Q1: What is Kubernetes?
A1: Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a framework for running and coordinating containers across a cluster of machines.

Q2: What are the main components of Kubernetes?
A2: The main components of Kubernetes are:
- Master node: Manages the cluster and makes global decisions about the cluster state.
- Worker node: Executes tasks assigned by the master node.
- Pod: The basic building block of Kubernetes, which encapsulates one or more containers.
- ReplicaSet: Ensures that a specified number of pod replicas are always running.
- Deployment: Manages the rollout and updates of ReplicaSets and pods.
- Service: Provides networking and load balancing for pods.

Q3: What is a Pod in Kubernetes?
A3: A Pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a running process in a cluster and encapsulates one or more containers, shared storage, and network resources.

Q4: What is a ReplicaSet?
A4: A ReplicaSet is a Kubernetes object that ensures a specified number of pod replicas are running at any given time. It monitors the status of pods and creates or terminates replicas to match the desired state.

Q5: What is a Deployment?
A5: A Deployment is a Kubernetes object that provides declarative updates for pods and ReplicaSets. It allows you to define the desired state of your application and manages the rollout and rollback of changes.

Q6: How does Kubernetes handle container scaling?
A6: Kubernetes provides horizontal scaling through the use of ReplicaSets or Deployments. By adjusting the desired number of replicas, you can scale up or down the number of running instances of a pod.

Q7: What is a Kubernetes Namespace?
A7: A Kubernetes Namespace is a virtual cluster inside a Kubernetes cluster. It provides a way to divide cluster resources into logical groups, enabling multiple teams or projects to use the same cluster without interfering with each other.

Q8: How does Kubernetes handle service discovery and load balancing?
A8: Kubernetes uses a Service object to provide service discovery and load balancing. A Service defines a stable network endpoint to access a group of pods. It automatically load balances traffic between the pods and provides DNS resolution for the Service's hostname.

Q9: What is the role of a Kubernetes ConfigMap?
A9: A ConfigMap is a Kubernetes object that stores configuration data as key-value pairs. It allows you to separate configuration from application code and provides a way to inject configuration into containers at runtime.

Q10: How can you expose a Kubernetes deployment to the outside world?
A10: You can expose a Kubernetes deployment to the outside world by creating a Service of type `LoadBalancer`. This type of Service provisions an external load balancer that routes traffic from the external network to the pods.

Kubernetes interview question and answers for INTERMEDIATE LEVEL

Q1: What is a StatefulSet in Kubernetes?
A1: A StatefulSet is a Kubernetes object used for managing stateful applications. It provides guarantees about the ordering and uniqueness of pods, stable network identities, and stable storage for each pod. StatefulSets are commonly used for databases, key-value stores, and other stateful workloads.

Q2: Explain the difference between a Deployment and a StatefulSet.
A2: Deployments are suitable for stateless applications, where each instance of the application is identical and can be scaled horizontally. StatefulSets, on the other hand, are used for stateful applications that require stable network identities and persistent storage. StatefulSets provide ordering guarantees and allow for scaling vertically.

Q3: How does Kubernetes handle storage for applications?
A3: Kubernetes provides several storage options, including Persistent Volumes (PV) and Persistent Volume Claims (PVC). PVs are resources provisioned in the cluster, while PVCs are requests for storage resources by applications. PVCs consume PVs, which can be dynamically provisioned by storage classes or statically provisioned by an administrator.

Q4: What is a DaemonSet in Kubernetes?
A4: A DaemonSet is a Kubernetes object that ensures that a specific pod runs on every node in a cluster. It is useful for running background tasks, monitoring agents, or any workload that needs to be deployed to all nodes in a cluster.

Q5: Explain the concept of a rolling update in Kubernetes.
A5: A rolling update is a strategy used by Kubernetes to update a Deployment or StatefulSet without causing downtime. It gradually replaces existing pods with new ones, ensuring that a specified number of pods are available at all times. This strategy allows for seamless updates and rollbacks if necessary.

