International Journal of Management, IT & Engineering
Vol. 14 Issue 12, December 2024,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Kubernetes Essentials: The Backbone of Modern Container Orchestration

Sankara Aditya Sarma Garimella[*]
Srivenkatesh Dudala[**]

Abstract

The rise of microservices has caused an increase in the usage of container technologies because the containers offer the perfect host for small independent applications like microservices. Managing containers across multiple environments using scripts and custom-made tools can quickly become complex and sometimes even unmanageable. That specific scenario has led to the need for container orchestration technologies. Kubernetes is one such container orchestration technology, and this paper explains the critical components, architecture, and K8s key features. It also provides a demo application setup with K8s components in a local minibuke cluster.

Keywords:

Kubernetes;
Containers;
Microservices;
K8s;
Orchestration;

Author correspondence:

Sankara Aditya Sarma Garimella,
M.S. Bioinformatics, The University of Texas at El Paso
Application Engineer, DFS Corporate Services, LLC, Texas, USA
Email: adi.garimella@gmail.com

Srivenkatesh Dudala
Bachelor Of Engineering, Andhra University, India
Technical Program Manager, Amazon.Com Services, LLC, Texas, USA
Email: venkateshdudala@gmail.com

1. Introduction

Kubernetes is an open-source container orchestration framework designed to manage containers from Docker or another technology. Kubernetes assists in managing applications composed of hundreds or even thousands of containers and helps manage them across various environments, such as physical machines, virtual machines, cloud environments, and hybrid deployment environments.

It offers the following critical features for applications deployed in the K8s cluster

- **High Availability:**The application has no downtime and is always accessible to users.
- **Scalability:** The application demonstrates high performance with quick load times, ensuring users experience very high response rates.
- **Disaster Recovery:** In case of infrastructure issues like losing data or the failure of servers or data centers, the infrastructure must include a reliable mechanism to retrieve and restore data to its most recent state. This ensures that the application doesn't lose any data and allows the containerized application to resume from its latest state after recovery, maintaining continuity and minimizing downtime.

2. Kubernetes Components

- **Pod**: A pod is a fundamental component or smallest unit of Kubernetes. It is an abstraction over a container. It creates a running environment similar to a docker container, with a layer on top of the container for abstracting away the container run-time so that it can be replaced. The best practice is to run one container per pod, although you can run multiple containers in one pod.
- **K8s Virtual Network:**Kubernetes offers an out-of-the-box virtual network, meaning each pod gets its private IP address, and pods can communicate using this IP address. It is important to note that these pods are ephemeral, meaning they can crash and spin up easily for many reasons, including resource limitations. Pods will get a new IP address up on re-creation, which is inconvenient because other pods referring to this IP address need to be adjusted after every pod restart.
- **Service:**A service is a static or permanent IP address that can be assigned to each pod. The service and pod have independent lifecycles, so even if the pod stops running, the service's IP address remains unchanged. Therefore, there's no need to update the IP address in other pods where it's referenced. It provides static IP address and acts as a load balancer, forwarding the request to the least busy pod.
- **Ingress:** Ingress helps users access an application using a user-friendly URL, i.e., with the help of a domain and in a secure manner using HTTPS protocol. It is used to route traffic into the cluster. The request first goes to Ingress, which acts as an entry point and forwards the request to the respective service.
- **ConfigMap:** ConfigMap is the application's external configuration, a database URL, or any other service configuration. It is connected to the pod, which gets all the data from the ConfigMap. If the external configuration changes, one needs to update the ConfigMap; a new build of the entire application is not needed.
- **Secrets:** Secrets are like ConfigMap, but they are used to store sensitive information, such as database credentials, not in plain text format but in a base64-encoded format.
- **Volumes:** A volume is a physical storage attached to the pod, such as a hard drive. This storage can either reside on a local machine, meaning it's hosted on the same server node where the pod operates, or it can be situated on external storage outside of the Kubernetes cluster, such as cloud-based solutions or on-premise storage systems. The data in the volumes is not ephemeral, meaning even if the pod connected to it gets restarted, the data is not lost and persists.
- **Deployment & Statefulsets:** A deployment is a blueprint for a Pod where one specifies the number of replicas. In practice, one does not work with Pods directly but uses deployments to specify replicas and scale them up or down. In other words, deployments are a layer of abstraction on top of the Pods, which makes it convenient to interact with them, replicate them, and do other configurations easily. Because of multiple Pod replicas, even if one of the application pods dies, the service would forward the request to another running pod, and the application would still be accessible to the users. Deployment is for Stateless applications, meaning each request from a client is processed independently and does not rely on data from previous requests.

