BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Work Integrated Learning Program

M. Tech. Software Engineering

Open Source Software Engineering

SEZG587

Assignment – 1

Name – Hemant Tiwari

BITS Student ID – 2022MT93184

Semester – 2nd FY 2022-2023

Open-Source Key Project - Kubernetes Project

Introduction and History of Movement

Kubernetes is a portable, extensible, **open source platform** for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

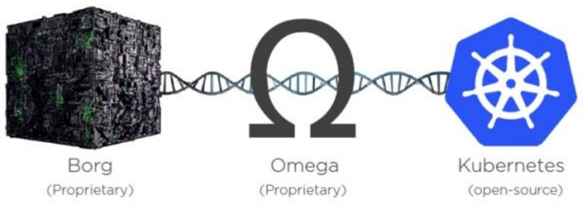
To begin our **Kubernetes history**, we must begin with **Google**. Google faced a major issue in the **early 2000s**. While they continued to dominate the majority of the digital world, they were still figuring out how to make revenue from their extensive number of free services. This included popular services like:

1. Google Search
2. YouTube
3. Gmail

The problem for Google, which was not the profit-making machine we know today, was finding a way to offer these free services without tanking the business. Keep in mind that it costs a ton to keep services like these up and running — especially if they are popular, which they were.

The difficulty for Google, which was not yet a profit-making machine, was figuring out how to deliver these free services without destroying the business. Keep in mind that maintaining services like this is expensive, especially if they are popular, as they were.

Google asked its team, "**How can we get maximum performance from our hardware?**" And the Story after that, **Kubernetes was born**.



2003-2004: Birth of the Borg System

- Google introduced the Borg System around 2003-2004. It started off as a small-scale project, with about 3-4 people initially in collaboration with a new version of Google's new search engine. Borg was a large-scale internal cluster management system, which ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines.

2013: From Borg to Omega

- Following Borg, Google introduced the Omega cluster management system, a flexible, scalable scheduler for large compute clusters.

2014: Google Introduces Kubernetes

- Google introduced Kubernetes as an *open-source version of Borg.*
- **June 7:** Initial release – first github commit for Kubernetes
- Microsoft, RedHat, IBM, Docker joins the Kubernetes community.

2015: The year of Kube v1.0 & CNCF

- Kubernetes v1.0 gets released. Along with the release, Google partnered with the Linux Foundation to form the Cloud Native Computing Foundation (CNCF). The CNFC aims to build sustainable ecosystems and to foster a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture.

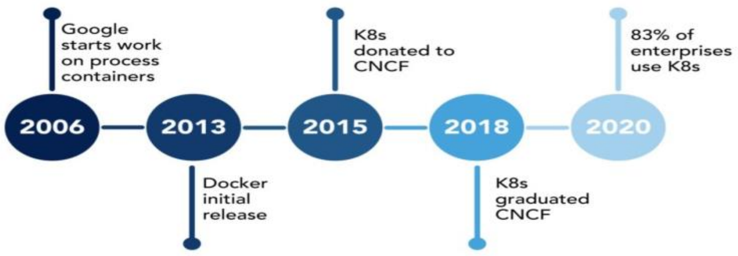
2016: The Year Kubernetes Goes Mainstream!

- First release of Helm, the package manager of Kubernetes.
- Official release of Minikube: a tool that makes it easy to run Kubernetes locally.
- Kubernetes 1.4 introduces a new tool, kubeadm, that helps improve Kubernetes' installability. This release provides easier setup, stateful application support with integrated Helm, and new cross-cluster federation features.
- Node feature discovery for Kubernetes Arrives – This package enables node feature discovery for Kubernetes.

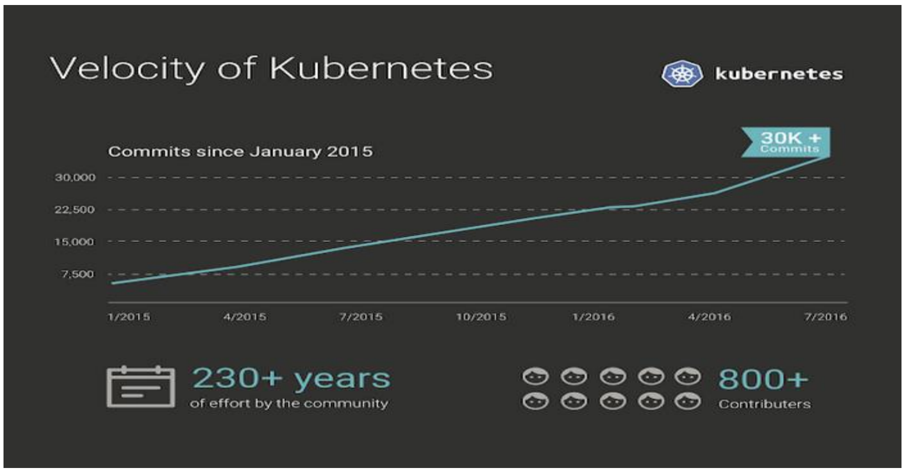
- Kubernetes 1.5 – Windows Server Support Comes to Kubernetes. The new features include containerized multiplatform applications, support for Windows server containers and hyper-V containers, an expanded ecosystem of applications, coverage for heterogeneous data centers, & more.
- Kubernetes supports OpenAPI, which allows API providers to define their operations & models and enables developers to automate their tools.

