**Pods:**

A Pod can contain one or more containers and it provides a way to define environment variables. As a matter of fact, your application code goes in a container. Each pod encapsulates containerized applications, storage, a unique network IP, and instructions on how to run those containers using Linux namespaces, cgroups, and other kernel features. Which means that all the containers in a pod will have the same IP address and that they can talk to each other by connecting to each other’s ports on localhost. Pods cannot be updated once they are deployed, they can only be deleted or replaced. As you may have guessed, once that happens, the pod will get a different IP address (cluster IP) every time the pods come up and if you configured a pod to interact with another pod via their IP (cluster IP), you are going to get a lot of errors.

**Services:**

Services get connected to pods using selector labels, instead of pointing to pods directly, you point to the service. The service knows which pod to send your request to via the labels. A set of pods created via a replica set can be load balanced using a service within the cluster network. Service lets you bind an *external* load balancer to your services and expose pod endpoints to the outside world. This makes scaling and even rolling out new versions of pods (your application) super easy, as the service will keep listening for new pods using the labels you configured and will remove any deleted pods from its list automatically.

**Labels:**

Labels are just a key value pair, or tag, that provides metadata on our objects. A Kubernetes Service can select the pods it is supposed to abstract through a label selector. When we create Kubernetes constructs we have the option to set a label, which acts as a tag for that construct, it is the same concept as tagging resources in AWS. This means you can access a group of objects by specifying the label assigned to those objects. For example, labels distinguish containers as being part of a web or database tier. The selectorfield tells Kubernetes which labels to use in finding pods to forward traffic to.

**Deployments:**

Deployment is how you deploy pods to your cluster. Deployment is a higher-level abstraction for enabling declarative updates for deploying replicaSets and their associated pods. Well, there are ways to get a pod running without a deployment, but without deployment you can’t easily specify the number of replicas you want, automatically redeploy pods when they fail, roll back to earlier states, etc. The deployment is what makes pod lifecycle management easier. Basically, every time the application code changes, a new version of your application container image will be built, and your continuous automation logic will update deployment manifest with the newer version pod and tell K8s to apply the changes. K8s will then handle the rolling-out of this newer version pod, it will start terminating older version pod, as it spins up the newer version pod with the updated container.

**Namespaces:** allow you to create virtual clusters on top of a physical cluster. Namespaces are intended for use in environments with many users spread across multiple teams or projects. They assign resource quotas and logically isolate cluster resources.

**Master:**

The Master is the control panel of the whole cluster. All commands will be run on the master instance. It will then decide which node, in the cluster will take the workload. Usually, the master instance doesn’t run any pod on its own server**.** It’s important to note that the only piece that we can directly interact with here is the **api-server**. And we do so using the **kubectl**, a CLI tool to send commands to the K8s cluster. This CLI tool will talk to the api-server and things will kick in from there. It’s a REST API that uses JSON format. We declare the desired state of our app (services that will run, and other configuration) in *manifest* files, and we send these *manifest* files to the K8s cluster using this REST API on the master server. The master will validate the *manifest* and, if everything is fine with it, it will save it in another piece of the master node, called ***etcd*.**

**etcd**:

**etcd** is the cluster state database. The desired state of the cluster will be stored here. In other words, **etcd** will be the single source of truth of our cluster. **api-server** is the only one that talks to **ectd** using **etc** clients, request that submit to **api-server** will be encoded and transformed before committing to etcd.

**Node:**

A node is a worker machine in Kubernetes. It may be a VM or physical machine. Each node in a Kubernetes cluster has the required services to be managed by the master. Nodes also have the required services to run pods, including the container runtime, a kubelet, and a service proxy. Kubelet is a K8s agent (daemon) that runs on each node. It registers the node in the K8s cluster and watches for commands from the *master*, such as pods instantiation. If for whatever reason the node fails to run the master commands, it reports back to the master saying the node can’t run the assignment and the master decides what to do next.