  While deployment is a blueprint for Stateless applications, Statefulsets manage stateful applications that require stable, persistent identities and storage. They are specifically designed for applications that maintain state across restarts and need predictable network identities or persistent storage like databases. Statefulsets like deployments take care of replicating the pods, scaling up and down, but ensure that the database reads and writes are synchronized to find no inconsistencies.

3. Kubernetes Architecture

Worker Node: In Kubernetes terms, a worker node is a simple server, either a physical or virtual machine. Each node can have multiple pods running on it. Nodes are the cluster servers that do the actual work. Three processes need to run on every worker node for it to successfully run the pods.

- **Container Run-time:**As application Pods have containers running inside, a container run-time needs to be installed on every node. For example, if Docker containers are used, a Docker run-time environment must be installed on the worker node.
- **Kubelet:**Kubelet is a process of Kubernetes itself, unlike the container runtime, which schedules and runs the Pods. It interacts with the time and worker node as it takes the configuration, runs a Pod with a container inside, and assigns resources from node to container, such as CPU, RAM, etc.

- **Kube Proxy:**Kube proxy is also a Kubernetes process responsible for forwarding service requests to Pods and must be installed on all the worker nodes. Kube proxy has an intelligent request forwarding logic that makes the communication between Pods happen in a performant way. For example, if one of the replicas of the pod makes a request to another replica of a different Pod, instead of forwarding the request to any other replica, it forwards it to the replica that is running on the same worker node as the pod that initiated the request. This would avoid the network overhead of sending the request to a different machine.

Master Nodes: Master nodes are responsible for interacting with the Kubernetes cluster and managing actions such as scheduling an application or database pod on a worker node, re-scheduling or restarting a Pod when they crash, joining a new worker node and adjusting Pods and other components on to the new worker node when added to the cluster—4 processes run on every master node that control the cluster state.

- **API Server:** The API Server is a cluster Gateway that gets initial requests of any updates into the cluster or queries from the cluster. Clients such as Kubernetes dashboard and command line tools like Kubectl or Kubernetes API can interact with the API Server and deploy a new application in Kubernetes. API Server also acts as a gatekeeper for authentication, ensuring only authenticated and authorized requests get through to the rest of the processes to make updates/queries into the cluster.
- **Scheduler:** The Scheduler is responsible for assigning a worker node when a request to add a new Pod comes to the cluster via the API Server. Instead of randomly assigning the Kubernetes component to any worker node, it has an intelligent of deciding the worker node by analyzing the resource requirements, such as RAM and CPU of the Pod that need to be added, and then goes through the worker node and computes the available resources in each one of them and picks the least busy one. In summary, the Scheduler chooses the least busy worker node on which the component needs to be scheduled.
- **Controller Manager:** The Controller Manager is responsible for detecting any state changes in a cluster, such as the crashing of pods, and recovering the cluster state as soon as possible. It recovers the state of the cluster by requesting the Scheduler to re-schedule the dead Pods, and the entire cycle of deciding the worker node by Scheduler based on resource requirements and then the actual starting of Pods by Kubelet repeats.
- **etcd:**The etcd is the brain of the cluster, saving the cluster data in a key-value store. All the changes in the cluster, like adding a new pod, deleting a pod, pod crashes, etc., get saved in the etcd's key-value store. This data will be used by other components, such as the Scheduler to calculate the resources available on each worker node, the Controller manager to detect the state changes, and the API Server to handle query requests about the cluster health. Application data is not part of the data stored in etcd.

