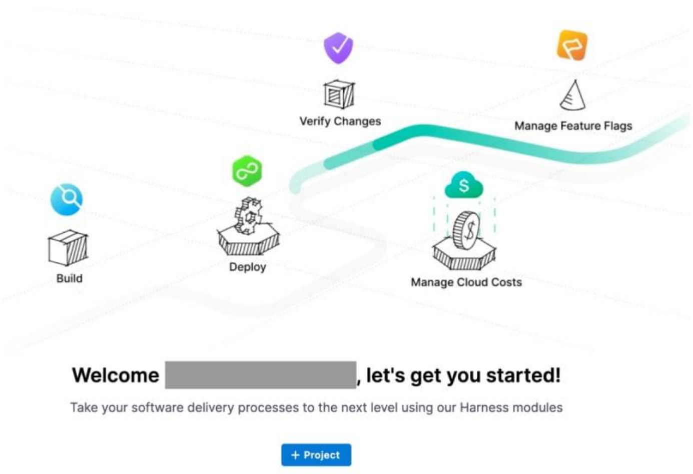


```
• zsh ❯ kubectl get pods
  NAME                                   READY   STATUS    RESTARTS   AGE
  notes-app-deployment-dccb65484-7tqt2   1/1     Running   0          45h
  notes-app-deployment-dccb65484-nbwzr   1/1     Running   0          45h
```

Step 5: Let’s automate the deployment using Harness

You need a CI/CD tool to automate your continuous integration and deployment process. Harness is known for its innovation and simplicity in the CI/CD space. Hence, we will use this platform to set up automated continuous deployment of our application.

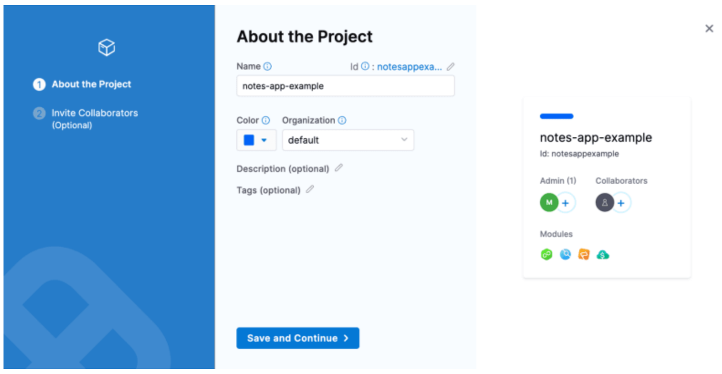
Once you sign up and verify your account, you will be presented with a welcome message and project creation set up. Proceed to create a project.



Welcome [redacted], let's get you started!

Take your software delivery processes to the next level using our Harness modules

Add the name to the project, save and continue.


About the Project

About the Project

Invite Collaborators (Optional)

Name: notes-app-example

Id: notesappexa...

Color: 🟦

Organization: default

Description (optional)

Tags (optional)

Save and Continue

notes-app-example

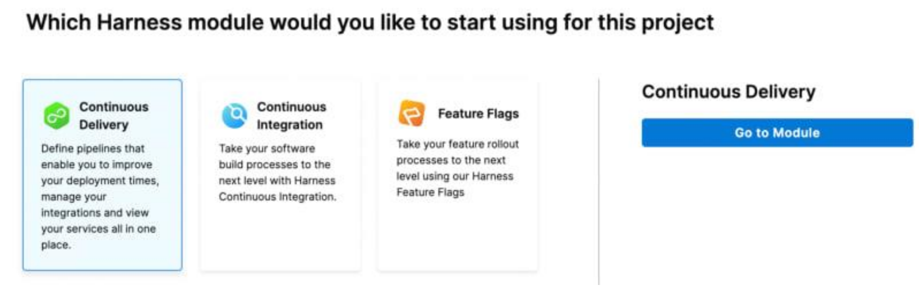
Id: notesappexample

Admin (1)

Collaborators

Modules

Select the ‘Continuous Delivery’ module and start your free plan.


Which Harness module would you like to start using for this project

Continuous Delivery
Define pipelines that enable you to improve your deployment times, manage your integrations and view your services all in one place.

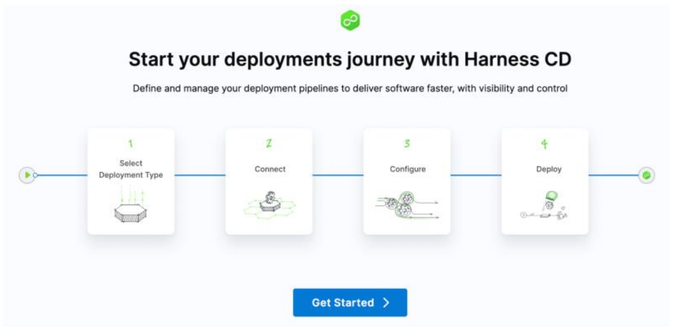
Continuous Integration
Take your software build processes to the next level with Harness Continuous Integration.

Feature Flags
Take your feature rollout processes to the next level using our Harness Feature Flags

Continuous Delivery

Go to Module

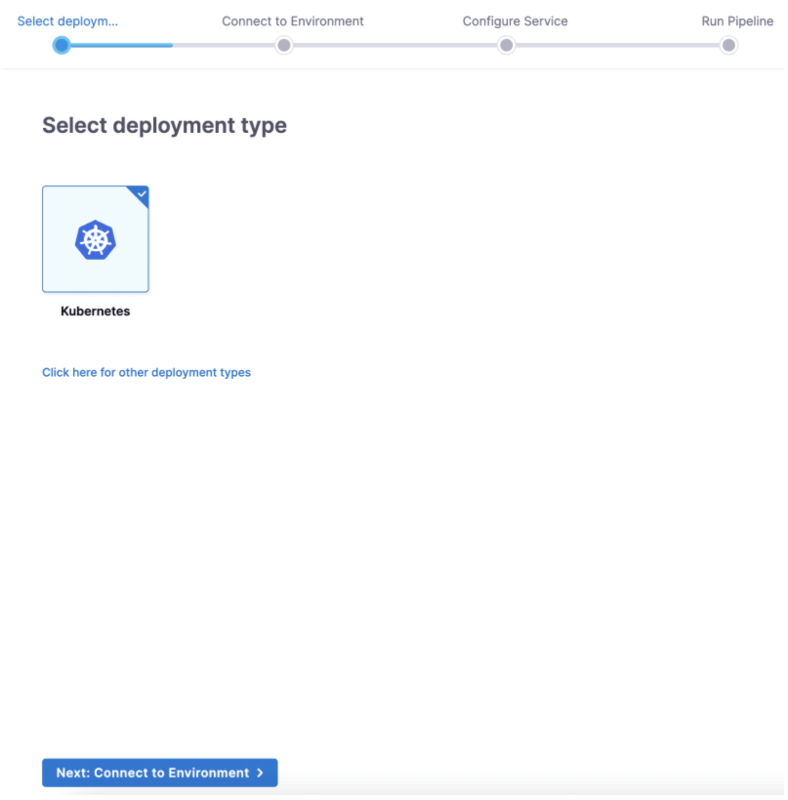
Go to the module and start your deployment journey.


Start your deployments journey with Harness CD

Define and manage your deployment pipelines to deliver software faster, with visibility and control

The set up is very straightforward, as shown in the above image; you can deploy your application in just four simple steps.

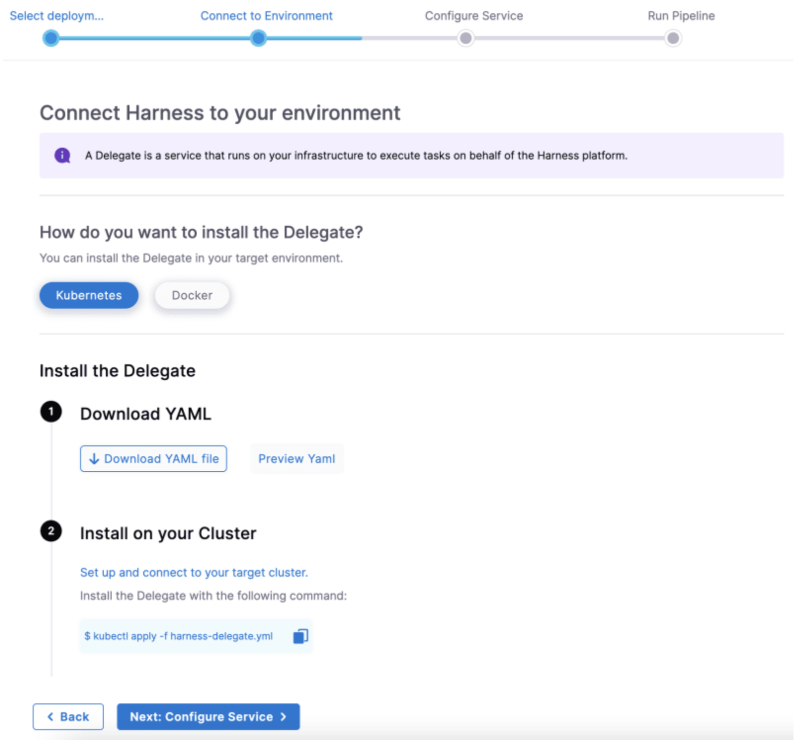
Select your deployment type i, e Kubernetes and click ‘Connect to Environment’.



Select deployment type

Kubernetes
Click here for other deployment types
Next: Connect to Environment

Connect to your Kubernetes environment with Delegate. A Delegate is a service that runs on your infrastructure to execute tasks on behalf of the Harness platform.



Connect Harness to your environment

A Delegate is a service that runs on your infrastructure to execute tasks on behalf of the Harness platform.

How do you want to install the Delegate?

You can install the Delegate in your target environment.

Install the Delegate

1 Download YAML

2 Install on your Cluster

Set up and connect to your target cluster.