**Volumes:**

Containers are ephemeral in nature, However, there are some services that do require some kind of state to be persisted like MySQL container, for example. Because when a container crashes for any reason in the world, Kubernetes will restart the container in a clean state where all the data is lost. Now, how do we provide some “external” storage to this Pod? This is where [**Persistent Volumes**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/), more commonly known as PVs, come in. Possibly the most interesting thing about PVs is that they exist outside of the lifecycle of the Pod. A Pod that uses a PV can come and go, but the PV can remain, and therefore so can your data remain.

* **CORE CONCEPTS**
  1. **Kubernetes -** is an open-source system for automating deployment, scaling, and management of containerized applications.
  2. **Pods –** Smallest deployable unit of computing that you can create and manage in Kubernetes cluster
  3. **Replica Set –** Maintains a stable set of replica pods running at any given time.
  4. **Deployments –** Provides declarative updates for Pods and Replica sets
  5. **Namespaces –** Isolates group of resources within a single cluster
  6. **Imperative Commands –** Manages live objects. Used for creating, updating and deleting K8s objects
* **CONFIGURATION**
  1. **Commands and arguments –** Defined in configuration file overwrites the commands and arguments provided in container image. Command fields may correspond to entry point for container runtimes.
  2. **Config Maps –** Stores non-confidential data in key-value pairs. Pods can consume Config Maps as environment variables, command line arguments, configuration file in a volume.
  3. **Secret –** Stores confidential data such as password, token or key.
  4. **Security Contexts –** Defines privilege and access control settings for a pod or container.
  5. **Resource Requests and Limits -**  are defined in POD specification and illustrates how much resources such as CPU and memory a container needs. Kubernetes scheduler uses them while allocating POD to a particular node. Kubelet uses these values to ensure that a POD does not use the resources more than the limit specified.
  6. **Service Account –** Used for authentication. It provides an identity for processes that run in a POD and maps it to Service Account object.
  7. **Taints and Tolerations –** Taints allow nodes to repel a set of Pods. Tolerations allow scheduler to schedule Pods with a matching taint.
  8. **Node Affinity –** Is a property of a Pod that attracts it to a set of nodes.
* **MULTI CONTAINER POD-** 
  1. **Init Container-** Runs before app container within a Pod. Can contain utilities or setup scripts that are not present in application container.
  2. **Readiness Probes –** Kubelet uses readiness probes to know when Pod is ready to start accepting the traffic.
* **POD DESIGN –** 
  1. **Labels and selectors –** Label is a key-value identifying attribute of an object. Selector is the core grouping primitive in Kubernetes, which can be either equity-based or set-based.
  2. **Rolling updated and rollback-** Rolling update results to zero downtime deployment, as it updates the pod instances with new ones using incremental approach. Rollback updates the revision of the deployment.
  3. **Jobs and Cron Jobs –** These are automated jobs that are useful for creating periodic and recurring tasks, like running backup and sending mails.
* **SERVICES AND NETWORKING –** 
  1. **Kubernetes Services –** Abstract access to Kubernetes Pods
  2. **Network Policies –** specify how a pod can allow to communicate with various network entities (endpoint/ services)
  3. **Ingress Networking –** An API object that manages external access to the services in a cluster
* **STATE PERSISTENCE –** 
  1. **Persistent Volume –** is a storage cluster resource. Unlike volume, it has its lifecycle independent of Pod that uses it
* **Persistent Volume Claim –** is a request for storage by a user
* Implementing a canary deployment: You need to roll out a new version of your application to a small subset of your users, gradually increasing the number of users who receive the new version, while monitoring metrics and logs to ensure that the new version is working correctly.
* Running a multi-container application: You need to run a complex application that consists of multiple containers that need to communicate with each other and share resources.
* Securing your application with TLS/SSL: You need to enable HTTPS for your application, using TLS/SSL certificates to encrypt traffic between clients and the server.
* Running a batch job: You need to run a batch job that performs a long-running computation or data processing task, with the ability to recover from failures and to run the job on a schedule.
* Running a stateless service: You need to run a stateless service that can be scaled up and down based on demand, with the ability to handle requests and respond to failures gracefully.
* Running a stateful service: You need to run a stateful service that requires persistent storage and a stable network identity, with the ability to recover from failures and to handle data replication and consistency.
* Configuring a CI/CD pipeline: You need to set up a continuous integration and continuous deployment pipeline that automatically builds and deploys your application to Kubernetes, with the ability to run tests, perform code reviews, and ensure that changes are rolled out gradually and safely.
* Managing resource usage with resource quotas: You need to ensure that your application's Pods and containers are not using more resources than they are allowed, using Kubernetes resource quotas to limit the amount of CPU, memory, and other resources that can be used.
* Managing networking with a Service Mesh: You need to manage complex network traffic patterns between your application's Pods and services, using a Service Mesh solution like Istio or Linkerd to provide advanced traffic management, observability, and security features.
* Running a multi-cloud or hybrid cloud application: You need to run your application across multiple cloud providers or on-premises data centers, using Kubernetes to provide a common abstraction layer that abstracts away the underlying infrastructure and provides a consistent API for deploying and managing your application.