Q6: What are Kubernetes labels and selectors?
A6: Labels are key-value pairs attached to Kubernetes objects to help identify and organize them. Selectors are used to query objects based on labels. They allow for grouping and selecting subsets of objects that share common labels, enabling more flexible management and control.

Q7: What is a Kubernetes Operator?
A7: A Kubernetes Operator is an extension to the Kubernetes API that introduces custom resources and controllers. It encapsulates domain-specific knowledge and automates the management of complex applications or services. Operators are used to simplify the deployment, configuration, and lifecycle management of stateful applications.

Q8: How can you perform rolling updates with zero downtime in Kubernetes?
A8: Rolling updates with zero downtime can be achieved by using readiness probes in Kubernetes. Readiness probes allow the Kubernetes control plane to verify if a pod is ready to serve traffic before sending requests to it. By configuring appropriate readiness probes, you can ensure that pods are only added to the load balancer once they are ready to handle traffic, avoiding any disruption during updates.

Q9: What is Kubernetes Helm?
A9: Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. It uses charts, which are pre-configured packages containing all the necessary resources and configurations for running an application in Kubernetes. Helm allows for versioning, dependency management, and easy installation of applications.

Q10: How does Kubernetes handle security and access control?
A10: Kubernetes provides various security features, such as Role-Based Access Control (RBAC), which allows fine-grained control over who can access and perform actions on cluster resources. It also supports pod security policies, network policies, and secrets management to ensure secure communication and data protection within the cluster.

Kubernetes interview question and answers for ADVANCED LEVEL

1. Q: What are StatefulSets in Kubernetes, and when would you use them?
A: StatefulSets are a Kubernetes controller used to manage stateful applications. They provide guarantees about the ordering and uniqueness of pods, making them suitable for applications

that require stable network identities or persistent storage. StatefulSets are often used for databases, such as MySQL or PostgreSQL, where each pod has a unique identity and persistent storage attached.

2. Q: How does Kubernetes handle rolling updates? Explain the update strategy and any available options.
A: Kubernetes handles rolling updates by gradually replacing the existing pods with the new version while ensuring the availability of the application. The default update strategy is called "RollingUpdate," where Kubernetes replaces pods incrementally, verifying the health of each new pod before proceeding to the next. This strategy ensures that the application remains available during the update process. There are additional options available, such as setting a maximum surge or maximum unavailable pods to control the update behavior further.

3. Q: What are Kubernetes Operators, and how do they extend the functionality of Kubernetes?
A: Kubernetes Operators are a way to package, deploy, and manage applications using Kubernetes primitives. They extend the functionality of Kubernetes by encapsulating complex, application-specific operational knowledge. Operators automate application-specific tasks, such as managing backups, scaling, upgrades, and failure recovery. They leverage the Kubernetes API to provide a more declarative and scalable way of managing applications.

4. Q: How does Kubernetes perform service discovery and load balancing for pods within a cluster?
A: Kubernetes uses a combination of DNS-based service discovery and a built-in load balancer called kube-proxy. Each Service within Kubernetes is assigned a unique DNS name, which can be resolved to the cluster IP. The kube-proxy component then load balances traffic across the pods associated with the Service, distributing requests evenly.

5. Q: Explain the concept of a PodDisruptionBudget (PDB) in Kubernetes and its significance.
A: A PodDisruptionBudget (PDB) is a resource in Kubernetes used to limit the number of pods that can be simultaneously unavailable during disruptive events, such as node maintenance or pod evictions. PDBs ensure that a certain number of pods are always available for the application, enforcing high availability guarantees. PDBs help prevent unwanted downtime by allowing controlled disruptions and avoiding scenarios where critical applications are left with insufficient replicas.