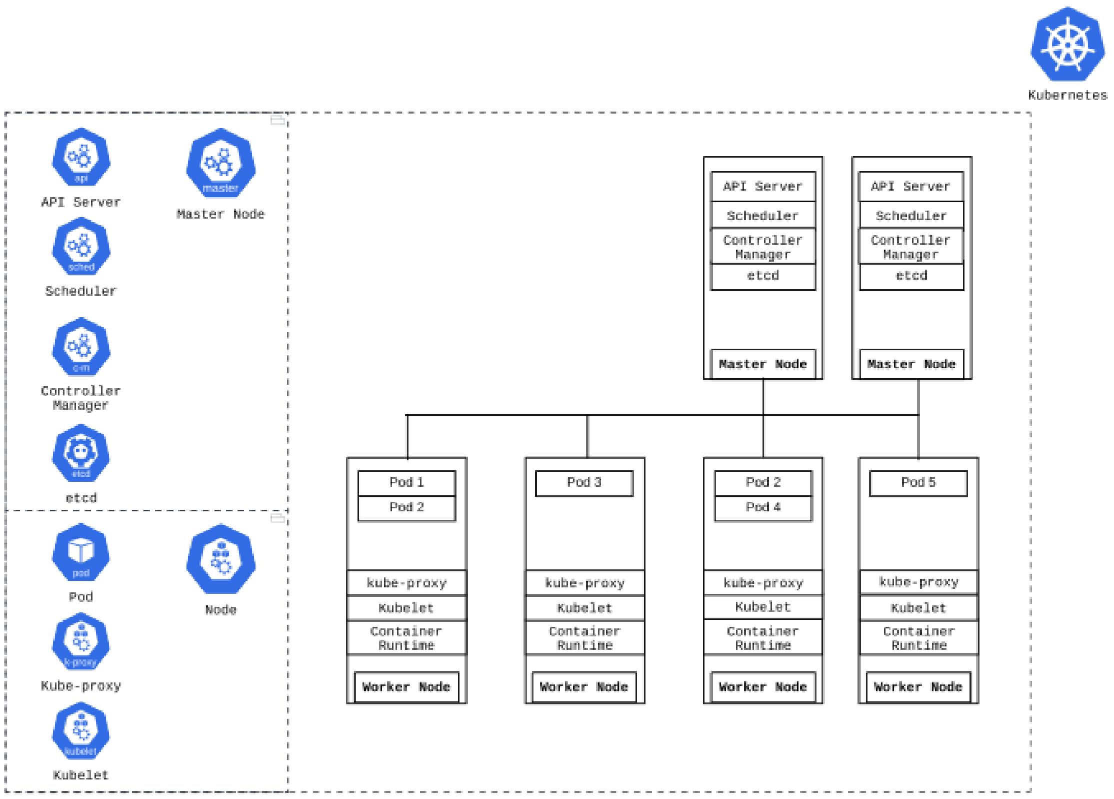

Fig 1

4. Kubectl as a client for interacting with the API Server

Kubectl is the command line tool that serves as the primary interface for interacting with the Kubernetes cluster's API Server and controlling the entire cluster. It allows users with cluster resource management, cluster inspection, managing configuration, namespace management, and cluster automation. It communicates with the Kubernetes API Server using RESTful calls over HTTP/HTTPS. In short, Kubectl is an invaluable tool for users working with Kubernetes, providing the needed control and flexibility to manage cluster operations efficiently.

Basic Commands

Command	**Description**
kubectl cluster-info	Displays cluster info, including API endpoints and other relevant details.
kubectl get nodes	Lists all nodes in the cluster.
kubectl get pods	Lists all pods in the default namespace.
kubectl get services	Lists all services in the default namespace.
kubectl get deployments	Lists all deployments in the default namespace.
kubectl get statefulsets	Lists all statefulsets in the default namespace.

Working with Pods

Command	Description
kubectl run my-pod --image=nginx	Creates a pod named my-pod using the Nginx image.
kubectl delete pod my-pod	Deletes a specified pod.
kubectl logs my-pod	Retrieves logs from a specified pod.
kubectl exec -it my-pod -- /bin/bash	Opens an interactive bash shell inside the specified pod.