Install the Delegate with the following command:

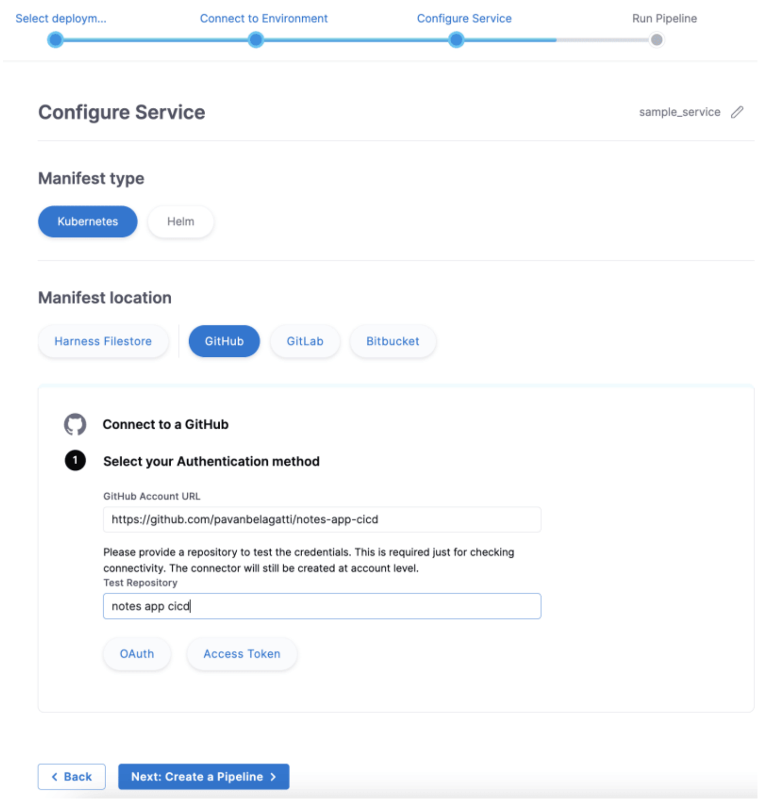
```
$ kubectl apply -f harness-delegate.yml
```

Download the Delegate YAML file and install it on your Kubernetes cluster by applying the kubectl apply command as stated in the above step.

Make sure to execute the command kubectl apply -f harness-delegate.yaml in the right path where you downloaded your delegate YAML file.

Ensure your Delegate installation is successful.

Next, configure the service and add the manifest details.

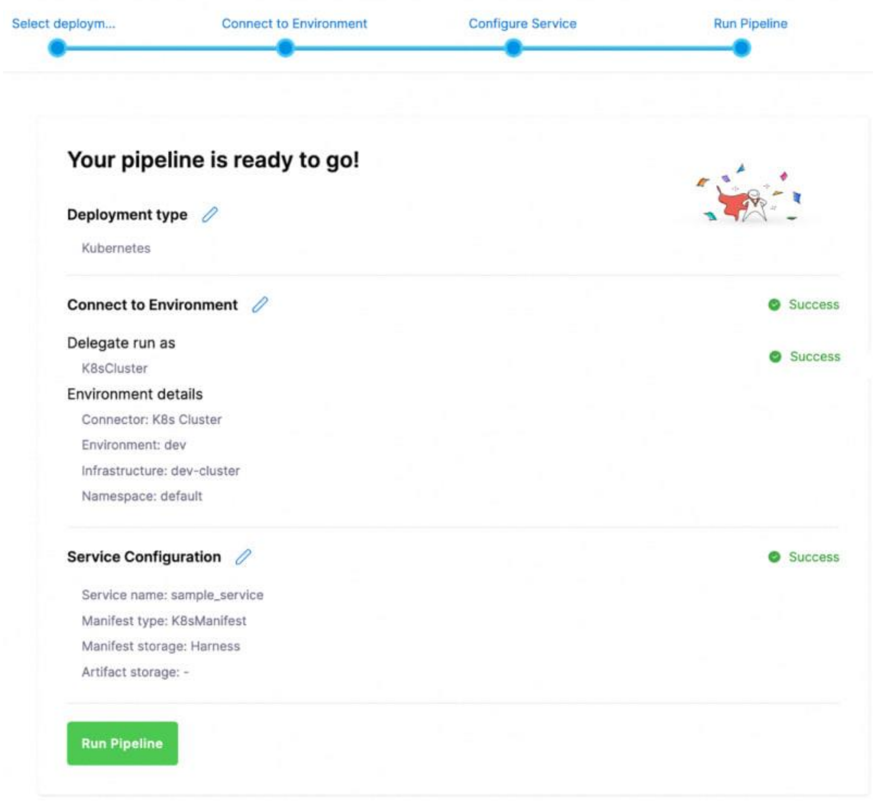


GitHub Account URL
https://github.com/pavanbelagatti/notes-app-cicd

Please provide a repository to test the credentials. This is required just for checking connectivity. The connector will still be created at account level.

Test Repository
notes app cicd

After adding all the details, click ‘Create a Pipeline’.



Your pipeline is ready to go!

Deployment type

Kubernetes

Connect to Environment ● Success

Delegate run as ● Success

K8sCluster

Environment details

Connector: K8s Cluster

Environment: dev

Infrastructure: dev-cluster

Namespace: default

Service Configuration ● Success

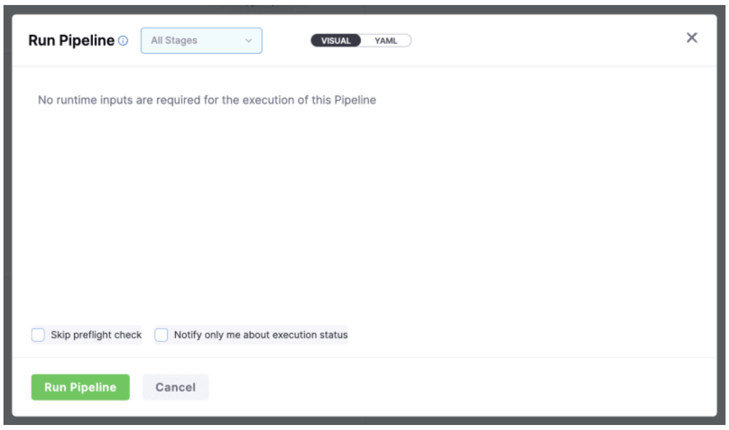
Service name: sample_service

Manifest type: K8sManifest

Manifest storage: Harness

Artifact storage: -

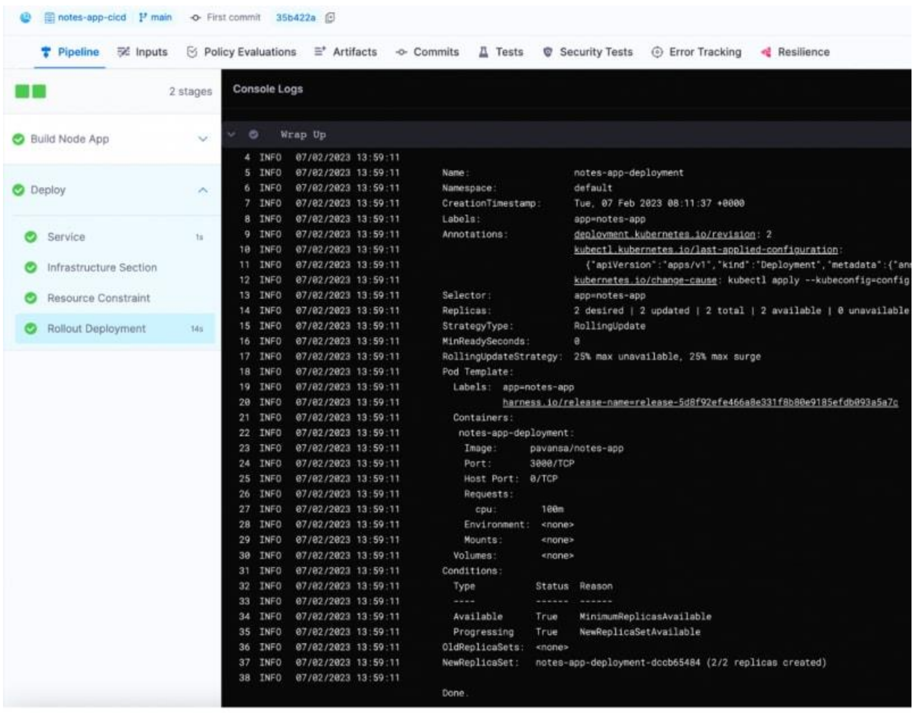
Check if all the connections are successful. Once everything looks fine, click on ‘Run Pipeline’.



No runtime inputs are required for the execution of this Pipeline

☐ Skip preflight check ☐ Notify only me about execution status

Click on ‘Run Pipeline’ to see the successful deployment.

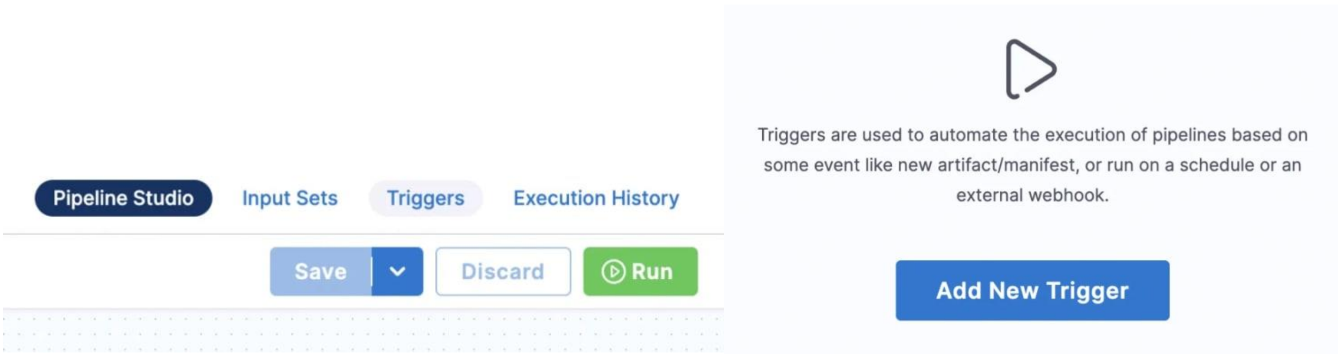


```
INFO  07/02/2023 13:59:11
INFO  07/02/2023 13:59:11       Name:                   notes-app-deployment
INFO  07/02/2023 13:59:11       Namespace:              default
INFO  07/02/2023 13:59:11       CreationTimestamp:      Tue, 07 Feb 2023 08:11:37 +0000
INFO  07/02/2023 13:59:11       Labels:                 app=notes-app
INFO  07/02/2023 13:59:11       Annotations:            deployment.kubernetes.io/revision: 2
INFO  07/02/2023 13:59:11                               kubectl.kubernetes.io/last-applied-configuration:
INFO  07/02/2023 13:59:11                                 {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"an
INFO  07/02/2023 13:59:11                               kubernetes.io/change-cause: kubectl apply --kubeconfig=config
INFO  07/02/2023 13:59:11       Selector:               app=notes-app
INFO  07/02/2023 13:59:11       Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
INFO  07/02/2023 13:59:11       StrategyType:           RollingUpdate
INFO  07/02/2023 13:59:11       MinReadySeconds:        0
INFO  07/02/2023 13:59:11       RollingUpdateStrategy:  25% max unavailable, 25% max surge
INFO  07/02/2023 13:59:11       Pod Template:
INFO  07/02/2023 13:59:11         Labels:  app=notes-app
INFO  07/02/2023 13:59:11                  harness.io/release-name=release-5d8f92efe466a8e331f8b80e9185efdb093a5a7c
INFO  07/02/2023 13:59:11         Containers:
INFO  07/02/2023 13:59:11          notes-app-deployment:
INFO  07/02/2023 13:59:11           Image:      pavansa/notes-app
INFO  07/02/2023 13:59:11           Port:       3000/TCP
INFO  07/02/2023 13:59:11           Host Port:  0/TCP
INFO  07/02/2023 13:59:11           Requests:
INFO  07/02/2023 13:59:11             cpu:        100m
INFO  07/02/2023 13:59:11           Environment:  <none>
INFO  07/02/2023 13:59:11           Mounts:       <none>
INFO  07/02/2023 13:59:11         Volumes:        <none>
INFO  07/02/2023 13:59:11       Conditions:
INFO  07/02/2023 13:59:11         Type           Status  Reason
INFO  07/02/2023 13:59:11         ----           ------  ------
INFO  07/02/2023 13:59:11         Available      True    MinimumReplicasAvailable
INFO  07/02/2023 13:59:11         Progressing    True    NewReplicaSetAvailable
INFO  07/02/2023 13:59:11       OldReplicaSets:  <none>
INFO  07/02/2023 13:59:11       NewReplicaSet:   notes-app-deployment-dccb65484 (2/2 replicas created)
INFO  07/02/2023 13:59:11
                                Done.
```

