Imperative Vs Declarative.

```
>kubectl run --image=nginx nginx
```

Kubectl run Command to Create a pod

To Create a deployment

```
kubectl create deployment --image=nginx nginx
```

To Create a Service.

```
kubectl expose deployment nginx --port 80
```

"kubectl expose" is the imperative Command used to create "service"

To edit existing object

```
kubectl edit deployment nginx
```

update / edit

^ This edit command will make changes directly in the live object without updating the "object configuration file's definition"

update object

To scale a deployment or replicaset

```
kubectl scale deployment nginx --replicas=5
```

To update image on deployment

```
kubectl set image deployment nginx nginx=nginx:1.18
```

To create object

```
kubectl create -f nginx.yaml
```

object definition file

To edit an object If you use "replace" Command, then the object in the will be deleted & recreated

```
kubectl replace -f nginx.yaml
```

To delete an object

```
kubectl delete -f nginx-yaml
```

Disadvantages of imperative command

1) They are limited in functionality & require long & complex commands for advanced use cases. Eg: such as creating a multi-container pod (or) deployment

2) These commands are run once and forgotten, they're only available in session history of the user who ran the commands, so the other person may not aware of how these resources were created, so it's hard to keep track

If you've created an object using declarative way [YAML file] & you want to make changes, for instance editing an image name to another version", there are different ways to go for it.

One way is to use "kubectl edit" command & specify the object name.

```
kubectl edit deployment nginx
```

when the above command is run, it opens a "YAML definition file" and make changes & there which will get reflected in the "object" in cluster this yaml definition file is not the file developed by the developer & stored in "source code repo" but whereas it is stored in "kes" memory; whereas it is the "live-object-configuration-file"

Directly making changes in live object without getting updating an object configuration file application

difference

The difference b/w the "yaml file developed by IDE, which is stored in Github/source code repo" & the "yaml file for the live object in k8s' memory" is, it will have "status"

nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypp-pod
  labels:
    app: mypp
    type: front-end
spec:
  containers:
    - image: nginx-container
      image: nginx
```

"nginx.yaml" file is created by IDE stored in source code repo

Pod-definition

```
apiVersion: v1
kind: Pod
metadata:
  name: mypapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
status:
  conditions:
    - lastProbeTime: null
      status: "True"
      type: Initialized
```

Stored in k8s memory

YAML file created by k8s stored in k8s memory for the live objects in cluster

NOTE:-

we can make changes to this "pod definition" file [created by k8s & stored in k8s memory] then save & exit & which will make changes to the object in k8s cluster.

So if you make any changes using kubectl edit command, it is not really recorded anywhere, after the change is applied, we're only left with the YAML definition file developed by SDE [local file]

The local file will have the old image name & if one of our team mate decide to make a change to this object unaware that a change was made using the kubectl edit command, when the new change is applied, the previous change to the image is lost.

We can use the "kubectl edit" command for the objects which we wont rely on object configuration file in future, but the best practice is to first edit the local version of the object configuration file, with the required changes ie, by updating the image name and then run the

"kubectl replace" command to update the object

```
kubectl replace -f nginx.yaml
```

This way, going forward, the changes made are recorded & tracked as a part of change review process

So at times, we may have to completely delete & re-create object, for which we've to use.

```
kubectl replace --force -f nginx.yaml
```

If you're going to update the object, first make sure the object is already there

If the object doesn't exist, the replace command fails with an error message

We can create multiple objects at once by specifying the path, Eg:-

`kubectl apply -f /path/to/config-files`

So if we have to make changes to an object, then we simply update the object configuration file [file developed by us] & run the "kubectl apply" command once again

POINTS TO REMEMBER :-

To make any changes in the objects, simply make changes/update our local directory & then apply "kubectl apply" command

* (add a property of existing object & update an image)

Configuration Tips

We've to know about 2 options before working with Commands

1) --dry-run=client (validates & simulates the command locally without sending it to API server)

This will not create the resource, instead it will tell whether the ~~[illegible]~~ the resource can be created & your command is correct

2) -o yaml (Outputs the object in YAML format)

This will output the resource definition in YAML format on screen.

To create the object later, you can save the output to a file & apply it

> kubectl run nginx --image=nginx --dry-run=client -o yaml > nginx-deployment.yaml

> kubectl apply -f nginx-deployment.yaml

Service

⇒ Create a Service named "redis-service" of type "ClusterIP" to expose pod "redis" on port 6379

This will automatically use the pod's labels as selectors

kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml

(object) (pod port) (service name)

(object name)

[OR]

kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml

This will not use the pods labels as selectors, instead it will assume selectors as app=redis, so it doesnot work very well if your pod has different label set. So generate the file and modify the selectors before creating the service.

kubectl replace -f file.yaml

Purpose:-
Replace the live object with whats in the file, only if it already exists

Behaviour:
Does a clean overwrite of the object's configuration
Fails if object doesn't exists
will error out if you try to change immutable fields like spec.selector, clusterIP etc,
Replace the existing deployment - but only if it exists and the changes are valid.

kubectl replace --force -f file.yaml

Purpose:- forces a delete and recreate of the object
Behaviour:- First deletes the existing resources
Then creates new one from your YAML
this bypasses immutable field errors like clusterIP, selectors etc,
also resets metadata like UID, status, creationtimestamp etc,
Deletes old service
Create new one - which may change its clusterIP unless you explicitly