6. Q: What are custom resource definitions (CRDs) in Kubernetes, and how do they enable the creation of custom resources?
A: Custom Resource Definitions (CRDs) allow users to define their custom resources and extend the Kubernetes API. CRDs enable the creation of custom resources and controllers that can manage them. With CRDs, users can introduce new object types and declaratively manage them through the Kubernetes API server. This extensibility allows Kubernetes to be adapted to various use cases and domains beyond the core set of resources.

7. Q: Explain the concept of resource quotas in Kubernetes and how they help with cluster resource management.
A: Resource quotas in Kubernetes are used to limit and track the resource consumption of objects within a namespace. They help ensure fair resource allocation, prevent resource starvation, and avoid any single application from consuming excessive resources. Resource quotas can be set for CPU, memory, storage, and other resource types. They play a crucial role in managing multi-tenant clusters and preventing runaway applications from impacting the overall cluster performance.

Kubernetes interview question and answers SCENARIO BASED

1. **Scenario: You have a Kubernetes cluster with multiple worker nodes. One of the nodes becomes unresponsive and needs to be replaced. Explain the steps you would take to replace the node without affecting the availability of applications running on the cluster.**

Answer: To replace the unresponsive node without affecting application availability, I would follow these steps:

1. Drain the unresponsive node: Use the kubectl drain command to gracefully evict all the pods running on the unresponsive node. This ensures that the pods are rescheduled on other healthy nodes.
2. Cordon the unresponsive node: Use the kubectl cordon command to mark the node as unschedulable. This prevents new pods from being scheduled on the node while it's being replaced.
3. Remove the unresponsive node: Once all the pods are safely rescheduled, you can remove the unresponsive node from the cluster, either by repairing it or provisioning a new node.
4. Uncordon the node: Once the new node is ready, use the kubectl uncordon command to mark it as schedulable again. This allows new pods to be scheduled on the replacement node.

2. **Scenario: You have a stateful application running on Kubernetes that requires persistent storage. How would you ensure that the data is retained when the pods are rescheduled or updated?**

Answer: To ensure data retention for a stateful application, I would use the following Kubernetes features:

1. Persistent Volumes (PVs) and Persistent Volume Claims (PVCs): I would create a Persistent Volume that represents the storage resource (e.g., a network-attached disk) and then create a Persistent Volume Claim that binds to the PV. This ensures that the same volume is attached to the pod when it's rescheduled or updated.
2. StatefulSets: I would use StatefulSets to manage the stateful application. StatefulSets ensure that each pod has a unique identity and stable network

identity, allowing the pod to retain its storage even during rescheduling or updates. StatefulSets can use PVCs to attach the appropriate Persistent Volumes to each pod.

3. Storage Classes: I would define Storage Classes to dynamically provision Persistent Volumes based on predefined storage requirements. This allows for automated volume provisioning when new PVCs are created. By leveraging these features, the stateful application can maintain its data even when pods are rescheduled, updated, or scaled up/down.

3. Scenario: A Kubernetes pod is stuck in a "Pending" state. What could be the possible reasons, and how would you troubleshoot it?

Possible reasons for a pod being stuck in the "Pending" state could include:
- Insufficient resources: Check if the cluster has enough resources (CPU, memory, storage) to accommodate the pod. You can use the `kubectl describe pod <pod-name>` command to view detailed information about the pod, including any resource-related issues.
- Unschedulable nodes: Check if all the nodes in the cluster are in the "Ready" state and can schedule the pod. You can use the `kubectl get nodes` command to see the node status.
- Pod scheduling constraints: Verify if the pod has any scheduling constraints or affinity/anti-affinity rules that are preventing it from being scheduled. Check the pod's YAML or manifest file for any such specifications.
- Persistent Volume (PV) availability: If the pod requires a Persistent Volume, ensure that the required storage is available and accessible.
- Network-related issues: Check if there are any network restrictions or misconfigurations preventing the pod from being scheduled or communicating with other resources.