Congratulations! We successfully deployed our application successfully on Kubernetes using Harness. Now, we can easily automate the deployment using the Harness CD module.

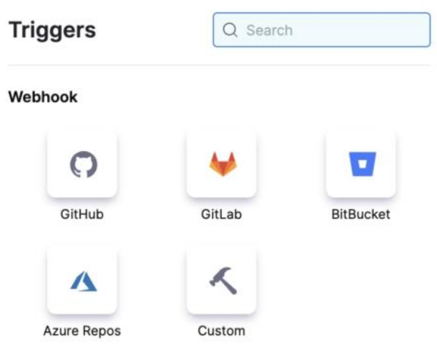
You can automate your CD process by adding Triggers. When any authorised person pushes any new code to your repository, your pipeline should get triggered and do CD. Let's see how to do that.

In the pipeline studio, you can click the 'Triggers' tab and add your desired trigger.

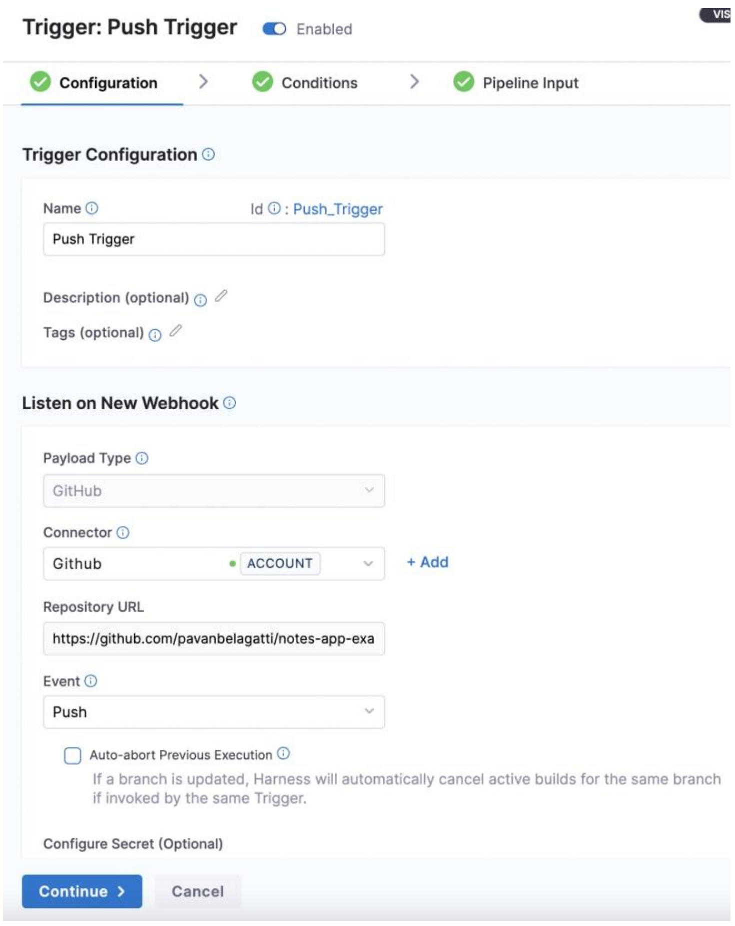


Triggers are used to automate the execution of pipelines based on some event like new artifact/manifest, or run on a schedule or an external webhook.

Click on 'Add New Trigger' and select 'GitHub'.



Add the required details and continue. As you can see, we are selecting ‘Push’ as our event. So whenever any authorised push happens to our repository, the pipeline should trigger and run.



Trigger: Push Trigger   Enabled

Trigger Configuration

Name: Push Trigger   Id: Push_Trigger

Description (optional)

Tags (optional)

Listen on New Webhook

Payload Type: GitHub

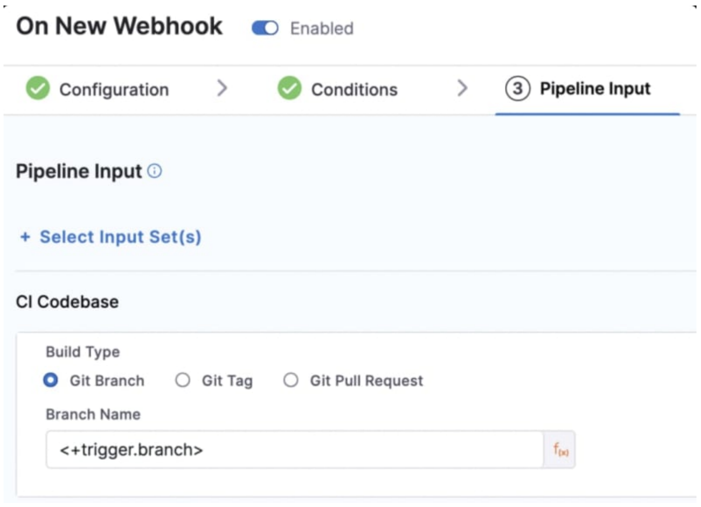
Connector: Github ACCOUNT   + Add

Repository URL: https://github.com/pavanbelagatti/notes-app-exa

Event: Push

Auto-abort Previous Execution
If a branch is updated, Harness will automatically cancel active builds for the same branch if invoked by the same Trigger.

Configure Secret (Optional)


On New Webhook Enabled

Configuration > Conditions > 3 Pipeline Input

Pipeline Input

+ Select Input Set(s)

CI Codebase

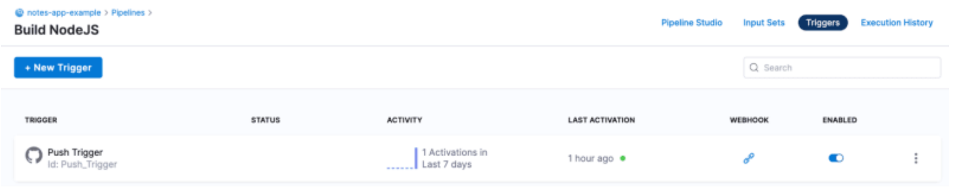
Build Type

[✓ Git Branch] Git Tag Git Pull Request

Branch Name

<+trigger.branch>

You can see your newly created trigger in the ‘Triggers’ tab.


notes-app-example > Pipelines >

Build NodeJS

Pipeline Studio Input Sets Triggers Execution History

+ New Trigger

TRIGGER	STATUS	ACTIVITY	LAST ACTIVATION	WEBHOOK	ENABLED
Push Trigger Id: Push_Trigger		1 Activations in Last 7 days	1 hour ago ●		

Now, whenever any authorised person pushes any code changes to your main/master repository, the pipeline automatically gets triggered.

If you have not seen my other two articles on continuous integration and automated testing, please take a look.

POD HEALTH CHECKING

Using health checks gives your Kubernetes services a solid foundation, better reliability and higher uptime.

Kubernetes is an open source container orchestration platform that significantly simplifies an application’s creation and management.

Distributed systems like Kubernetes can be hard to manage, as they involve many moving parts and all of them must work for the system to function.

Even if a small part breaks, it needs to be detected, routed and fixed.

These actions also need to be automated.

Kubernetes allows us to do that with the help of readiness and liveness probes.

What Is a Health Check

Health checks are a simple way to let the system know whether an instance of your app is working.

If the instance of your app is not working, the other services should not access it or send requests to it.

Instead, requests should be sent to another instance that is ready or you should retry sending requests.

The system should be able to bring your app to a healthy state.

By default, Kubernetes will start sending traffic to the pod when all the containers inside the pod have started.

Kubernetes will restart containers when they crash.

This default behavior should be enough to get started.

Making deployments more robust becomes relatively straightforward as

Kubernetes helps create custom health checks.

But before we do that, let’s discuss the pod life cycle.

Pod Life Cycle

A Kubernetes pod follows a defined life cycle. These are the different phases:

- When the pod is first created, it starts with a *pending* phase. The scheduler tries to figure out where to place the pod. If the scheduler can't find the node to place the pod, it will remain pending. (To check why the pod is in pending state, run the **kubectl describe pod <pod name>** command).
- Once the pod is scheduled, it goes to the container *creating* phase, where the images required for the application are pulled, and the container starts.
- Once the containers are in the pod, it moves to the *running* phase, where it continues until the program is completed successfully or terminated.

To check the status of the pod, run the **kubectl get pod** command and check the **STATUS** column.

As you can see, in this case all the pods are in *running* state.

Also, the **READY** column states the pod is ready to accept user traffic.

```
# kubectl get pod

NAME READY STATUS RESTARTS AGE

my-nginx-6b74b79f57-fldq6 1/1 Running 0 20s

my-nginx-6b74b79f57-n67wp 1/1 Running 0 20s

my-nginx-6b74b79f57-r6pcq 1/1 Running 0 20s
```

Different Types of Probes in Kubernetes

Kubernetes gives you the following types of health checks:

- **Readiness probes:** This probe will tell you when your app is ready to serve traffic. Kubernetes will ensure the readiness probe passes before allowing a service to send traffic to the pod. If the readiness probe fails, Kubernetes will not send the traffic to the pod until it passes.
- **Liveness probes:** Liveness probes will let Kubernetes know whether your app is healthy. If your app is healthy, Kubernetes will not interfere with pod functioning, but if it is unhealthy, Kubernetes will destroy the pod and start a new one to replace it.