Working with Deployments

Command	Description
kubectl create deployment my-deployment --image=nginx	Creates a deployment named my-deployment using the Nginx image.
kubectl scale deployment my-deployment --replicas=3	Scales the deployment to 3 replicas.
kubectl set image deployment/my-deployment nginx=nginx:1.16	Updates the Nginx container in the deployment to use a different version.
kubectl delete deployment my-deployment	Deletes a specified deployment.

Working with Services

Command	Description
kubectl expose deployment my-deployment --type=NodePort --port=80	Exposes the deployment as a service accessible on a node port.
kubectl describe service my-service	Displays detailed information about the specified service.

5. Kubernetes YAML Configuration File

The K8s YAML Configuration file helps define the desired state of the Kubernetes cluster components, such as Pods, Services, Deployments, ConfigMaps, Secrets, Volumes, and more. These YAML configurations are a declarative way of defining and managing the cluster's desired state. The user describes the final state, and Kubernetes ensures it happens instead of the user imperatively specifying the actions for Kubernetes to take.

Example YAML configuration file of a Deployment:

```
apiVersion: apps/v1            # Specifies API Version. apps being the group and v1 being the
version
kind:Deployment# Specifies the type of resource.
metadata:
  name:my-deployment          # Name of the deployment.
namespace:my-namespace
  labels:# For identifying and grouping the resources.
    app:my-application
spec:
  replicas:3# No of pods to run as part of the deployment.
  selector:
    matchLabels:# Pods with this label are matched.
      app:my-application
template:# Defines template of the pod.
    metadata:
      labels:
        app:my-application    # Labels created for pods by this deployment.
    spec:
      containers:
- name:my-container     # Name of the container.
        image: nginx:1.27.1# Docker image.
        ports:
- containerPort:80# Port exposed on container for taking traffic.
```

apiVersion: It specifies the Kubernetes API version used by the template file to create or manage resources. The apiVersion is composed of two components: the group and the version. The version indicates the levels of stability and support. If the version contains alpha, the software may contain bugs, and the feature may be dropped in a future release; if the version contains beta, the feature is tested and enabled by default—the feature will not be dropped, but some details may change.

The first introduced API resources like Pods or Services in Kubernetes do not have groups. So, the template uses *apiVersion: v1*. Later, resources are linked to a group. For example, Jobs and Cronjobs are in the group batch using *apiVersion: batch/v1*. Deployments and replicasets are in the apps group, using *apiVersion: apps/v1*.

kind: It identifies the specific Kubernetes resource being defined, such as a Deployment, Service, Pod, ConfigMap, or any other resource type supported by Kubernetes. It informs Kubernetes how to handle the resource. Different resource types have distinct behaviors, rules, and fields associated with them. Kubernetes relies on kind to determine how to manage and operate the resource.

metadata: It provides the essential information about the resource, such as its name, labels, and annotations. It helps identify and organize resources within the cluster. The following are key fields of metadata:

- **name:** Specifies the resource's name, allowing it to be uniquely identified within its namespace.
- **labels:** Enables categorization and grouping of resources based on key-value pairs. They are widely used for selecting resources when using selectors or applying deployments.
- **annotations:** Provides additional information or metadata about the resource. They are typically used for documentation purposes, tooling integrations, or adding custom metadata.

spec: This contains detailed information that describes the desired state of the resource. The structure and content of the spec field vary depending on the resource kind. Here are a few examples:

- **Pod:** It defines the container specifications, such as the image, ports, environment variables, and volumes.
- **Service:** It represents the networking rules for the service, including the exposed ports, service type (e.g., ClusterIP, NodePort, LoadBalancer), and target ports.
- **Deployment:** It contains details such as the number of replicas, container specifications (e.g., image, ports, environment variables), and volume mounts.

status: This section is automatically generated and added by Kubernetes. Kubernetes constantly compares the desired state to the actual state, and if they do not match, it prompts Kubernetes to bring the current state to the desired state. This process of self-correcting the state based on the status is the basis for Kubernetes's self-healing capability.