4. Scenario: You have a Kubernetes Deployment with multiple replicas, and some pods are failing health checks. How would you identify the root cause and fix it?

To identify the root cause and fix failing health checks for pods in a Kubernetes Deployment:
- Check the pod's logs: Use the `kubectl logs <pod-name>` command to retrieve the logs of the failing pod. Inspect the logs for any error messages or exceptions that could indicate the cause of the failure.
- Verify health check configurations: Examine the readiness and liveness probe configurations in the Deployment's YAML or manifest file. Ensure that the endpoints being probed are correct, the expected response is received, and the success criteria are appropriately defined.
- Debug container startup: If the pods are failing to start, check the container's startup commands, entrypoints, or initialization processes. Use the `kubectl describe pod <pod-name>` command to get detailed information about the pod, including any container-related errors.
- Resource constraints: Inspect the resource requests and limits for the pods. It's possible that the pods are exceeding the allocated resources, causing failures. Adjust the resource specifications as necessary.
- Image issues: Verify that the Docker image being used is correct and accessible. Ensure that the image's version, registry, and repository details are accurate.
- Rollout issues: If the pods were recently deployed or updated, ensure that the rollout process

completed successfully. Check the deployment's status using `kubectl rollout status <deployment-name>` and examine any rollout history with `kubectl rollout history <deployment-name>`.

5. Scenario: You need to scale a Kubernetes Deployment manually. How would you accomplish this?

To manually scale a Kubernetes Deployment:
- Use the `kubectl scale` command: Run `kubectl scale deployment/<deployment-name> --replicas=<number-of-replicas>` to scale the deployment. Replace `<deployment-name>` with the name of your deployment, and `<number-of-replicas>` with the desired number of replicas.
- Alternatively, update the Deployment YAML: Modify the `replicas` field in the Deployment's YAML or manifest file to the desired number of replicas. Then, apply the changes using `kubectl apply -f <path-to-deployment-yaml>`.

6. Scenario: You have a Kubernetes cluster with multiple worker nodes, and some pods are experiencing high CPU usage. How would you investigate and mitigate this issue?

To investigate and mitigate high CPU usage in Kubernetes pods:
- Identify the affected pods: Use `kubectl top pods` to view CPU and memory usage across the cluster and identify the pods with high CPU usage.
- Check pod resource limits: Verify if the affected pods have appropriate resource limits defined. If the limits are too low, the pods may struggle to handle the workload, resulting in high CPU usage. Adjust the resource limits accordingly.
- Analyze application code: Review the application code running inside the pods to identify any inefficiencies or resource-intensive operations that could be causing the high CPU usage. Optimize the code where possible.
- Scale horizontally: If the high CPU usage is due to increased traffic or workload, consider scaling the affected deployment horizontally by increasing the number of replicas. This distributes the load across multiple pods and can help alleviate CPU pressure.
- Implement resource quotas: Define resource quotas at the namespace level to limit the amount of CPU resources each pod can consume. This prevents individual pods from monopolizing the CPU and affecting other workloads.

7. Scenario: You have a Kubernetes cluster hosting multiple microservices, and you need to enforce communication restrictions between them. How would you achieve this?

To enforce communication restrictions between microservices in a Kubernetes cluster:
- Use network policies: Network policies are Kubernetes objects that control the traffic flow between pods based on defined rules. Create network policies to specify which pods can communicate with each other based on labels, namespaces, or IP ranges. Configure the policies to allow or deny traffic as per your desired communication restrictions.
- Deny all traffic by default: Set up a default-deny rule in the network policies to block all inter-pod communication by default. Then, explicitly define policies to allow specific communication paths between authorized microservices.
- Label pods and apply policies selectively: Label the microservice pods based on their roles or

functions. Then, define network policies that target specific labels, enabling you to enforce communication restrictions at a granular level.
- Validate and test policies: Regularly validate and test the network policies to ensure they are functioning as intended. Deploy test pods with different labels and verify if the communication restrictions are being correctly enforced.