To understand this further, let’s use a real-world scenario as an example.

You have an application that needs some time to warm up or download the application content from some external source like GitHub.

Your application shouldn’t receive traffic until it’s fully ready. By default, Kubernetes will start sending traffic as soon as the process inside the container starts.

Using the readiness probe, Kubernetes will wait until the app has fully started before it allows the service to send traffic to the new copy.

Let’s take another scenario where your application crashes due to a bug in code (maybe an edge case), and it hangs indefinitely and stops serving requests.

Because your process continues to run by default, Kubernetes will send traffic to the broken pod.

Using the liveness probes, Kubernetes will detect the app is no longer serving requests and restart the malfunctioning pod by default.

With the theory part done, let us see how to define the probes. There are three types of probes:

- HTTP
- TCP
- Command

Note: You have an option to start by defining either the readiness or liveness probes, as the implementation for both requires a similar template.

For example, if we first define livenessProbe, we can use it to define readinessProbe or vice-versa.

- **HTTP probes (httpGet):** This is the most common probe type. Even if your app isn't an HTTP server, you can usually create a lightweight HTTP server inside your app to respond to the liveness probe. Kubernetes will ping a path (for example, **/healthz**) at a given port (8080 in this example). If it gets an HTTP response in the 200 or 300 range, it will be marked as healthy. (For more information regarding HTTP response codes, refer to this link). Otherwise, it will be marked as unhealthy. Here is how you can define HTTP livelinessProbe:

```
livenessProbe:
httpGet:
path: /healthz
port: 8080
```

HTTP readiness probe is defined just like the HTTP livelinessProbe; you just have to replace liveness with readiness.

```
readinessProbe:
httpGet:
path: /healthz
port: 8080
```

- **TCP probes (tcpSocket):** With TCP probes, Kubernetes will try to establish a TCP connection on the specified port (for example, port 8080 in the below example). If it can establish a connection, the container is considered healthy. If it can’t, it’s considered a failure. These probes will be handy where HTTP or command probes don’t work well. For example, the FTP service will be able to use this type of probe.

```
readinessProbe:
tcpSocket:
port: 8080
```

- **Command probes (exec command):** In the case of commandprobes, Kubernetes will run a command inside your container. If the command returns an exit code zero, the container will be marked as healthy. Otherwise, it will be marked as unhealthy. This type of probe is useful when you can’t or don’t want to run an HTTP server, but you can run a command that will check whether your app is healthy. In the example below, we check whether the file **/tmp/healthy** exists, and if the command returns an exit code zero, the container will be marked as healthy; otherwise, it will be marked as unhealthy.

```
livenessProbe:
exec:
command:
- cat
- /tmp/healthy
```

Probes can be configured in many ways based on how often they need to run, the success and failure thresholds, and how long to wait for responses.

- **initialDelaySeconds (default value 0):** If you know your application needs n seconds (for example, 30 seconds) to warm up, you can add delay in seconds until the first check is executed by using **initialDelaySeconds**.
- **periodSeconds (default value 10):** If you want to specify how often you execute a check, you can define that using **periodSeconds**.
- **timeoutSeconds (default value 1):** This defines the maximum number of seconds until the probe operation is timed out.
- **successThreshold (default value 1):** This is the number of attempts until the probe is considered successful after the failure.
- **failureThreshold (default value 3):** In case of probe failure, Kubernetes makes multiple attempts before the probe is marked as failed.

Note: By default, the probe will stop if the application is not ready after three attempts.

In case of a liveness probe, it will restart the container.

In the case of a readiness probe, it will mark pods as unhealthy.

Let’s combine everything we have discussed so far. The key thing to note here is the use of readinessProbe with **httpGet**.

The first check will be executed after 10 seconds, and then it will be repeated after every 5 seconds.

```
apiVersion: v1
kind: Pod
metadata:
labels:
run: nginx
name: nginx
spec:
containers:
- image: nginx
name: nginx
readinessProbe:
httpGet:
path: /
port: 80
initialDelaySeconds: 10
periodSeconds: 5
```

- To create a pod, use the **kubectl create** command and specify the YAML manifest file with **-f** flag.

- You can give any name to the file, but it should end with a **.yaml** extension.

```
kubectl create -f readinessprobe.yaml
```

```
pod/nginx created
```

- If you check the pod’s status now, it should show the status as Running under the **STATUS** column. But if you check the **READY** column, it will still show 0/1, which means it’s not ready to accept a new connection.

```
kubectl get pod
```

```
NAME READY STATUS RESTARTS AGE
nginx 0/1 Running 0 16s
```

- Verify the status after a few seconds as we set the initial delay of a second. By now, the pod should be running.

```
kubectl get pod
```

```
NAME READY STATUS RESTARTS AGE
nginx 1/1 Running 0 28s
```

- To check the detailed status of all the parameters (for example, initialDelaySeconds, periodSeconds, etc.) used when defining readiness probe, run the **kubectl describe** command.

```
kubectl describe pod nginx |grep -i readiness
```

```
Readiness: http-get http://:80/ delay=10s timeout=1s
period=5s #success=1 #failure=3
```

Let's further reinforce the concept of liveness and readiness probe with the help of an example. First, let's start with a liveness probe. In the below example, we are executing a command, '**touch healthy; sleep 20; rm -rf healthy; sleep 600**'.

With this command, we have created a filename "healthy" using **touch** command. This file will exist in the container for the first 20 seconds, then it will be removed by using the **rm -rf** command. Lastly, the container will sleep for 600 seconds.

Then we defined the liveness probe. It first checks whether the file exists using the **cat healthy** command. It does that with an initial delay of five seconds. We further define the parameter **periodSeconds** which performs a liveness probe every five seconds. Once we delete the file, after 20 seconds the probe will be in a failed state.

```
apiVersion: v1
kind: Pod
metadata:
labels:
name: liveness-probe-exec
spec:
containers:
- name: liveness-probe
image: busybox
```

```
args:
- /bin/sh
- -c
- touch healthy; sleep 20; rm -rf healthy; sleep 600
livenessProbe:
exec:
command:
- cat
- healthy
initialDelaySeconds: 5
periodSeconds: 5
```

- To create a pod, store the above code in a file that ends with **.yaml** (for example, **liveness-probe.yaml**) and execute the **kubectl create** command with **-f <file name>**, which will create the pod.

```
# kubectl create -f liveness-probe.yaml
```

```
pod/liveness-probe-exec created
```

- Run the **kubectl get events** command, and you will see that the liveness probe has failed, and the container has been killed and restarted.

```
54s Normal Scheduled pod/liveness-probe-exec Successfully assigned default/liveness-probe-exec to controlplane
```

```
53s Normal Pulling pod/liveness-probe-exec Pulling image "busybox"
```

```
52s Normal Pulled pod/liveness-probe-exec Successfully pulled image "busybox" in 384.330188ms
52s Normal Created pod/liveness-probe-exec Created container liveness-probe
52s Normal Started pod/liveness-probe-exec Started container liveness-probe
18s Warning Unhealthy pod/liveness-probe-exec Liveness probe failed: cat: can't open 'healthy': No such file or directory
18s Normal Killing pod/liveness-probe-exec Container liveness-probe failed liveness probe, will be restarted
```

- You can also verify it by using the **kubectl get pods** command, and as you can see in the restart column, the container is restarted once.

```
# kubectl get pods
```

```
NAME READY STATUS RESTARTS AGE
liveness-probe-exec 1/1 Running 1 24s
```

- Now that you understand how the liveness probe works, let’s understand how the readiness probe works by tweaking the above example to define it as a readiness probe. In the example below, we execute a command inside the container (sleep 20; touch healthy; sleep 600), which first sleeps for 20 seconds, creates a file and finally sleeps for 600 seconds. As the initial delay is set to 15 seconds, the first check is executed with a delay of 15 seconds.

```
apiVersion: v1
kind: Pod
```

```
metadata:
labels:
name: readiness-probe-exec
spec:
containers:
- name: readiness-probe
image: busybox
args:
- /bin/sh
- -c
- sleep 20;touch healthy;sleep 600
readinessProbe:
exec:
command:
- cat
- healthy
initialDelaySeconds: 15
periodSeconds: 5
```

- To create a pod, store the above code in a file that ends with **.yaml**, and execute the **kubectl create** command, which will create the pod.

```
# kubectl create -f readiness-probe.yaml
```

```
pod/readiness-probe-exec created
```

- If you execute **kubectl get events** here, you will see that the probe failed as the file is not present.

```
63s Normal Scheduled pod/readiness-probe-exec Successfully assigned default/readiness-probe-exec to controlplane
62s Normal Pulling pod/readiness-probe-exec Pulling image "busybox"
62s Normal Pulled pod/readiness-probe-exec Successfully pulled image "busybox" in 156.57701ms
61s Normal Created pod/readiness-probe-exec Created container readiness-probe
61s Normal Started pod/readiness-probe-exec Started container readiness-probe
42s Warning Unhealthy pod/readiness-probe-exec Readiness probe failed: cat: can't open 'healthy': No such file or directory
```

If you check the status of the container initially, it is not in a ready state.

```
# kubectl get pods
NAME READY STATUS RESTARTS AGE
readiness-probe-exec 0/1 Running 0 5s
```

- But if you check it after 20 seconds, it should be in the running state.

```
# kubectl get pods
NAME READY STATUS RESTARTS AGE
readiness-probe-exec 1/1 Running 0 27s
```

Kubernetes - Kubectl

Kubectl is the command line utility to interact with Kubernetes API. It is an interface which is used to communicate and manage pods in Kubernetes cluster.

One needs to set up kubectl to local in order to interact with Kubernetes cluster.

Setting Kubectl

Download the executable to the local workstation using the curl command.

