WELLS FARGO

The Journey:
Making My Application Cloud Ready

JUNE 2021

AUTHORS

Vikas Chandra is a graduate from IIT Kanpur and he has over 15 years of technology experience. In the last 16 years, he has worked in start-ups and MNCs like Oracle, Goldman Sachs, and Wellsfargo. He is passionate about technology and he drives technical initiatives at WellFargo.

Ishita Agarwal, an Electrical and Electronics Engineering graduate from VTU, holds around 11 years of experience in the field of Software Development. She has worked with the likes of Apple(client for Infosys Ltd), FIS Global, and Wells Fargo. She is a technology enthusiast and is actively involved in technical events at Wells Fargo.

INTRODUCTION

The last two decades have seen unfathomable growth in the size and complexity of software systems. And this is unlikely to slow down in the coming few decades - it's absolutely unimaginable to picture the future systems. But the one thing that can be guaranteed is that more and more software systems will need to be built with constant growth — more requests, more data, more analysis — as a primary design driver. With this growing complexity, designing systems that are highly scalable and portable is a challenge.

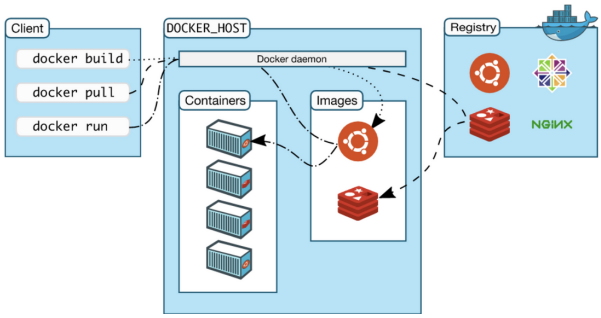
Applying good design principles and patterns and adopting standard operational practices can help in the early identification of the most common problems that applications face in highly distributed env. While software design patterns and development methodologies can produce applications with the right scaling characteristics, the infrastructure and environment influence the deployed system's operation. Technologies like Docker and Kubernetes help teams package software and then distribute, deploy, and scale on platforms of distributed computers. One of the most important factors with Kubernetes is the ability to scale the application **horizontally**, building identical copies to distribute the load and increase their availability.

BUILDING BLOCKS

Containerization

Looking at the Google way, from Gmail to Youtube to Search, everything is running in containers. Containerization helps the developers with faster and efficient development and deployment, building easily portable applications. Kubernetes "orchestrates" these containers and brings the release time from a few months down to a few mins. Container runtimes like Docker help encapsulate applications that can be run inside these containers.

Although containerization may be an important requirement of Kubernetes, in application it puts in practice many of the 12-factor app methodologies - they run in isolated env, support networked, service-oriented approach, maintain process-based concurrency and dev/prod parity.



As per Docker document , "Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

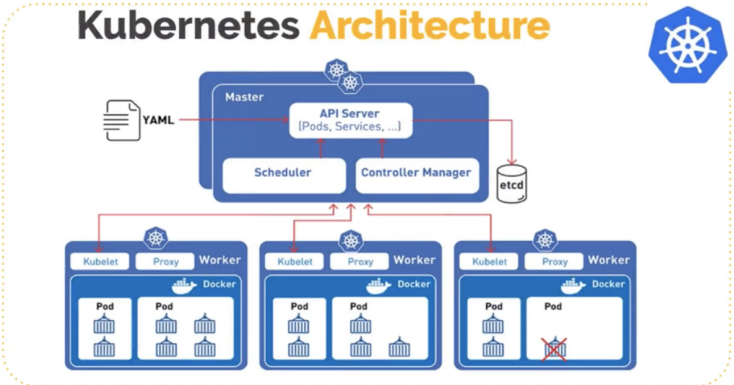
With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production."

Kubernetes

It's an open-source container orchestration platform and it needs container runtime in order to orchestrate. It's commonly used with Docker but it can work with other container runtimes too.

Why is it so popular?

Assume there is no Kubernetes and we are living only with a container platform. A lot of containers are running across different machines and we get super excited.
But wait, suddenly some container has crashed for a service and we need a few more containers for another service because the load has increased. How do we handle that? Frustrating right?



Here comes Kubernetes.