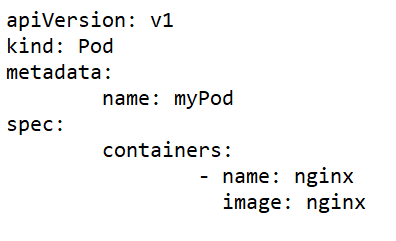
**SECTION 1 : Level – easy**

1. **POD**
   1. Check number of pods in the system (ns : default).

kubectl get pods

* 1. Create a new POD with nginx image.

*Pod-def.yaml*



kubectl create -f Pod-def.yaml

* 1. How to check which image is used in a deployed pod, node on which Pod is deployed, and how many containers are part of existing pod?

kubectl describe pod <pd-name>

* 1. Delete an existing Pod.

kubectl delete pod <pd-name>

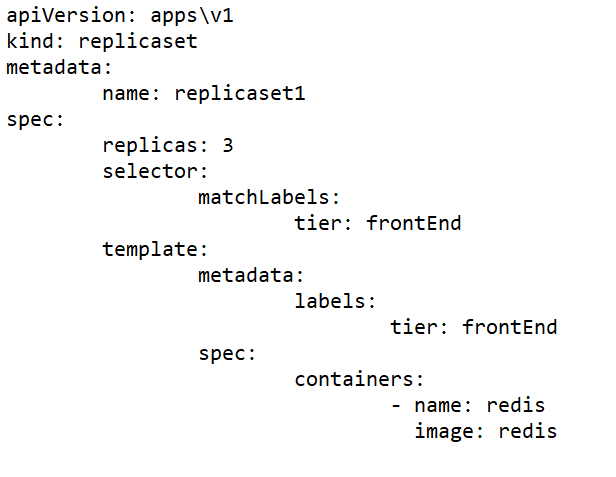
kubectl delete -f <pd\_name>.yaml

1. **REPLICA SETS**
   1. Check number of Replica Sets in the system (ns : default).

kubectl get rs

* 1. Create replica-set

*rs.yaml*



kubectl create -f rs.yaml

* 1. If you delete one pod created from rs. What will happen? rs will create a new Pod automatically
  2. Edit existing rs

kubectl edit rs <rs-name>

<OR> kubectl get rs <rs-name> -o yaml > filename.yaml

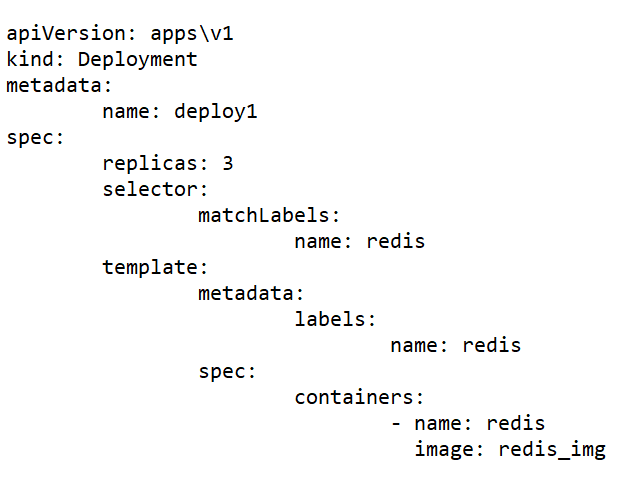
kubectl apply -f <fn>.yaml

kubectl replace - -force -f <>.yaml

* 1. Use scale command to scale down pods in a rs kubectl scale rs <rs-name> --replicas=<val>