On Linux

```
$ curl -O https://storage.googleapis.com/kubernetesrelease/
release/v1.5.2/bin/linux/amd64/kubectl
```

On OS X workstation

```
$ curl -O https://storage.googleapis.com/kubernetesrelease/
release/v1.5.2/bin/darwin/amd64/kubectl
```

After download is complete, move the binaries in the path of the system.

```
$ chmod +x kubectl
$ mv kubectl /usr/local/bin/kubectl
```

Configuring Kubectl

Following are the steps to perform the configuration operation.

```
$ kubectl config set-cluster default-cluster --server = https://${MASTER_HOST} --
certificate-authority = ${CA_CERT}
```

```
$ kubectl config set-credentials default-admin --
certificateauthority = ${
CA_CERT} --client-key = ${ADMIN_KEY} --clientcertificate =
${
ADMIN_CERT}
```

```
$ kubectl config set-context default-system --cluster = default-
cluster --
user = default-admin
$ kubectl config use-context default-system
```

- Replace **${MASTER_HOST}** with the master node address or name used in the previous steps.
- Replace **${CA_CERT}** with the absolute path to the **ca.pem** created in the previous steps.
- Replace **${ADMIN_KEY}** with the absolute path to the **admin-key.pem** created in the previous steps.
- Replace **${ADMIN_CERT}** with the absolute path to the **admin.pem** created in the previous steps.

Verifying the Setup

To verify if the **kubectl** is working fine or not, check if the Kubernetes client is set up correctly.

```
$ kubectl get nodes
```

```
NAME    LABELS                          STATUS
```

Vipin.com Kubernetes.io/hostname = vipin.mishra.com   Ready

Kubernetes - Kubectl Commands

Kubectl controls the Kubernetes Cluster. It is one of the key components of Kubernetes which runs on the workstation on any machine when the setup is done. It has the capability to manage the nodes in the cluster.

Kubectl commands are used to interact and manage Kubernetes objects and the cluster. In this chapter, we will discuss a few commands used in Kubernetes via kubectl.

kubectl annotate − It updates the annotation on a resource.

```
$kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME)
KEY_1=VAL_1 ...
KEY_N = VAL_N [--resource-version = version]
```

For example,

```
kubectl annotate pods tomcat description = 'my frontend'
```

kubectl api-versions − It prints the supported versions of API on the cluster.

```
$ kubectl api-version;
```

kubectl apply − It has the capability to configure a resource by file or stdin.

```
$ kubectl apply –f <filename>
```

kubectl attach − This attaches things to the running container.

```
$ kubectl attach <pod> –c <container>
$ kubectl attach 123456-7890 -c tomcat-conatiner
```

kubectl autoscale − This is used to auto scale pods which are defined such as Deployment, replica set, Replication Controller.

```
$ kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min = MINPODS] --
max = MAXPODS [--cpu-percent = CPU] [flags]
$ kubectl autoscale deployment foo --min = 2 --max = 10
```

kubectl cluster-info − It displays the cluster Info.

```
$ kubectl cluster-info
```

kubectl cluster-info dump − It dumps relevant information regarding cluster for debugging and diagnosis.

```
$ kubectl cluster-info dump
$ kubectl cluster-info dump --output-directory = /path/to/cluster-
state
```

kubectl config − Modifies the kubeconfig file.

```
$ kubectl config <SUBCOMMAD>
$ kubectl config –-kubeconfig <String of File name>
```

kubectl config current-context − It displays the current context.

```
$ kubectl config current-context
#deploys the current context
```

kubectl config delete-cluster − Deletes the specified cluster from kubeconfig.

```
$ kubectl config delete-cluster <Cluster Name>
```

kubectl config delete-context − Deletes a specified context from kubeconfig.

```
$ kubectl config delete-context <Context Name>
```

kubectl config get-clusters − Displays cluster defined in the kubeconfig.

$ kubectl config get-cluster

$ kubectl config get-cluster <Cluser Name>

kubectl config get-contexts − Describes one or many contexts.

$ kubectl config get-context <Context Name>

kubectl config set-cluster − Sets the cluster entry in Kubernetes.

$ kubectl config set-cluster NAME [--server = server] [--certificateauthority =

path/to/certificate/authority] [--insecure-skip-tls-verify = true]

kubectl config set-context − Sets a context entry in kubernetes entrypoint.

$ kubectl config set-context NAME [--cluster = cluster_nickname] [--

user = user_nickname] [--namespace = namespace]

$ kubectl config set-context prod –user = vipin-mishra

kubectl config set-credentials − Sets a user entry in kubeconfig.

$ kubectl config set-credentials cluster-admin --username = vipin --

password = uXFGweU9l35qcif

kubectl config set − Sets an individual value in kubeconfig file.

$ kubectl config set PROPERTY_NAME PROPERTY_VALUE

kubectl config unset − It unsets a specific component in kubectl.

$ kubectl config unset PROPERTY_NAME PROPERTY_VALUE

kubectl config use-context − Sets the current context in kubectl file.

$ kubectl config use-context <Context Name>

kubectl config view

$ kubectl config view

$ kubectl config view –o jsonpath='{.users[?(@.name == "e2e")].user.password}'

kubectl cp − Copy files and directories to and from containers.

$ kubectl cp <Files from source> <Files to Destinatiion>

$ kubectl cp /tmp/foo <some-pod>:/tmp/bar -c <specific-container>

kubectl create − To create resource by filename of or stdin. To do this, JSON or YAML formats are accepted.

$ kubectl create –f <File Name>

$ cat <file name> | kubectl create –f -

In the same way, we can create multiple things as listed using the **create** command along with **kubectl**.

- deployment
- namespace
- quota
- secret docker-registry
- secret
- secret generic
- secret tls

- serviceaccount
- service clusterip
- service loadbalancer
- service nodeport

kubectl delete − Deletes resources by file name, stdin, resource and names.

$ kubectl delete –f ([-f FILENAME] | TYPE [(NAME | -l label | --all)])

kubectl describe − Describes any particular resource in kubernetes. Shows details of resource or a group of resources.

$ kubectl describe <type> <type name>

$ kubectl describe pod tomcat

kubectl drain − This is used to drain a node for maintenance purpose. It prepares the node for maintenance. This will mark the node as unavailable so that it should not be assigned with a new container which will be created.

$ kubectl drain tomcat –force

kubectl edit − It is used to end the resources on the server. This allows to directly edit a resource which one can receive via the command line tool.

$ kubectl edit <Resource/Name | File Name)

Ex.

$ kubectl edit rc/tomcat

kubectl exec − This helps to execute a command in the container.

$ kubectl exec POD <-c CONTAINER > -- COMMAND < args...>

$ kubectl exec tomcat 123-5-456 date

kubectl expose − This is used to expose the Kubernetes objects such as pod, replication controller, and service as a new Kubernetes service. This has the capability to expose it via a running container or from a **yaml** file.

$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol = TCP|UDP]

[--target-port = number-or-name] [--name = name] [--external-ip = external-ip-ofservice]

[--type = type]

$ kubectl expose rc tomcat –-port=80 –target-port = 30000

$ kubectl expose –f tomcat.yaml –port = 80 –target-port =

kubectl get − This command is capable of fetching data on the cluster about the Kubernetes resources.

$ kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|custom-columnsfile=...|

go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...]

(TYPE [NAME | -l label] | TYPE/NAME ...) [flags]

For example,

$ kubectl get pod <pod name>

$ kubectl get service <Service name>

kubectl logs − They are used to get the logs of the container in a pod. Printing the logs can be defining the container name in the pod. If the POD has only one container there is no need to define its name.

```
$ kubectl logs [-f] [-p] POD [-c CONTAINER]
```

Example

```
$ kubectl logs tomcat.
$ kubectl logs –p –c tomcat.8
```

kubectl port-forward − They are used to forward one or more local port to pods.

```
$ kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT
[...[LOCAL_PORT_N:]REMOTE_PORT_N]
$ kubectl port-forward tomcat 3000 4000
$ kubectl port-forward tomcat 3000:5000
```

kubectl replace − Capable of replacing a resource by file name or **stdin**.

```
$ kubectl replace -f FILENAME
$ kubectl replace –f tomcat.yml
$ cat tomcat.yml | kubectl replace –f -
```

kubectl rolling-update − Performs a rolling update on a replication controller. Replaces the specified replication controller with a new replication controller by updating a POD at a time.

```
$ kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --
image = NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC)
$ kubectl rolling-update frontend-v1 –f freontend-v2.yaml
```

kubectl rollout − It is capable of managing the rollout of deployment.

$ Kubectl rollout <Sub Command>

$ kubectl rollout undo deployment/tomcat

Apart from the above, we can perform multiple tasks using the rollout such as −

- rollout history
- rollout pause
- rollout resume
- rollout status
- rollout undo

kubectl run − Run command has the capability to run an image on the Kubernetes cluster.

$ kubectl run NAME --image = image [--env = "key = value"] [--port = port] [--

replicas = replicas] [--dry-run = bool] [--overrides = inline-json] [--command] --

[COMMAND] [args...]

$ kubectl run tomcat --image = tomcat:7.0

$ kubectl run tomcat –-image = tomcat:7.0 –port = 5000

kubectl scale − It will scale the size of Kubernetes Deployments, ReplicaSet, Replication Controller, or job.

$ kubectl scale [--resource-version = version] [--current-replicas = count] --

replicas = COUNT (-f FILENAME | TYPE NAME)

$ kubectl scale –-replica = 3 rs/tomcat

$ kubectl scale –replica = 3 tomcat.yaml

kubectl set image − It updates the image of a pod template.

$ kubectl set image (-f FILENAME | TYPE NAME)

CONTAINER_NAME_1 = CONTAINER_IMAGE_1 ... CONTAINER_NAME_N = CONTAINER_IMAGE_N

$ kubectl set image deployment/tomcat busybox = busybox ngnix = ngnix:1.9.1

$ kubectl set image deployments, rc tomcat = tomcat6.0 --all

kubectl set resources − It is used to set the content of the resource. It updates resource/limits on object with pod template.

$ kubectl set resources (-f FILENAME | TYPE NAME) ([--limits = LIMITS & --

requests = REQUESTS]

$ kubectl set resources deployment tomcat -c = tomcat --

limits = cpu = 200m,memory = 512Mi

kubectl top node − It displays CPU/Memory/Storage usage. The top command allows you to see the resource consumption for nodes.

$ kubectl top node [node Name]

The same command can be used with a pod as well.

Cloud Application Component Architecture

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

A cloud native application is designed to run on a cloud native infrastructure platform with the following four key traits:

- **Cloud native applications are resilient**. Resiliency is achieved when failures are treated as the norm rather than something to be avoided. The application takes advantage of the dynamic nature of the platform and should be able to recover from failure.
- **Cloud native applications are agile**. Agility allows the application to be deployed quickly with short iterations. Often this requires applications to be written as microservices rather than monoliths, but having microservices is not a requirement for cloud native applications.
- **Cloud native applications are operable**. Operability concerns itself with the qualities of a system that make it work well over its lifetime, not just at deployment phase. An operable application is not only reliable from the end-user point of view, but also from the vantage of the operations team. Examples of

operable software is one which operates without needing application restarts or server reboots, or hacks and workarounds that are required to keep the software running. Often this means that the application itself should expose a health check in order for the infrastructure it is running on to query the state of the application.