6. Demo Application Setup with Kubernetes

In this demo setup, two applications, mongo-express and mongoDB, will be deployed using Kubernetes Components on a minikube. A minikube is a one-node cluster where the master and worker processes run on one machine. This node will have a docker container run-time pre-installed. In short, a minikube is a one-node Kubernetes cluster that runs on a virtual box. This is used to test Kubernetes in a local setup. A command-line tool, kubectl, will interact with the minikube Kubernetes cluster.

Request flow from the browser through the K8s components.

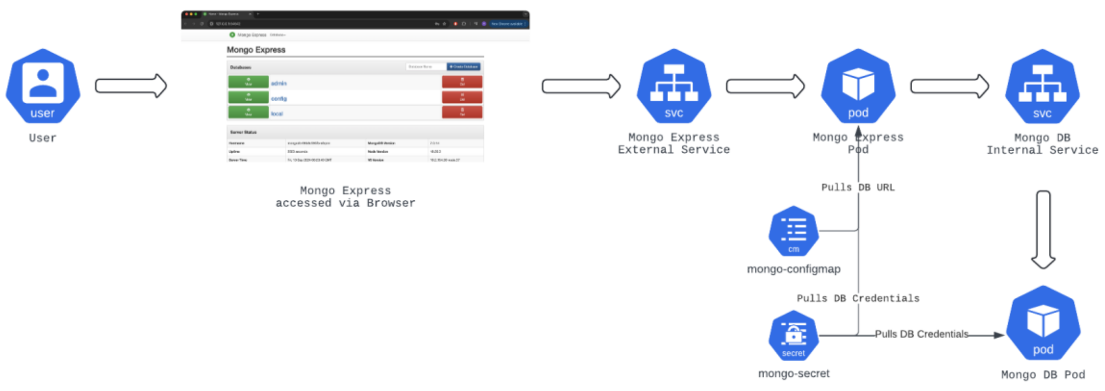

Fig 2

The following steps are to be taken to set up this application

- A Secret is created to hold the values for MongoDB credentials. These secret values are referenced in the MongoDB Pod. Hence, the secrets need to be created prior to MongoDB Pods creation. Here, the secret file is***mongo-secret.yaml****, and* its contents are as follows:

  Execute the command ***kubectl apply -f mongo-secret.yaml***to create the secret.

```
apiVersion: v1
kind:Secret# Specifies the type of resource.
metadata:
  name:my-mongodb-secret            # Name given to the secret.
type:Opaque# Default secret type used to store key-value pairs.
data:# Actual contents of key-value pairs.
  mongo-root-username:YWRtaW4=
  mongo-root-password: cGFzcw==
```

- A MongoDB pod is created; we need a service component to route traffic to the pod. An Internal Service will be created, which means no external requests are allowed to the pod, and only components within the same cluster can communicate with the MongoDB Pod. The specification for a Pod and its service can coexist in a single yaml file. The contents of this file, ***mongo.yaml*** are shown below.

  Execute the command ***kubectl apply -f mongo.yaml*** to create the MongoDB Pod and the mongo-service linked to the pod.

```
apiVersion: apps/v1                           # Specifies API Version.
kind:Deployment# Specifies the type of resource as Deployment
metadata:
  name:my-mongodb
  labels:
    app:my-mongodb
spec:
  replicas:1# Specifies no of pods to be created with Deployment.
  selector:
    matchLabels:
      app:my-mongodb
template:
    metadata:
      labels:
        app:my-mongodb
    spec:
      containers:
- name:my-mongodb
        image: mongo                          # Specifies the container image.
        ports:
- containerPort:27017# Port on which a container listens for network traffic.
        env:
- name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:# Pulls environment values dynamically and avoids hardcoding.
            secretKeyRef:
              name:my-mongodb-secret         # Name of the secret file where the
values are being pulled.
              key: mongo-root-username        # Key in which username is stored in
the secret file.
- name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name:my-mongodb-secret
              key: mongo-root-password        # Key in which password is stored in
the secret file.

---
apiVersion: v1
kind:Service# Specifies the type of resource as Service.
metadata:
  name:my-mongodb-service
spec:
  selector:
    app:my-mongodb                           # Connects to the pod with name my-
mongodb.
```

```
  ports:
- protocol: TCP
      port:27017# Exposes service port.
      targetPort:27017# Should match containerPort of Deployment.
```