Orchestration and Load Balancing

- It ensures multiple containers are running together seamlessly.
- It can handle demand and supply, so if less load then kill the containers, and if more load then bring more containers. It can do Load balancing between containers.

Software Deployment and Updates

It makes our life so easy with software deployment and updates. It can:

- Automate deployments and rollbacks
- Orchestrate updates
- Configuration across all the PODs

YAML driven

You don't have to be a developer to understand Kubernetes. All the resources like POD, Services, Deployment, Volumes can be done using YAML.

Kubernetes in Public Cloud

Originally it was developed in Google but now it's used heavily across all public clouds: GKE(Google Kubernetes Engine), Amazon EKS (Amazon Elastic Kubernetes Service) & AKS (Azure Kubernetes Service)

Kubernetes On-Premises

All organizations can't go on the public cloud due to security constraints, data privacy and compliance. But those companies do take advantage of Kubernetes with their existing infrastructure. They are writing applications for cloud-native.

Our Technical OVERVIEW

56 percent of enterprises say they'll increase use of containerized applications during the next 12 months, according to the 2020 Red Hat Enterprise Open Source Report. Gartner predicts that by 2023, more than 70 percent of global organizations will be running more than two containerized applications in production, up from less than 20 percent in 2019.

KECD - Kubernetes Enterprise Cluster Design

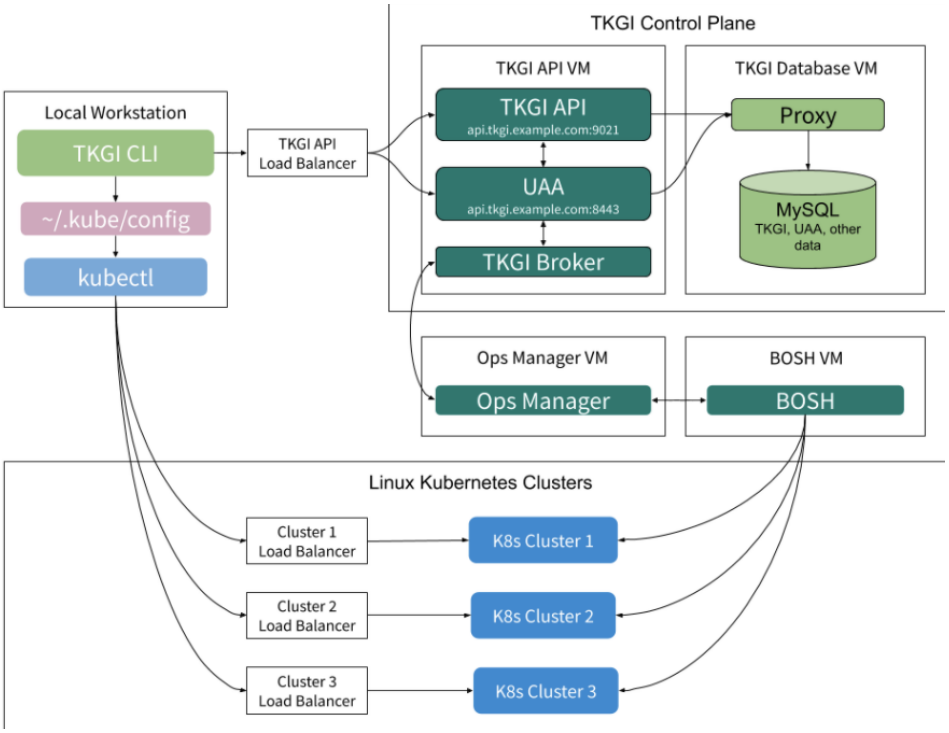
Image Pull

All the images are present in a centralized image repository called docker hub. The images can contain multiple versions with the latest version being defaulted. In order to use this image, we need to provide the address of this image