- **Cloud native applications are observable**. Observability provides answers to questions about application state. Operators and engineers should not need to make conjectures about what is going on in the application. Application logging and metrics are key to making this happen.

The above list suggests that cloud native applications impact the infrastructure that would be necessary to run such applications. Many responsibilities that have been traditionally handled by infrastructure have moved into the application realm.

Cloud Native Infrastructure Flavors

The ephemeral nature of the cloud demands automated development workflows that can be deployed and redeployed as needed. Cloud-native applications must be designed with infrastructure ambiguity in mind. This has led developers to rely on tools, like Docker, to provide a reliable platform to run their applications on without having to worry about the underlying resources. Influenced by Docker, developers have built applications with the microservices model, which enables highly focused, yet loosely coupled services that scale easily with demand.

Containerized Applications

Application containerization is a rapidly developing technology that is changing the way developers test and run application instances in the cloud.

The principal benefit of application containerization is that it provides a less resource-intensive alternative to running an application on a virtual machine. This is because application containers can share computational resources and memory without requiring a full operating system to underpin each application. Application containers house all the runtime components that are necessary to execute an application in an isolated environment, including files, libraries, and environment variables. With today's available containerization technology, users can run multiple isolated applications in separate containers that access the same OS kernel.

Serverless Applications

In the cloud computing world, it is often the case that the cloud provider provides the infrastructure necessary to run the applications to the users. The cloud provider takes care of all the headaches of running the server, dynamically managing the resources of the machine, etc. It also provides auto scalability, which means the serverless applications can be scaled based on the execution load they are exposed to. All these are done so that the user can solely focus on writing the code and leave worrying about these tasks to the cloud provider.

Serverless applications, also known as Function-as-a-Service or FaaS, is an offering from most of the enterprise cloud providers in which they allow the users to only write code and the infrastructure behind the scenes is managed by them. Users can write multiple functions to implement business logic and then all these functions can be easily integrated to talk to each other. Such a system that involves the use of the above model is known as serverless architecture.

Does Cloud Native Introduce a Cultural Change in Organizations?

Cloud Native is more than a tool set. It is a full architecture, a philosophical approach for building applications that take full advantage of cloud computing. In this context, culture is how individuals in an organization interact, communicate, and work with each other.

In short, culture is the way your enterprise goes about creating and delivering your service or product. If you have the perfect set of cloud platform, tools, and techniques yet go about using them wrong—if you apply the right tools, only in the wrong culture—it's simply not going to work. At best you'll be functional but far from capturing the value that the system you've just built can deliver. At worst, that system simply won't work at all.

The three major types of culture within enterprises are Waterfall, Agile, and Cloud Native. A quick primer:

Waterfall organizations are all about long-term planning. They deliver a large, solidly built and tested project bundling many separate services into one deployment every six months to one year (perhaps even longer) timeframe. They are risk-averse, with a long decision-making chain and rigid hierarchy. There are a lot of managers, including project managers. Waterfall organizations tend to have specialist teams handling very specific skills—security, testing, database administration, etc.

Agile organizations recognise the limitations imposed by monolithic long-term releases, and have adapted to deliver faster by using an iterative, feature-driven approach to releasing in two- to four-week ‘sprints’. Agile development breaks applications into their functional components, each of which is worked on, start to finish, by a single team.

Cloud Native organizations are built to take optimum advantage of functioning in cloud technologies (clouds will look quite different in the future, but we also build to anticipate this). Applications are built and deployed in a rapid cadence by small, dedicated feature teams made up of developers who also know how to build in networking, security, and all other necessities so all parts of the distributed system become part of the application.

Cultural awareness also grants the ability to start changing your organization from within, by a myriad of small steps all leading in the same direction. This allows you to evolve gradually and naturally alongside a rapidly changing world. ‘Long-term direction’ is literally the definition of strategy. So small steps towards this direction are strategic steps. Strategy is setting a long-term direction and taking steps on that direction.

Challenges of Cloud Native Apps

Below we cover a few challenges, which any organization should carefully consider before, and during, the move to cloud native.

Managing a CI/CD Pipeline for Microservices Applications

Microservices applications are composed of a large number of components, each of which could be managed by a separate team, and has its own development lifecycle. So instead of one CI/CD pipeline, like in a traditional monolithic app, in a microservices architecture there may be dozens or hundreds of pipelines.

This raises several challenges:

- Low visibility over quality of changes being introduced to each pipeline
- Limited ability to ensure each pipeline adheres to security and compliance requirements (see the following section)
- No central control over all the pipelines
- Duplication of infrastructure
- Lack of consistency in CI/CD practices – for example, one pipeline may have automated UI testing, while others may not

To address these challenges, teams need to work together to align on a consistent, secure approach to CI/CD for each microservice. At the organizational level, there should be a centralized infrastructure that provides CI/CD services for each team, allowing customization for the specific requirements of each microservice.

Cloud Native Security Challenges

Cloud native applications present tremendous challenges for security and risk professionals:

- **A larger number of entities to secure:** DevOps and infrastructure teams are leveraging microservices – using a combination of containers, Kubernetes and serverless functions – to run their cloud native applications. This growth is happening in conjunction with a constantly increasing cloud footprint. This combination leads to a larger number of entities to protect, both in production and across the application lifecycle.

- **Environments are constantly changing:** Public and private cloud environments are constantly changing due to the rapid-release cycles employed by today’s development and DevOps teams. As enterprises deploy weekly or even daily, this presents a challenge for security personnel looking to gain control over these deployments without slowing down release velocity.
- **Architectures are diverse:** Enterprises are using a wide-ranging combination of public and private clouds, cloud services and application architectures. Security teams are responsible for addressing this entire infrastructure and how any gaps impact visibility and security.
- **Networking is based on service identity:** Unlike traditional applications that use a physical or virtual machine as a stable reference point or node of a network, in cloud native applications different components might run in different locations, be replicated multiple times, be taken down and then get spun up elsewhere. This requires a network security model that understands the application context, the identity of microservices and their networking requirements, and builds a zero trust model around those requirements.

Learn More About Cloud Native Applications

Open Policy Agent: Authorization in a Cloud Native World

The Open Policy Agent (OPA) is a policy engine that automates and unifies the implementation of policies across IT environments, especially in cloud native applications. OPA was originally created by Styra, and has since been accepted by the Cloud Native Computing Foundation (CNCF). The OPA is offered for use under an open source license.

Learn about Open Policy Agent (OPA) and how you can use it to control authorization, admission, and other policies in cloud native environments, with a focus on K8s.

Getting started with IBM Cloud Kubernetes Service

IBM Cloud Kubernetes Service is a managed offering to create your own cluster of compute hosts where you can deploy and manage containerized apps on IBM Cloud. Combined with an intuitive user experience, built-in security and isolation, and advanced tools to secure, manage, and monitor your cluster workloads, you can rapidly deliver highly available and secure containerized apps in the public cloud.

Complete the following steps to get familiar with the basics, understand the service components, create your first cluster, and deploy a starter app.

Step 1: Review the basics

- Get an overview of the service by reviewing the concepts and terms, and benefits. For more information, see Understanding IBM Cloud Kubernetes Service.
- Review the FAQs

Already familiar with containers and IBM Cloud Kubernetes Service? Continue to the next step to prepare your account for creating clusters.

Step 2: Prepare your account

To set up your IBM Cloud account so that you can create clusters, see Preparing your account to create clusters.

If you've already prepared your account and you're ready to create a cluster, continue to the next step.

Step 3: Create a cluster

Follow a tutorial, or set up your own custom cluster environment.

- Tutorial Create a cluster in your own Virtual Private Cloud.
- Create a custom cluster on Classic infrastructure.
- Create a custom cluster on VPC infrastructure.

Already have a cluster? Continue to the next step to deploy an app!

Step 4: Deploy a sample app

After you create a cluster, deploy your first app. You can use a sample `websphere-liberty` Java application server that IBM provides and deploy the app to your cluster by using the Kubernetes dashboard.

1. Select your cluster from the cluster list.
2. Click **Kubernetes dashboard**.

3. Click the **Create new resource** icon (`+`) and select the **Create from form** tab.
    a. Enter a name for your app, such as `liberty`.
    b. Enter `websphere-liberty` for your container image. Remember that your cluster's VPC subnet must have a public gateway so that the cluster can pull an image from DockerHub.
    c. Enter the number of pods for your app deployment, such as `1`.
    d. From the **Service** drop-down menu, select **External** to create a `LoadBalancer` service that external users can use to access the app. Configure the external service as follows.
        - **Port**: `80`
        - **Target port**: `9080`
        - **Protocol**: `TCP`
4. Click **Deploy**. During the deployment, the cluster downloads the `websphere-liberty` container image from Docker Hub and deploys the app in your cluster. Your app is exposed by a Layer 4, version 1.0 network load balancer (NLB) so that it can be accessed by other users internally or externally. For other ways to expose an app such as Ingress, see Planning in-cluster and external networking for apps.

5. From the **Pods** menu, click your `liberty` pod and check that its status is **Running**.
6. From the **Services** menu, click the **External Endpoint** of your `liberty` service. For example, `169.xx.xxx.xxx:80` for classic clusters or `http://<hash>-<region>.lb.appdomain.cloud/` for VPC clusters. The **Welcome to Liberty** page is displayed.

Great job! You just deployed your first app in your Kubernetes cluster.

Benefits of using the Service

- **Choice of container platform provider:** Deploy clusters with **Red Hat OpenShift** or community **Kubernetes** installed as the container platform orchestrator.
  Choose the developer experience that fits your company, or run workloads across both Red Hat OpenShift or community Kubernetes clusters.
  Built-in integrations from the IBM Cloud console to the Kubernetes dashboard or Red Hat OpenShift web console.
  Single view and management experience of all your Red Hat OpenShift or community Kubernetes clusters from IBM Cloud. For more information, see Comparison between Red Hat OpenShift and community Kubernetes clusters.
- **Single-tenant Kubernetes clusters with compute, network, and storage infrastructure isolation:** Create your own customized infrastructure that meets the requirements of your organization.
  Choose between infrastructure providers.
  Provision a dedicated and secured Kubernetes master, worker nodes, virtual networks, and storage by using the resources provided by IBM Cloud infrastructure.
  Fully managed Kubernetes master that is continuously monitored and updated by IBM to keep your cluster available.
  Option to provision worker nodes as bare metal servers for compute-intensive workloads such as data, GPU, and AI.