2017: The Year of Enterprise Adoption & Support

- Kubernetes 1.6 is a stabilization release. Specific updates: etcdv3 enabled by default, direct dependency on a single container runtime removed, RBAC in beta, automatic provisioning of Storage Class objects.



Today, Kubernetes has 1800+ contributors, 500+ meetups worldwide, and 42,000+ users (many of them joining the public #kubernetes-dev channel on Slack). 83% of enterprises surveyed by the Cloud Native Computing Foundation (CNCF) in 2020 are using Kubernetes.



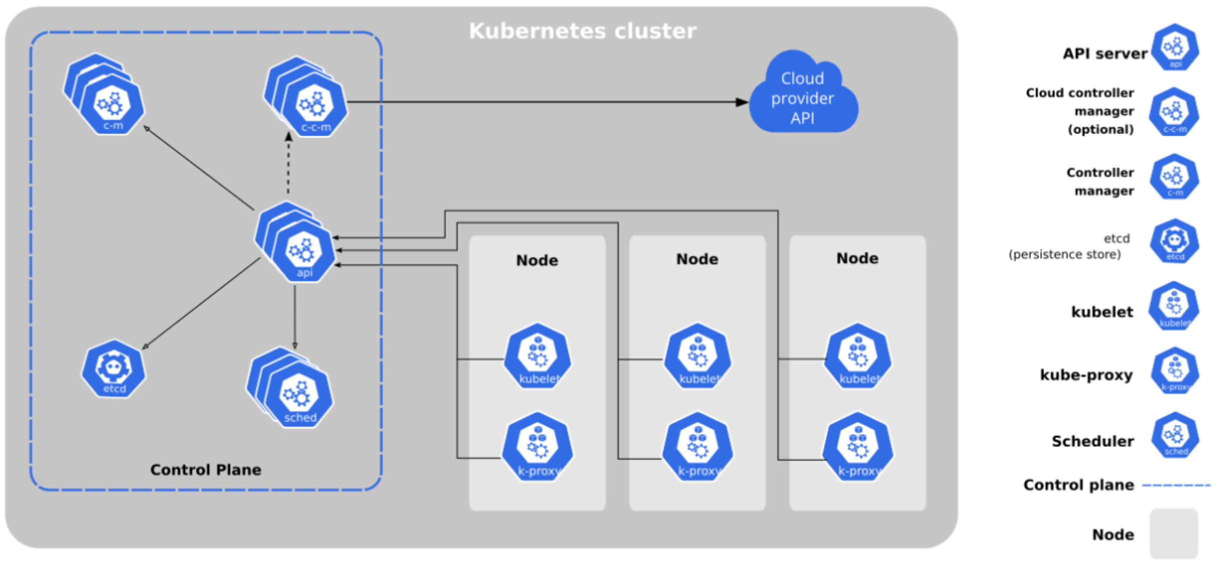
Kubernetes Components

When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster.

In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.



Control Plane Components -

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple machines.

kube-apiserver **-** The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

***etcd* -** Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
If your Kubernetes cluster uses etcd as its backing store, make sure you have a back-up plan for the data.

***kube-scheduler* -** Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

***kube-controller-manager* -** Control plane components that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.
Some types of these controllers are:

- *Node controller:* Responsible for noticing and responding when nodes go down.
- *Job controller:* Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- *Endpoint Slice controller:* Populates Endpoint Slice objects (to provide a link between Services and Pods).
- *Service Account controller:* Create default Service Accounts for new namespaces.

***cloud-controller-manager* -** A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.
The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.
As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process.

You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.
The following controllers can have cloud provider dependencies:

- *Node controller:* For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- *Route controller:* For setting up routes in the underlying cloud infrastructure
- *Service controller:* For creating, updating and deleting cloud provider load balancers

Node Components –

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

***kubelet* -** An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

***kube-proxy* -** It is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.
It maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

***Container runtime* -** The container runtime is the software that is responsible for running containers.
Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

Addons –

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

Selected addons are described below; for an extended list of available addons, please see Addons.

DNS - While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it. Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard) - Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring - Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging - A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

Kubernetes API, Kubernetes Objects, Pods

Kubernetes API –

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.
The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).
Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls.
Consider using one of the client libraries if you are writing an application using the Kubernetes API.

Complete API details are documented using OpenAPI.