docker pull ubuntu

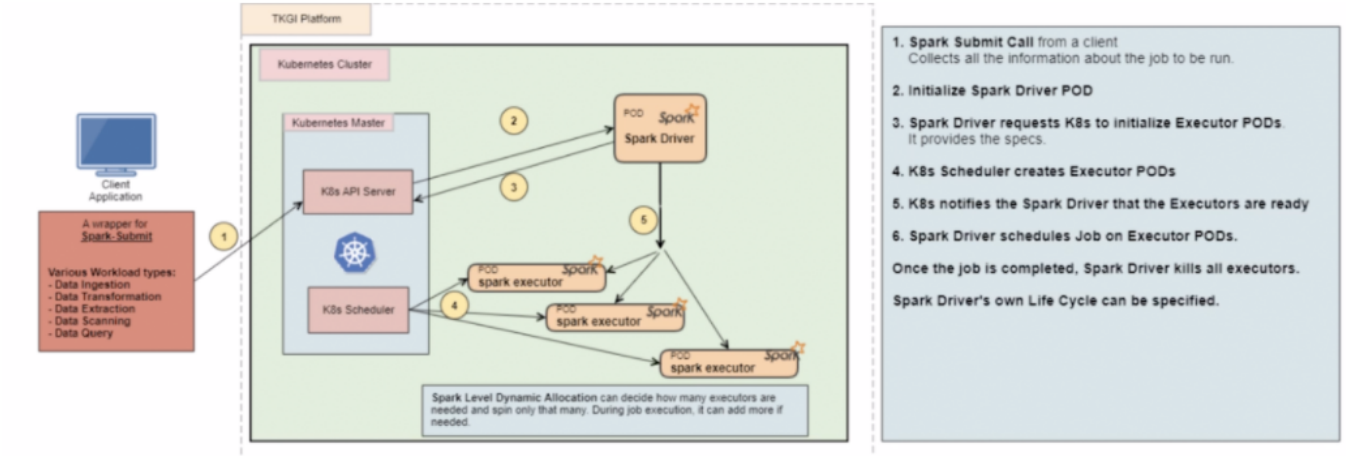
TKGI CLuster

We know we need to containerize applications and we know we need to orchestrate them. What we need to hold this is a cluster. There are many market players providing Kubernetes clusters. Google, Microsoft, VMWare, IBM to name a few. For our solution, we will be going with VMWare's TKGI (Tanzu Kubernetes Grid Integrated Edition). The typical TKGI architecture looks like below.



Namespace

Multiple teams can use the same cluster for a given environment. In order to provide separate logical scope, we create namespaces. Names of the resources have to be unique within a namespace, although different namespaces can use the same-named resources. The default namespace is default but as a good practice, we should avoid using default in productionized environments.



Spark as a service

Spark is known for its high compute capabilities, processing large amounts of data, and parallelizing work. These capabilities are managed by a cluster manager and in our case, its Kubernetes. Dynamic spark cluster is the first offering of KEC. There are many benefits of this:

- This makes Spark more reliable and efficient.
- Efficient resource sharing in-turn saving costs:
  - Dependency isolation
  - Performance isolation

A dynamic spark cluster comprises many resources. Deployments, services, virtual services, driver and executor pods, storages like PV/PVC (Persistent Volume Claim), Amazon S3 Object store, MinIO.
All these can be configured with the help yaml files and supported APIs provided by Kubernetes.

Storage

This has been the biggest challenge for the practitioners in the IT industry. The distributions, replication, migrations, and integrity being the primary lookout areas. Kubernetes handles all aspects of a container lifecycle starting from creation, management, automation, load balancing as well as interfaces to storage devices.

- **PV/PVC -** It introduces two API resources Persistent Volume (PV) and Persistent Volume Claim(PVC). PV and PVC separate storage implementations from functionality and allow pods to use storage in a portable way. It also separates users and applications from storage configuration requirements. While PV is a piece of storage in the cluster that is provisioned either dynamically or by an administrator, PVC is a request for storage by a user which consumes PV resources. Claims can request specific size and access modes like ReadWriteOnce, ReadOnlyMany, ReadWriteMany

- **Amazon S3 Object Store -** As businesses grow, managing the ever-expanding isolated pools of data from many fragmented sources is becoming challenging. Object stores help break down these silos by providing massively scalable and cost-effective storage. There is no denying the fact that there are many advantages besides scalability.