Store persistent data, share data between Kubernetes pods, and restore data when needed with the integrated and secure volume service.
Benefit from full support for all native Kubernetes APIs.

- **Multizone clusters to increase high availability:** Easily manage worker nodes of the same flavor (CPU, memory, virtual or physical) with worker pools.
  Guard against zone failure by spreading nodes evenly across select multizones and by using anti-affinity pod deployments for your apps.
  Decrease your costs by using multizone clusters instead of duplicating the resources in a separate cluster.
  Benefit from automatic load balancing across apps with the multizone load balancer (MZLB) that is set up automatically for you in each zone of the cluster.
- **Highly available masters:** Reduce cluster downtime such as during master updates with highly available masters that are provisioned automatically when you create a cluster.
  Spread your masters across zones in a multizone cluster to protect your cluster from zonal failures.
- **Image security compliance with Vulnerability Advisor:** Set up your own repo in our secured Docker private image registry where images are stored and shared by all users in the organization.
  Benefit from automatic scanning of images in your private IBM Cloud registry.
  Review recommendations specific to the operating system used in the image to fix potential vulnerabilities

MICROSERVICES

Microservices architecture is a modern approach to software development where applications are broken down into small independent components. Kubernetes was designed to manage, deploy and scale applications built this way.

This article explores the pros and cons of microservices and explains why Kubernetes is an excellent tool for the job. It also provides best practices for enhancing the experience of deploying, scaling and managing microservices in Kubernetes.

ADVANTAGES OF MICROSERVICES

Microservices-based architectures enable developers to build powerful, highly efficient applications. The big advantages of the approach are:

- **Independent Deployment and Scaling:** Microservices allow individual components and services to be deployed and scaled independently without affecting the entire application. For instance, if a specific service requires more resources, it can be scaled up to meet demand without impacting other services. The opposite is also true; services can be scaled down to save cost when demand is low. The result is an architecture that better utilizes the available resources and can quickly respond to changing market conditions and customer needs.
- **Technology Stack Flexibility:** Each microservice can be built using the technologies, frameworks and languages that best suit application-specific requirements. A hypothetical e-commerce application, for example, might consist of the following microservices:

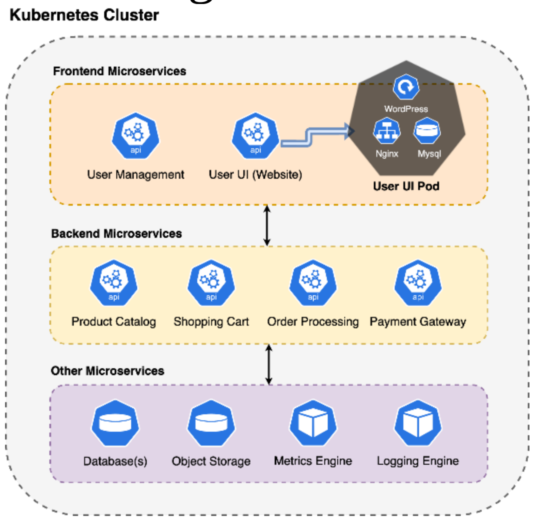


Diagram by Damaso Sanoja

The UI here uses WordPress, Nginx and MySQL to serve the website's pages. But that's not set in stone. Suppose you're not

comfortable developing PHP modules for WordPress. If that's the case, you can use a static website builder, such as Hugo, written in Go; Gatsby, written in React; or any other framework that best suits your team's skills.
Likewise, each API can be written in a different language. The Product Catalog API could be written in Python, the Shopping Cart API in Node.js and the Payment Gateway could use the SDK that best fits your needs.
This decoupling empowers developer teams to choose the most appropriate tools and technologies for each service, resulting in better performance and overall system efficiency.

- **Enhanced Fault Isolation:** Failure of a single component in a monolithic application usually affects the entire application. The advantage of a microservices-based architecture here is obvious.
  Developers can enhance fault isolation through mechanisms that minimize cascading failures, such as continuous integration/continuous delivery (CI/CD) pipelines that test each microservice before its deployed in production or probes that monitor microservice health. In mission-critical applications, developers can intentionally isolate critical services to facilitate debugging in case of failures.
  All this improves fault isolation, leading to greater application stability, less downtime and better user experience.
- **Easier Maintenance and Faster Development:** Breaking an application into small independent components simplifies maintenance and accelerates development. It reduces teams' dependency on each other, allowing them to work on different services concurrently. Since a single service's codebase is much smaller than an entire monolith's, troubleshooting and maintenance are far easier, creating a more maintainable application and reducing technical debt.

To summarize, the biggest benefits of a microservices-based architecture are in the areas of scaling, flexibility, fault isolation and development speed. However, as with all things in life, these benefits come at a cost.

DISADVANTAGES OF MICROSERVICES

Key drawbacks of a microservices-based architecture include:

- **Higher Complexity:** The divide-and-conquer premise of microservices inevitably increases complexity. Each microservice must be developed, deployed and managed individually, which can become cumbersome as the number of microservices grows. Additionally, your most effective developers will be those who are proficient in multiple technologies and languages, folks who can work effectively across multiple microservices.
- **Network Latency and Communication Overhead:** In a microservices-based architecture, components communicate with each other over a network. This can introduce latency and communication overhead, negatively impacting performance, particularly in high-traffic applications. Developers must also be able to account for potential network failures and design appropriate fallback mechanisms.
- **Data Consistency Challenges:** Ensuring data consistency across multiple microservices can be challenging. Implementing transactions and handling data synchronization across services requires careful planning and can lead to complex solutions, such as the saga pattern or event-driven architecture.
- **Deployment and Monitoring Complexities:** The sheer number of components involved in a microservices based application complicates deployment and monitoring. Orchestrating deployment across multiple services and

environments requires advanced tooling and expertise. Similarly, monitoring a distributed system demands a comprehensive approach to collecting and analyzing data from each service, which can be both resource-intensive and time-consuming.

In short, despite its many benefits, a microservices architecture is not a one-size-fits-all solution. Developers and businesses must weigh all these trade-offs before deciding if microservices are the right choice for a specific application.

Kubernetes and its capabilities should factor greatly in that decision. It addresses a lot of the inherent drawbacks in microservices architectures while upleveling their advantages.

WHY KUBERNETES IS A GREAT FIT FOR MICROSERVICES

Kubernetes features perfectly complement the microservices pattern, making it a great fit for this architecture. Let's unpack that statement.

- **Container Orchestration:** As discussed, the number of independent components running simultaneously complicates work with microservices architectures. Kubernetes addresses this complexity by managing the deployment and scaling of containers. It ensures that each microservice has the resources it needs to run effectively while also minimizing the operational overhead associated with manually managing multiple containers.
- **Scalability and Autoscaling:** One of the primary advantages of microservices is the ability to independently scale individual components. Kubernetes supports this through several scaling mechanisms. The most common is the Horizontal Pod

Autoscaler (HPA), which automatically adjusts the number of replicas of a specific pod based on demand.

Meanwhile, a DevOps team can implement the Vertical Pod Autoscaler (VPA) to set appropriate container resource limits based on live data and the Cluster Autoscaler (CA) to dynamically scale infrastructure to accommodate the fluctuating needs of a microservices-based application. (Here's a handy guide for using the Cluster Autoscaler.)

Combined, its autoscaling features allow Kubernetes to handle the complex workloads associated with microservices architectures in real time.

- **Self-Healing Capabilities:** Kubernetes's self-healing capabilities enable the enhanced fault isolation of microservices. Kubernetes automatically monitors pods' health and restarts failed containers, ensuring that the system remains operational overall even as some of its components fail. This significantly reduces the risk of system-wide outages, improving application resilience.
- **Load Balancing and Service Discovery:** Microservices often face network latency and communication overhead challenges. Kubernetes addresses these issues by offering built-in load balancing and service discovery. Load balancing distributes traffic evenly among microservice instances to avoid performance bottlenecks.

  Service discovery enables microservices to easily locate and communicate with one another through a centralized mechanism, reducing the complexity of interservice communication.
- **Declarative Configuration and Versioning:** Managing data consistency and deployment complexities can be challenging in a microservices environment. Kubernetes uses a declarative configuration approach, allowing developers to describe the system's desired state in configuration files. This approach

simplifies the deployment process and ensures that all components are in sync. Furthermore, versioning of configuration files helps track changes and enables rolling back to previous versions if needed, promoting a consistent and predictable deployment process.

- **High Availability:** High availability (HA) is a powerful feature that ensures continuous operation of a Kubernetes cluster and all the microservices that run on it despite failures. This is achieved by distributing and replicating resources, such as control plane components (i.e. API server, etcd, controller manager and scheduler), worker nodes and microservices across multiple nodes and availability zones. As a result, Kubernetes minimizes downtime, maintains consistent performance and ensures seamless user experience.

As you might expect, HA is crucial for mission-critical applications that require uninterrupted service and reliable infrastructure, making Kubernetes a great fit for this purpose.

To illustrate why Kubernetes is a really good option for deploying and managing microservices, let's look at an example.

MICROSERVICES ARCHITECTURE EXAMPLE

Recall our hypothetical e-commerce platform, where several microservices are running on the Kubernetes cluster.

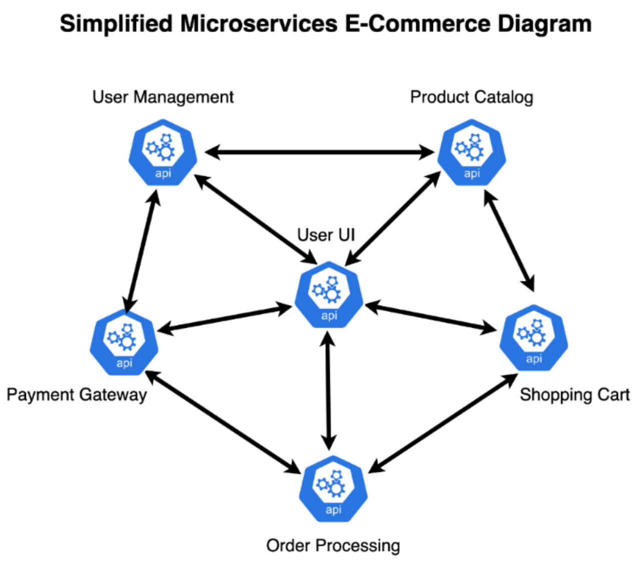


Diagram by Damaso Sanoja

In this simplified representation, you can see how the different microservices constantly interact with each other, forming a complicated network.