1. **DEPLOYMENTS**
   1. Create a deployment

*deploy-def.yaml*



kubectl create -f deploy-def.yaml

1. **NAMESPACES**
   1. Create a dev namespace

kubectl create namespace dev

* 1. Fetch number of pods in dev namespace

kubectl get pods -n dev

* 1. Create a Pod in dev namespace using pd.yaml

kubectl create -f pd.yaml -n dev

* 1. How to check a pod lies in which namespace?

kubectl get pods –all-namespaces

1. **IMPERATIVE COMMANDS**
   1. Deploy a pod named nginx-pod using the nginx:alpine image.

kubectl run nginx-pod - -image nginx:alpine

* 1. Create a service redis-service to expose the redis application within the cluster on port 6379.

kubectl expose pod redis - -port=6379 - -name=redis-service

* 1. Create a deployment named webapp using the image webapp-color with 3 replicas.

kubectl create deployment webapp - -image=webapp-color –replicas=3

* 1. Create a new pod called custom-nginx using the nginx image and expose it on container port 8080.

kubectl run custom-nginx - -image=nginx - -port=8080

1. **CONFIGURATION**
   1. How will you pass command and argument which will run when container starts?

Add below in pod-def.yaml spec section

command: [“cmd”]

args: [“arg0”,”arg1”]

<OR>

command:

* + - * “cmd”
      * “arg0”
      * “arg1”

<OR>

Add below in DockerFile used

ENTRYPOINT [“cmd”]

CMD [“arg0”,”arg1”]

<OR>

kubectl run <pod-name> --image=<img-name > --command -- <cmd> <arg0> <arg1>

* 1. Create a config-map.

kubectl create cm <cm\_name> - -from-literal=<data>

* 1. How to refer config map in pod-definition file?

Use below content in spec: containers section

envFrom:

* + - * configMapRef:

name: <cm\_name>

* 1. Create a secret.

kubectl create secret generic <sec-name> - -from-literal=<data>

* 1. How to refer secret in a Pod definition file?

envFrom:

* + - * secretRef:

name: <sec>

* 1. Add security-Context in pod definition file for user ID 1001 and capability of SYS\_TIME

securityContext:

runAsUser: 1001

capabilities:

add: [“SYS\_TIME”]

* 1. Add a Service Account in deployment file.

Add following inside template.spec:

serviceAccountName: <sa-name>

* 1. Create a taint on node01

kubectl taint node node01 <key>=<value>:<NoSchedule>

* 1. Add a toleration on a pod

spec:

tolerations:

* + - * key: <>

value: <>

effect: <>

operator: Equal

* 1. Remove a taint from the node

kubectl taint node node01 node-role.kubernetes.io/control-plane:NoSchedule

* 1. Apply a label color=blue to node node01

Kubectl label node node01 color=blue

1. **MONITORING**
   1. How to view the logs of a pod?

kubectl logs <pd\_name>

* 1. How to execute shell in a pod?

kubectl exec <pd\_name> -I -t <cmd>

* 1. How get the node and pod using most resources?

kubectl top node

* 1. If pod crashed check the previous logs of the pod.

*kubectl logs busybox -p*

1. **POD DESIGN**
   1. Fetch pods with the label as env=dev

kubectl get pods - - selector=env=dev

* 1. Fecth all Objects in k8s

Kubectl get all

* 1. Edit image in deployment

kubectl set image deployment/<deploy\_name> <container-name>=<image-name>

1. **JOB and CRONJOB**
   1. Create a job

kubectl create job <name> --image=<img\_name>

* 1. What is cronjob and schedule parameter

A Cron job is used to schedule recurring tasks in Kubernetes. To create a Cron job, you need to specify a schedule that determines when the job will run. The schedule parameter uses the Cron format, which consists of five fields:

\* \* \* \* \*

| | | | +----- day of the week (0 - 6) (Sunday = 0)

| | | +---------- month (1 - 12)

| | +--------------- day of the month (1 - 31)

| +-------------------- hour (0 - 23)

+------------------------- minute (0 - 59)

1. **SERVICES AND NETWORKING**
   1. Fetch services in all namespaces

kubectl get svc -A

* 1. Fetch Network Policies in default namespace

kubectl get netpol

* 1. Fetch all ingress resources

kubectl get ingress -A

**SECTION 2 : Level – Advance**

1. List all the pods showing name and namespace with a json path expression

kubectl get pods -o=jsonpath="{.items[\*]['metadata.name', 'metadata.namespace']}"

1. Output the yaml file of the pod you just created without the cluster-specific information

*kubectl get po nginx -o yaml --export*

1. Delete the pod you just created without any delay (force delete)

*kubectl delete po nginx --grace-period=0 --force*

1. Change the Image version back to *1.17.1**for the pod you just updated and observe the changes*

*kubectl set image pod/nginx nginx=nginx:1.17.1   
 kubectl get po nginx -w # watch it*

1. Check the Image version without the describe command

*kubectl get po nginx -o jsonpath='{.spec.containers[].image}{"\n"}'*

1. Execute the simple shell on the pod *kubectl exec -it nginx /bin/sh*
2. Get the IP Address of the pod you just created

*kubectl get po nginx -o wide*

1. Create a busybox pod with command sleep 3600

*kubectl run busybox --image=busybox --restart=Never -- /bin/sh -c "sleep 3600"*

1. Check the connection of the nginx pod from the busybox pod

*kubectl get po nginx -o wide// check the connection  
kubectl exec -it busybox -- wget -o- <IP Address>*

1. Create a busybox pod and echo message ‘How are you’ and have it deleted immediately *kubectl run busybox --image=nginx --restart=Never -it --rm -- echo "How are you"*
2. Create an nginx pod and list the pod with different levels of verbosity *kubectl run nginx --image=nginx --restart=Never --port=80   
   kubectl get po nginx --v=7  
   kubectl get po nginx --v=8  
   kubectl get po nginx --v=9*
3. List the nginx pod with custom columns POD\_NAME and POD\_STATUS

*kubectl get po -o=custom-columns="POD\_NAME:.metadata.name, POD\_STATUS:.status.containerStatuses[].state"*

1. List all the pods sorted by name

*kubectl get pods --sort-by=.metadata.name*

1. List all the pods sorted by created timestamp

*kubectl get pods--sort-by=.metadata.creationTimestamp*

1. Check the logs of each container that you just created

*kubectl logs busybox -c busybox1  
kubectl logs busybox -c busybox2*

1. Check the previous logs of the second container busybox2 if any

*kubectl logs busybox -c busybox2 --previous*

1. Run command ls in the third container busybox3 of the above pod

*kubectl exec busybox -c busybox3 -- ls*

1. Show metrics of the above pod containers and puts them into the file.log

*kubectl top pod busybox --containers// putting them into file  
kubectl top pod busybox --containers > file.log*

1. *Exec into both containers and verify that main.txt exist and query the main.txt from sidecar container with curl localhost*   
   kubectl exec -it multi-cont-pod -c main-container -- sh  
   cat /var/log/main.txt

kubectl exec -it multi-cont-pod -c sidecar-container -- sh  
cat /usr/share/nginx/html/index.html

kubectl exec -it multi-cont-pod -c sidecar-container -- sh  
# apt-get update && apt-get install -y curl  
# curl localhost