    1. Durability, Availability, & Scalability - Amazon S3 was built from the ground up to deliver 99.999999999% durability. Data is automatically distributed across a minimum of three physical facilities that are geographically separated by at least 10 kilometers within an AWS Region, and you can automatically replicate data to another AWS Region. According to Gartner, Inc., “Amazon S3 is the largest public-cloud-based object storage service as measured by data under management. AWS has more insights than any other vendor on how customers use public cloud storage services at scale.”

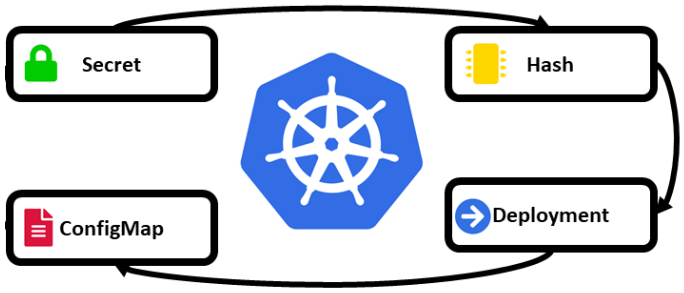
    2. Security & Compliance - Amazon S3 & Amazon Glacier are the only cloud storage services that support three different forms of encryption. They support more security standards and compliance certifications.

3. Flexible Management - AWS offers the most flexible set of storage management and administration capabilities. Storage administrators can classify, report and visualize data usage trends to reduce costs and improve service levels.

4. Query-in-Place - Amazon S3 is the only cloud storage platform that lets customers run sophisticated analytics on their data without requiring them to extract and move the data to a separate analytics database.

Secret/Configmap

We spoke about the plethora of resources the Kubernetes offer. Apart from these resources the ecosystem also includes integration with cloud-agnostic applications and storages like S3. While application configuration can be baked into container images, its best to make components configurable and provide configurations dynamically. To manage these, Kubernetes provides 2 types of objects:



- **ConfigMap** is an API object used to store non-confidential data in simple key-value pairs. This is consumed by pods as environment variables or as config files on an attached volume.
- **Secret** lets you manage and store sensitive information like passwords, authentication tokens, ssh keys, certificates, etc. By default, these are stored as unencrypted base-64 encoded strings. Based on the kind of data it holds, there are many types of secrets. Anyone with Secret API access or access to the underlying datastore will be able to retrieve the secret. So there are few recommendations on how we can use secrets more safely.

> Enable encryption at rest - The data is stored in plaintext in etcd. To encrypt that data, we can pass a config with the help of an argument --encryption-provider-config.

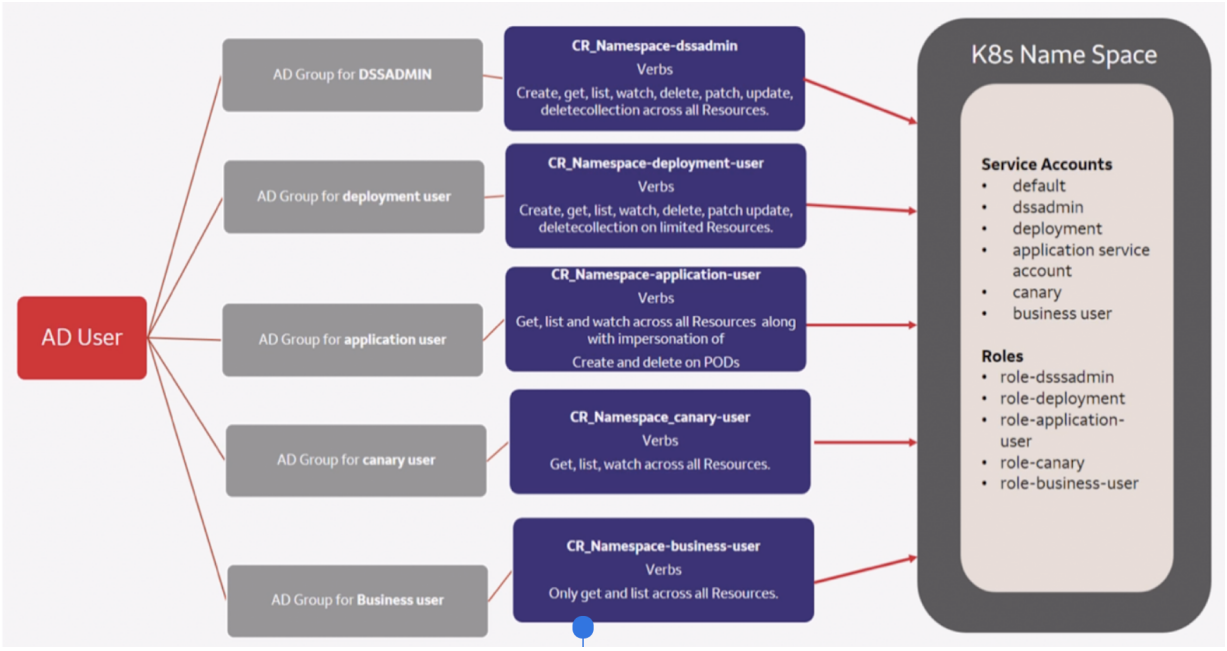
> HashiCorp's Vault - It centrally manages and enforces access to secrets and systems based on trusted sources of application and user identity.