- The MongoExpress Pod references a MongoDB URL to connect to the Mongo database. It can be externalized as an environment variable in a ConfigMap. Here, the ConfigMap file is ***mongo-configmap.yaml***and its contents are as follows:

  Execute the command ***kubectl apply -f mongo-config.yaml***to create the secret.

```
apiVersion: v1
kind:ConfigMap# Specifies the type of resource as ConfigMap
metadata:
  name:my-mongodb-configmap
data:
  database_url:my-mongodb-service      # Refers to mongo service to route traffic to 
MongoDB.
```

- A MongoExpress deployment is created. A MongoDB URL is needed so Mongo Express can connect to the database. Along with the URL, a set of MongoDB credentials, i.e., username and password, is needed to authenticate to the database. These values are provided via ConfigMap and Secret. An external service is created so that MongoExpress is accessible via the browser. This external service will allow external requests to communicate with the pod.

  Execute the command ***kubectl apply -f mongo-express.yaml*** to create the MongoExpress Pod and the mongo-express-service linked to the pod.

```
apiVersion: apps/v1                                   # Specifies API Version.
kind:Deployment# Specifies the type of resource as Deployment
metadata:
  name:my-mongo-express
  labels:
    app:my-mongo-express
spec:
  replicas:1# Specifies no of pods to be created with Deployment.
  selector:
    matchLabels:
      app:my-mongo-express
template:
    metadata:
      labels:
        app:my-mongo-express
    spec:
      containers:
- name:my-mongo-express
        image: mongo-express                           # Specifies the container 
image.
        ports:
- containerPort:8081
        env:
- name: ME_CONFIG_MONGODB_ADMINUSERNAME
          valueFrom:# Pulls environment values dynamically and avoids hardcoding.
            secretKeyRef:
              name:my-mongodb-secret                  # Name of the secret file 
where the values are being pulled.
              key: mongo-root-username                 # Key in which username is 
stored in the secret file.
- name: ME_CONFIG_MONGODB_ADMINPASSWORD
          valueFrom:
            secretKeyRef:
              name:my-mongodb-secret
              key: mongo-root-password                 # Key in which password is 
stored in the secret file.
- name: ME_CONFIG_MONGODB_SERVER
          valueFrom:
            configMapKeyRef:
              name:my-mongodb-configmap               # Name of the configmap file
where the db url is being pulled.
              key: database_url
---
apiVersion: v1
kind:Service
metadata:
  name:my-mongo-express-service
spec:
  selector:
    app:my-mongo-express
  type:LoadBalancer
  ports:
- protocol: TCP
      port:8081
      targetPort:8081
      nodePort:30000
```

- The my-mongo-express-service created after the deployment is of type LoadBalancer. By default, it is assigned an internal IP address. However, to allow external requests to reach the my-mongo-express pod within the Kubernetes cluster, an external IP address must be enabled for the LoadBalancer. The following command can be used to assign an external IP to the LoadBalancer.

  minikube service my-mongo-express-service

7. Conclusion

In conclusion, we have seen Kubernetes's components, architecture, and features and deployed a demo application in the Kubernetes cluster. Its robust architecture simplifies the complexities of deploying and scaling applications across various environments. Its self-healing capabilities were also discussed, i.e., automatically detecting and recovering from failures powered by continuous status checks and etcd data store. While this paper lays the foundation for understanding Kubernetes fundamentals, the platform offers more capabilities like enhanced security, persistent storage, network policies, etc. These can be further explored on a case-by-case basis.

8. References

- Kubernetes Official Website. (2024, October 13). *Kubernetes basics*. Retrieved from https://kubernetes.io/docs/tutorials/kubernetes-basics/
- The Cloud Native Computing Foundation. (2024, October 13). *Kubernetes*. Retrieved from https://www.cncf.io/projects/kubernetes/
- Red Hat. (2024, October 13). *Kubernetes architecture*. Retrieved from https://www.redhat.com/en/topics/containers/what-is-kubernetes
- Google Cloud. (2024, October 13). *Kubernetes Engine documentation*. Retrieved from https://cloud.google.com/kubernetes-engine/docs