1. Extract documentation for livenessProbe

kubectl explain Pod.spec.containers.livenessProbe

1. Set Configuration context

kubectl config use-context <context>

1. Get the pods with labels env=dev and env=prod and output labels as well

kubectl get pods -l 'env in (dev,prod)' –show-label

1. Change the label for one of the pod to env=uat

kubectl label pod/<pod-name> env=uat –overwrite

1. Remove the labels for the pods that we created now and verify all the labels are removed

kubectl label pod nginx-dev{1..3} env-

1. Verify the pod nginx that we just created has this label

kubectl describe po nginx | grep Labels

1. Annotate the pods with name=webapp

kubectl annotate pod <pod\_name> name=webapp

1. Get the deployment rollout status

kubectl rollout status deploy <deploy-name>

1. Check rollout history of deployment

kubectl rollout history deploy <deploy-name>

1. Undo deployment to previous version

kubectl rollout undo deploy <deploy-name>

1. Pause the rollout of the deployment

kubectl rollout pause deploy webapp

1. Resume the rollout of the deployment

kubectl rollout resume deploy webapp

1. Apply the autoscaling to this deployment with minimum 10 and maximum 20 replicas and target CPU of 85% and verify hpa is created and replicas are increased to 10 from 1

kubectl autoscale deploy webapp --min=10 --max=20 --cpu-percent=85

kubectl get hpa

kubectl get pod -l app=webapp

1. List all resources shortcuts

kubectl api-resources

1. List docker images

docker images

1. Build a docker image using the Dockerfile and name it webapp-color

docker build -t webapp-color .

1. Run an instance of the image webapp-color and publish port 8080 on the container to 8282 on the host.

docker run -p 8282:8080 --name my-webapp webapp-color

1. What is the base Operating System used by the python:3.6 image?

docker run -it --name temp-python36 python:3.6 /bin/bash

cat /etc/os-release

1. Build a docker image using the Dockerfile and name it webapp-color and tag lite

docker build -t webapp-color:lite .

1. Create a configmap named keyvalcfgmap and read data from the file config.txt

kubectl create cm keyvalcfgmap --from-file=config.txt

1. *Create an env file file.env with var1=val1 and create a configmap envcfgmap from this env file and verify the configmap*

*echo var1=val1 > file.env*

*cat file.env*

*kubectl create cm envcfgmap --from-env-file=file.env*

*kubectl get cm envcfgmap -o yaml –export*

**Section: 3 - ADDITIONAL QUESTIONS**

1. Create a Pod named my-pod with a single container that runs the command sleep 3600, and set the container's CPU limit to 0.5 CPU.
2. Update the Deployment named my-app to use image my-image:1.1.0.
3. Create a Service named my-service that exposes port 80 and forwards traffic to port 8080 on the Pods with the app=my-app label.
4. Create a CronJob named my-cronjob that runs the command /usr/bin/my-job every 5 minutes.
5. Modify the Pod named my-pod to use an Init Container that runs the command echo "Init container running" before the main container starts.
6. Create a PersistentVolumeClaim named my-pvc that requests 1Gi of storage and uses the standard storage class.
7. Scale the Deployment named my-app to 5 replicas.
8. Modify the Service named my-service to use a NodePort type and expose the port 30000 on all nodes.
9. Create a Deployment named my-app that runs two containers: one with image my-web-image:1.0 and port 80 exposed, and another with image my-worker-image:1.0 and command /usr/bin/my-worker that runs once at startup.
10. Create a ConfigMap named my-config that reads data from a file /etc/config/my-config.yaml in the my-namespace namespace.
11. Create a Pod named my-pod with two containers: one with image my-app-image:1.0 and command sleep 3600, and another with image busybox and command wget my-app:80 that waits until the my-app container is ready before running.
12. Create a CronJob named my-cronjob that runs a command /usr/bin/my-job every day at 2:30 PM, and sends an email to my-email@example.com with the job output.
13. Create a Service named my-service that exposes two ports: port 80 for HTTP traffic and port 443 for HTTPS traffic. The service should route traffic to the my-app Deployment only if the requesting client's IP address is in a list of allowed IP addresses.
14. Create a PersistentVolumeClaim named my-pvc that uses a StorageClass named my-sc, and has an access mode of ReadWriteMany and a storage request of 5Gi.
15. Create a Job named my-job that runs a command echo "Hello, world!" in a Pod. The Job should be terminated if it runs for longer than 10 seconds.
16. Create a Pod that runs with a ServiceAccount named my-sa and a security context that sets the fsGroup to 1000 and the runAsUser to 1000.
17. Create a Deployment named my-app that has a readiness probe that checks a TCP socket on port 8080, with a failure threshold of 5 and a timeout of 2 seconds.
18. Create a Pod named my-pod that uses a downward API volume to expose the Pod's IP address, Pod name, and Pod namespace as environment variables.