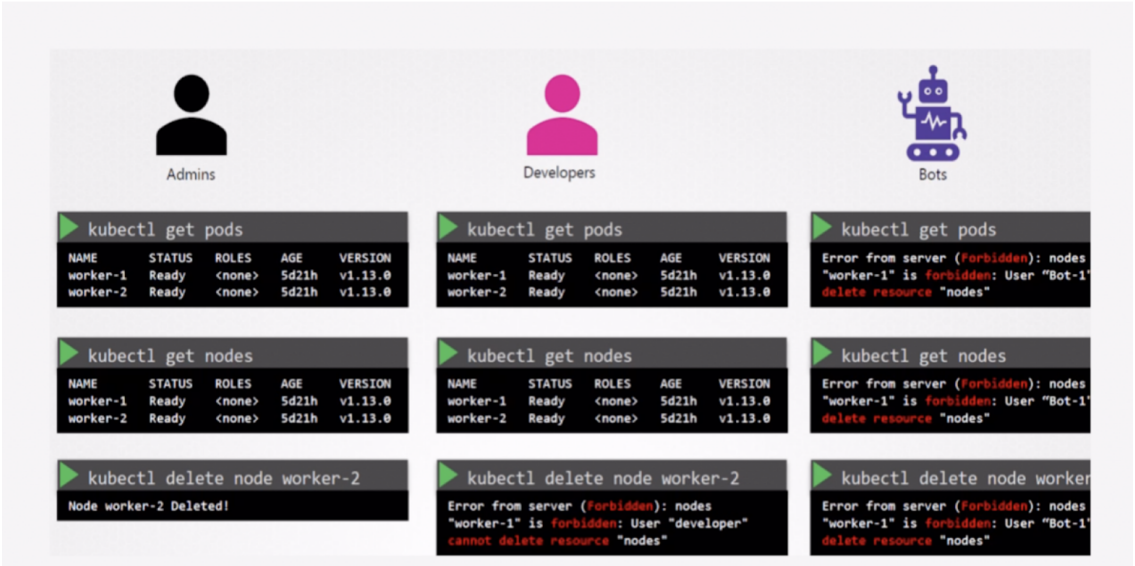
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  # You can include additional key value pairs as you do with Opaque Secret
  extra: YmFyCg==
```

Security

As it's said, With great power comes great responsibilities". In order to run a seamless operation and manage multiple dynamic spark clusters in the same KEC used by different teams, security becomes all the more an important aspect of this design.

RBAC - The access and management of these resources can be controlled based on the role of the user. This is called RBAC (Role-Based Access Control). RBAC authorization uses *rbac.authorization.k8s.io* API group. The requests are authorized via API server. Request attributes are evaluated against the policies.





The RBAC API provides four kinds of Kubernetes object:

- **Role** - Represents set of permissions. This sets the permissions with a particular namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- **ClusterRole** - Also represents a set of permissions. But unlike role, this is a non-namespaced resource. A Kubernetes object can be either namespaced or not, not both.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

- **RoleBinding** - A role binding grants permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, service accounts) and a reference to the role being granted.

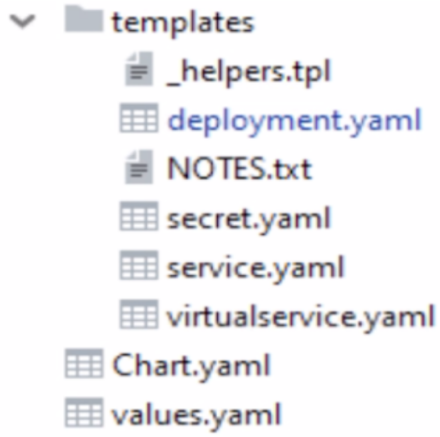
- **ClusterRoleBinding** - Similar to role binding, but while RoleBinding grants permission within a namespace, this one grants permission across the cluster.

HELM Deployment

As the Helm defines itself - “Helm is the best way to find , share and use content across Kubernetes.”