Now let’s see how Kubernetes can benefit this platform.

- **Frontend Microservices:** In this example, there are two microservices, User UI and User Management, responsible for managing the frontend and interacting with clients.
- **User UI:** The UI is the entry point for the platform’s visitors. It handles incoming HTTPS requests, routes them to the appropriate microservices and processes the responses. In other words, this microservice integrates the platform’s various services to ensure a seamless user experience.
  Given its importance, the UI must always be available. The HA functionality in Kubernetes makes that relatively simple. If necessary, you can even use a multicloud strategy to maximize performance and reliability for mission-critical apps and services.
- **User Management API:** The User Management API handles user authentication, registration and profile management. It communicates with the database backend to store and retrieve

user information. This microservice is essential to security and privacy of user data.
The flexible nature of the microservices architecture comes in handy here. For example, you could implement Auth0, an API designed with microservices in mind. It provides endpoints for users to securely log in, sign up or log out and supports multiple identity protocols, such as OpenID Connect, OAuth 2.0 and SAML.

- **Backend Microservices:** As often happens, the backend involves a greater number of microservices. To keep the example simple, we'll limit the discussion to the fictional microservices Product Catalog, Shopping Cart, Order Processing and Payment Gateway only. In a real production deployment the number of microservices can grow to include extra functionality, such as metrics and logs.
- **Product Catalog:** The Product Catalog microservice manages inventory. It provides APIs for adding, editing and removing products, as well as search and filtering. This service communicates with the frontend and the database backend to ensure that product information is up to date and consistent across the platform.
  Again, the microservices architecture's flexibility makes many solutions possible. One could be to include a Redis in-memory data store to get a fast, searchable product catalog. This would be a great user experience improvement.
- **Shopping Cart:** The Shopping Cart microservice in our example manages user sessions and their associated shopping carts. It allows users to add, remove and modify items in their cart while providing real-time pricing and availability updates. This service interacts primarily with the Product Catalog and User Management API to ensure a seamless user experience, something that requires great flexibility.

Because it’s a microservice, this API can be developed using different languages and frameworks that easily adapt to the rest of the components. Popular options include Nest.js, Spring Boot and .NET.

- **Order Processing:** The Order Processing microservice handles the entire order lifecycle, from checkout to fulfillment. It coordinates with the Shopping Cart, Payment Gateway and other relevant services to ensure that orders are processed correctly and efficiently. It also communicates with external shipping and logistics providers to facilitate order delivery. Given its crucial role, the scalability Kubernetes enables for microservices is especially beneficial for this microservice. Whenever e-commerce traffic spikes, say, during Christmas, Black Friday or Valentine's Day, Order Processing can scale up just enough to handle the demand.
- **Payment Gateway:** The Payment Gateway microservice manages the payment processing. It securely communicates with external payment providers (i.e. Visa or PayPal) to validate and process transactions.

Here Kubernetes shines again, thanks to its self healing capabilities. As you'll see shortly, developers can implement health checks in critical microservices like this one to ensure that transactions are as efficient and secure as possible.

Kubernetes enables developers to harness the full potential of microservices while minimizing their drawbacks. The truth is, however, that implementing Kubernetes itself comes with a hefty set of new challenges. Here are some best practices that help address those challenges.

Best Practices for Kubernetes and Microservices

In this section, you'll learn about five best practices to follow when running microservices on Kubernetes:

1. **Design Microservices for Failure:** Microservices should be designed to handle failures gracefully, ensuring that the entire system remains stable even during individual component failures.
   Developers should implement proper error handling and fallback mechanisms within the microservices. Coupled with Kubernetes's self-healing capabilities, this ensures that failed containers are rescheduled and restarted whenever necessary.
2. **Implement Proper Health Checks and Readiness Probes:** Health checks and readiness probes are crucial for maintaining the reliability and stability of microservices running on Kubernetes. Health checks allow Kubernetes to monitor the health of individual containers and take appropriate action whenever one becomes unresponsive.
   Readiness probes inform Kubernetes when a container is ready to start accepting traffic. By implementing these checks, Kubernetes can ensure that the only containers in use are containers that are healthy and ready, improving overall system resilience.
3. **Use Resource Quotas and Limits:** Resource quotas and limit ranges are essential for managing resources efficiently and preventing any individual microservice from consuming more resources than it needs.
   Kubernetes allows you to set resource quotas on namespaces, ensuring that each microservice has access to a fair share of resources. You can also define range limits on the resources that each container can use, preventing resource starvation and improving overall system stability.
4. **Implement Monitoring and Observability:** Monitoring and observability are vital for diagnosing issues, understanding system performance and making informed scaling and optimization decisions. Proper monitoring and logging solutions enable teams to collect and analyze metrics, logs and

traces and get valuable insights into system health and performance. This helps identify and resolve issues quickly and plan for future growth.

The Grafana observability stack is one monitoring and observability option that comes to mind, but there are dozens of observability, monitoring and analysis tools available.

5. **Secure Microservices with RBAC and Network Policies:** Cluster security is a top priority when deploying microservices on Kubernetes. Role-based access control (RBAC) and network policies enhance security of your microservices by controlling access to Kubernetes resources and limiting communication between different microservices.

RBAC allows you to define and enforce access control policies based on user roles, while network policies help restrict traffic between pods, ensuring that only authorized services can communicate with one another. (If you're interested in a deeper dive into Kubernetes security, there are other tasks that also help secure your Kubernetes cluster.)

MONOLITHIC APPLICATION

A monolithic application combines the user interface and data access layers for multiple features into one application. Usually, a monolithic application will exist as a single codebase that is modified by multiple teams within an organization, and be deployed as a single unit containing all the functionality that those teams maintain.

Monolithic applications can often be easier to develop and deploy thanks to the tight integration of their components. However, as the application's scope and performance demands increase, a monolith can consequently become difficult to maintain and scale.

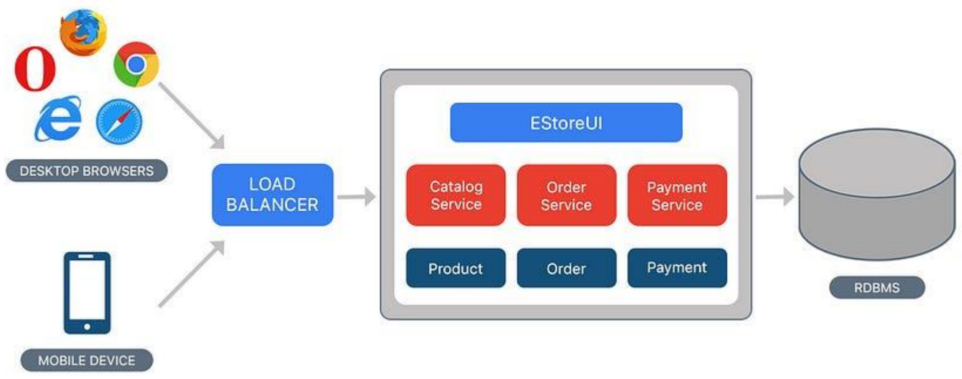
Monolith means composed all in one piece. The **Monolithic** application describes a single-tiered **software** application in which different components combined into a single program from a single platform. Components can be:

- Authorization — responsible for authorizing a user
- Presentation — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs).
- Business logic — the application's business logic.
- Database layer — data access objects responsible for accessing the database.
- Application integration — integration with other services (e.g. via messaging or REST API). Or integration with any other Data sources.
- Notification module — responsible for sending email notifications whenever needed.

Example for Monolithic Approach

Monolithic systems can be a good fit for smaller, less complex applications that don't need to quickly scale or be routinely maintained. Below are a few examples of applications that commonly have monolithic foundations (though their newer functionalities may be based on a more containerized infrastructure).

Ecommerce platforms – Monolithic applications are commonly seen in ecommerce because once the infrastructure is set up (building the online store, order processing, payment processing, and customer service), very few updates are required of the architecture.



Monolithic Architecture (for E-Commerce Application)

Content Management Systems (CMS) – This very glossary entry was made possible by a monolithic application (WordPress). CMS applications, when deployed, contain all the functionality necessary for content management in the form of web pages.

Banking systems – Many financial systems are built as monolithic applications because the points of entry are limited, making it more secure. Additionally, upon deployment, the codebase contains all the functionality a consumer expects from an online banking experience: managing financial transactions, payment processing, and tracking.

Advantages:

While some aspects of monolithic architectures have become outdated, there are still many purposes and positive attributes. Some advantages of monoliths include:

- **Simplicity** – The centralized architecture makes monoliths easier to develop, deploy, and maintain when compared to more complex architectures such as microservices architectures.
- **Faster testing** – By integrating every component into one program, a monolithic application can quickly be tested as a whole. Unlike architectures comprised of multiple, smaller

components (e.g., microservices), t here is no additional testing for complex communication protocols nor multiple code repositories.

- **Security –** With fewer points of entry for bad actors, monoliths are generally easier to secure. It is also easier to enforce security protocols across one application versus managing multiple security configurations.
- **Cost –** Deployed as a single unit, monoliths remove extra costs associated with deploying and securing additional communication protocols, building more connective infrastructure, and hiring employees with more specialized skills and training.

Drawbacks:

While the singular nature of monoliths has its positives, it can also lead to issues. Some disadvantages of monoliths include:

- **Complexity creep –** Over time, an application's growth and added functionality can cause a monolithic architecture to grow larger and more complex. This sprawl increases the risk that an update could compromise the single codebase that keeps the entire program running smoothly.
- **Lack of scalability –** When one feature or area of the application needs to scale horizontally, the entire large application (including subsystems that do not require additional resources) must be scaled. This can lead to both slowness in scaling since deployments take longer, as well as increased cost since each instance will have larger hardware requirements to run when compared to microservices.

- **Resilience –** In a monolithic architecture, all application components are tightly linked and run from a central codebase. Therefore, if one fails, the entire application could go down.
- **Full redeployment –** With a singular codebase representing the entire application, a monolith requires a full redeployment whenever a single component is changed or updated.
- **Technology stack –** Because developers have to make sure any tools or languages they use will fit in with the monolith, choice is restricted. Also, many monolithic applications are written in ways that aren’t completely compatible with newer, more efficient technologies like cloud computing and containerization.
- **Slower development –** When multiple development teams are working on one large codebase, extreme care is required to make sure interfaces and domain boundaries are respected and maintained. Sometimes, code can introduce complex coupling, making cross-team dependencies slow down the development of new features or fixes to critical issues.