**Scenario Based Questions**

1. You are deploying a microservices application to Kubernetes, and you have multiple teams responsible for different parts of the application. How would you organize the application's resources to ensure that each team can deploy and manage their components independently?
2. One of your application's Pods is crashing frequently, and you suspect that it is running out of memory. How would you troubleshoot the issue and identify the root cause?
3. Your application needs to communicate with a database that is running outside of Kubernetes. How would you configure your application's Pods to connect to the database securely and reliably?
4. You need to deploy a new version of your application to Kubernetes, but you want to minimize downtime and ensure that any in-progress requests are not interrupted. How would you perform a rolling update of the application's Pods?
5. Your application needs to store data persistently, but you are not sure which type of storage to use. What are some factors that you should consider when selecting a storage solution for your application, and how would you evaluate different options?
6. You want to monitor your application's performance and health using Prometheus and Grafana. How would you configure your application's Pods to expose metrics, and how would you set up Prometheus and Grafana to collect and visualize the metrics?
7. Your application has a component that requires GPU resources to run. How would you configure your Kubernetes cluster to support GPU-accelerated workloads, and how would you ensure that the workload is scheduled on a node with a GPU?
8. You need to configure your application to scale automatically based on incoming traffic. How would you use Kubernetes Horizontal Pod Autoscaling (HPA) to dynamically scale the number of Pods based on CPU or memory utilization?
9. You want to deploy a stateful application to Kubernetes that requires a fixed number of replicas and needs to maintain a stable network identity. How would you use a StatefulSet to manage the application's Pods, and how would you ensure that each Pod has a unique network identity?
10. Your application needs to process data from an external message queue that requires a long-lived connection. How would you deploy a message consumer as a Kubernetes Pod, and how would you ensure that the Pod can maintain a persistent connection to the message queue?

**ANSWERS**

1. Organizing an application for multiple teams: You would need to use Kubernetes namespaces and RBAC (role-based access control) to create separate environments for each team, and you might also use Kubernetes labels and selectors to allow each team to manage their own resources.
2. Troubleshooting a crashing Pod: You would need to use Kubernetes logs and events to diagnose the issue, and you might also use tools like kubectl exec or a debug container to explore the Pod's filesystem and processes.
3. Connecting to an external database: You would need to use Kubernetes secrets to store the database credentials, and you might also use a Service and an Endpoint object to expose the database to your application's Pods.
4. Performing a rolling update: You would need to use a Kubernetes Deployment object with a rolling update strategy to update your application's Pods without downtime, and you might also use readiness and liveness probes to ensure that the Pods are healthy.
5. Selecting a storage solution: You would need to consider factors such as performance, reliability, durability, and cost, and you might evaluate different storage options such as Persistent Volumes, Storage Classes, or third-party storage solutions.
6. Monitoring with Prometheus and Grafana: You would need to configure your application to expose metrics in a format that Prometheus can scrape, and you would need to set up a Prometheus server and a Grafana dashboard to collect and visualize the metrics.
7. Supporting GPU-accelerated workloads: You would need to use a Kubernetes GPU device plugin to expose the GPU resources to your application's Pods, and you might also use a Pod or NodeSelector to schedule the workload on a node with a GPU.
8. Scaling with Horizontal Pod Autoscaling (HPA): You would need to set up a Kubernetes HPA object that automatically scales the number of Pods based on CPU or memory utilization, and you might also use a Kubernetes Ingress or Service object to distribute traffic to your Pods.
9. Managing stateful applications with StatefulSets: You would need to use a StatefulSet object to manage your application's Pods and to provide each Pod with a stable network identity, and you might also use a Kubernetes Persistent Volume or StatefulSet volume to provide persistent storage.
10. Connecting to an external message queue: You would need to use a Kubernetes Pod or Deployment object to run the message consumer, and you might also use a Kubernetes ConfigMap to store the message queue configuration and a Kubernetes Service object to provide a stable network address.