It helps deploy complex applications by bundling necessary resources into Charts, which contains all information to run applications on a cluster. A basic helm chart can be generated using :

helm create <chart-name>

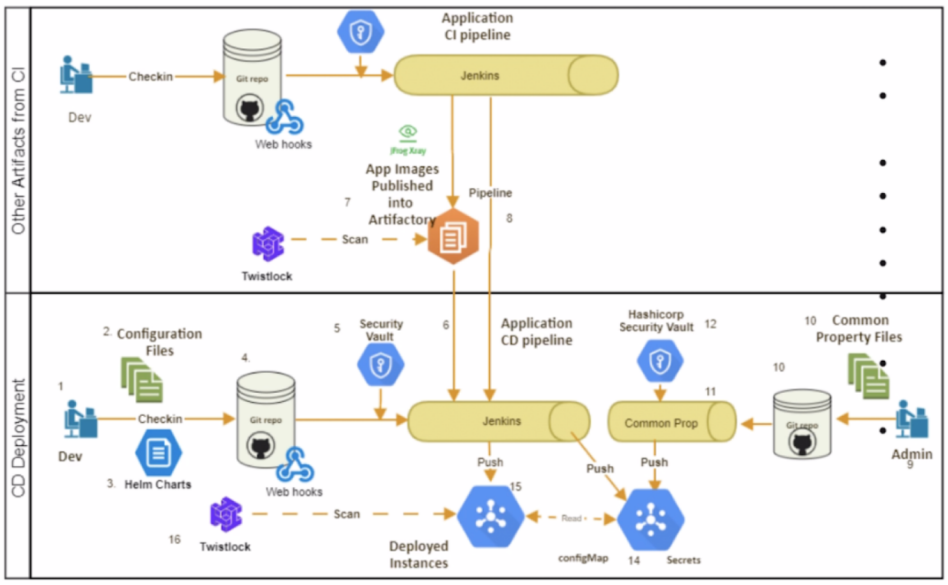
The basic package generated contains templates for deployment, service, virtual service, ingress, helper files and a value.yaml file. A values.yaml file is instrumental in making the deployment process dynamic in nature. Starting from the image that needs to be deployed, to a number of replicas, memory, and CPU allocation, this yaml file governs how the complete deployment needs to be orchestrated.



CI/CD

But in enterprise applications, we generally require to build our own images by writing customized docker files and publish that to enterprise artifactory. The Enterprise Pipeline plays an instrumental role in bringing the components together. The application resides in github, built using Jenkins, image build is done via kaniko and published to Wells Fargo's image repository : wfcertifiedvirtual.wfcr.wellsfargo.net

Build, Test, Publish, Deployment and Delivery Pipelines.



CI (Continuous Integration):
- Build, Test, Scan and Publish Pipeline

CD (Continuous Deployment):
- Helm Charts
- Health & Status / Resources
- Internal Services
- Security (Service Accounts and Role Mapping)
- Configuration and Secrets, Scan
- Delivery Pipeline

Reference

https://www.digitalocean.com/community/tutorials/architecting-applications-for-kubernetes
https://cloud.google.com/containers
https://docs.pivotal.io/tkgi/1-8/control-plane.html
https://cloud.netapp.com/blog/cvo-blg-kubernetes-storage-an-in-depth-look#H_H1_4
https://aws.amazon.com/what-is-cloud-object-storage/
https://kubernetes.io/docs/reference/access-authn-authz/rbac/
https://helm.sh/