OpenAPI V2 - The Kubernetes API server serves an aggregated OpenAPI v2 spec via the /openapi/v2 endpoint. You can request the response format using request headers as follows:

Header	Possible values	Notes
Accept-Encoding	gzip	not supplying this header is also acceptable
Accept	application/com.github.proto-openapi.spec.v2@v1.0+protobuf	mainly for intra-cluster use
	application/json	default
	*	serves application/json

Kubernetes implements an alternative Protobuf based serialization format that is primarily intended for intra-cluster communication. For more information about this format, see the Kubernetes Protobuf serialization design proposal and the Interface Definition Language (IDL) files for each schema located in the Go packages that define the API objects.

OpenAPI V3 - Kubernetes supports publishing a description of its APIs as OpenAPI v3. A discovery endpoint /openapi/v3 is provided to see a list of all group/versions available. This endpoint only returns JSON.

These group/versions are provided in the following format:

```
{
  "paths": {
    ...,
    "api/v1": {
      "serverRelativeURL":
"/openapi/v3/api/v1?hash=CC0E9BFD992D8C59AEC98A1E2336F899E8318D3CF4C68944C3DEC
640AF5AB52D864AC50DAA8D145B3494F75FA3CFF939FCBDDA431DAD3CA79738B297795818C
F"
    },
    "apis/admissionregistration.k8s.io/v1": {
      "serverRelativeURL":
"/openapi/v3/apis/admissionregistration.k8s.io/v1?hash=E19CC93A116982CE5422FC42B590A8
AFAD92CDE9AE4D59B5CAAD568F083AD07946E6CB5817531680BCE6E215C16973CD39003B04
25F3477CFD854E89A9DB6597"
    },
    ....
  }
}
```

The relative URLs are pointing to immutable OpenAPI descriptions, in order to improve client-side caching. The proper HTTP caching headers are also set by the API server for that purpose (Expires to 1 year in the future, and Cache-Control to immutable). When an obsolete URL is used, the API server returns a redirect to the newest URL.

The Kubernetes API server publishes an OpenAPI v3 spec per Kubernetes group version at the /openapi/v3/apis/<group>/<version>?hash=<hash> endpoint.

Refer to the table below for accepted request headers.

Header	Possible values	Notes
Accept-Encoding	gzip	*not supplying this header is also acceptable*
Accept	application/com.github.proto-openapi.spec.v3@v1.0+protobuf	*mainly for intra-cluster use*
	application/json	*default*
	*	*serves* application/json

A Golang implementation to fetch the OpenAPI V3 is provided in the package k8s.io/client-go/openapi3.

Kubernetes Objects –

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state.

To work with Kubernetes objects--whether to create, modify, or delete them--you'll need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

Object spec and status - Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the object spec and the object status. For objects that have a spec, you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its desired state.
The status describes the current state of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state you supplied.
For example: in Kubernetes, a Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment spec to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application--updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction--in this case, starting a replacement instance.

Describing a Kubernetes object - When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via kubectl), that API request must include that information as JSON in the request body. *Most often, you provide the information to kubectl in a .yaml file.* kubectl converts the information to JSON when making the API request.

Here's an example .yaml file that shows the required fields and object spec for a Kubernetes Deployment:

application/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

One way to create a Deployment using a .yaml file like the one above is to use the kubectl apply command in the kubectl command-line interface, passing the .yaml file as an argument. Here's an example:

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

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

Required fields – In the .yaml file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- apiVersion - Which version of the Kubernetes API you're using to create this object
- kind - What kind of object you want to create
- metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
- spec - What state you desire for the object

The precise format of the object spec is different for every Kubernetes object, and contains nested fields specific to that object. The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes.
For example, see the spec field for the Pod API reference. For each Pod, the .spec field specifies the pod and its desired state (such as the container image name for each container within that pod). Another example of an object specification is the spec field for the StatefulSet API. For StatefulSet, the .spec field specifies the StatefulSet and its desired state. Within the .spec of a StatefulSet is a template for Pod objects. That template describes Pods that the StatefulSet controller will create in order to satisfy the StatefulSet specification. Different kinds of object can also have different .status; again, the API reference pages detail the structure of that .status field, and its content for each different type of object.

Pods –

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain init containers that run during Pod startup. You can also inject ephemeral containers for debugging if your cluster offers this.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a container. Within a Pod's context, the individual applications may have further sub-isolations applied.
A Pod is similar to a set of containers with shared namespaces and shared filesystem volumes.

Using Pods –

The following is an example of a Pod which consists of a container running the image nginx:1.14.2.

pods/simple-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

To create the Pod shown above, run the following command:

```
kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

Pods are generally not created directly and are created using workload resources. See Working with Pods for more information on how Pods are used with workload resources.

Workload resources for managing pods

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Deployment or Job. If your Pods need to track state, consider the StatefulSet resource.

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.

- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate *sidecar* container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

Note: Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as *replication*. Replicated Pods are usually created and managed as a group by a workload resource and its controller.

See Pods and controllers for more information on how Kubernetes uses workload resources, and their controllers, to implement application scaling and auto-healing.