8. Scenario: You have a Kubernetes cluster, and you want to schedule certain pods on specific nodes based on node availability and custom requirements. How would you achieve this?

To schedule pods on specific nodes based on availability and custom requirements in Kubernetes:
- Use node affinity: Node affinity allows you to define rules for pod scheduling based on node labels. Assign specific labels to nodes that meet the desired requirements. Then, specify the corresponding node affinity rules in the pod's YAML or manifest file, ensuring that the pod gets scheduled on the desired nodes.
- Combine node affinity with node taints: Node taints can be used to mark specific nodes to be tolerated or avoided by pods. Taint the nodes that need to be scheduled for specific pods. Then, define corresponding tolerations in the pod's YAML or manifest file to ensure that the pod can be scheduled on those nodes.
- Leverage node selectors: Node selectors are a simple way to schedule pods on specific nodes. Assign labels to the nodes and define the matching node selector in the pod's YAML or manifest file to direct the scheduler to schedule the pod on the desired nodes.
- Utilize node-specific resources: If certain nodes have specialized hardware or unique capabilities required by specific pods, label those nodes accordingly. Then, use node affinity rules to schedule pods that require those resources on the corresponding nodes.

Kubernetes Commands

1. `kubectl apply -f deployment.yaml`:
 - Output: The deployment defined in the `deployment.yaml` file is created or updated.
 - Explanation: This command applies the configuration defined in the `deployment.yaml` file, creating or updating the deployment accordingly.

2. `kubectl get pods`:
 - Output: A list of pods in the default namespace.
 - Explanation: Lists all the pods running in the default namespace.

3. `kubectl get pods --namespace=my-namespace`:
 - Output: A list of pods in the `my-namespace` namespace.
 - Explanation: Lists all the pods running in the `my-namespace` namespace.

4. `kubectl describe pod my-pod`:
  - Output: Detailed information about the `my-pod` pod.
  - Explanation: Retrieves detailed information about the `my-pod` pod, including its status, events, and containers.

5. `kubectl logs my-pod`:
  - Output: The logs of the `my-pod` pod.
  - Explanation: Retrieves and displays the logs generated by the `my-pod` pod.

6. `kubectl exec -it my-pod -- /bin/bash`:
  - Output: Opens a shell inside the `my-pod` pod.
  - Explanation: Allows interactive shell access to the `my-pod` pod, enabling you to execute commands directly inside the container.

7. `kubectl port-forward my-pod 8080:80`:
  - Output: Forwards local port 8080 to port 80 of the `my-pod` pod.
  - Explanation: Sets up port forwarding, allowing you to access a specific port of the `my-pod` pod locally on port 8080.

8. `kubectl delete pod my-pod`:
  - Output: The `my-pod` pod is deleted.
  - Explanation: Deletes the `my-pod` pod and terminates its execution.

9. `kubectl scale deployment my-deployment --replicas=3`:
  - Output: The `my-deployment` deployment is scaled to 3 replicas.
  - Explanation: Adjusts the number of replicas for the `my-deployment` deployment to 3, effectively increasing the number of instances.

10. `kubectl get services`:
  - Output: A list of services in the default namespace.
  - Explanation: Lists all the services running in the default namespace.

11. `kubectl get services --namespace=my-namespace`:
  - Output: A list of services in the `my-namespace` namespace.
  - Explanation: Lists all the services running in the `my-namespace` namespace.

12. `kubectl describe service my-service`:
  - Output: Detailed information about the `my-service` service.

   - Explanation: Retrieves detailed information about the `my-service` service, including its type, IP, and ports.

13. `kubectl expose deployment my-deployment --port=8080 --target-port=80 --type=LoadBalancer`:
   - Output: The `my-deployment` deployment is exposed as a LoadBalancer service on port 8080.
   - Explanation: Creates a service of type LoadBalancer that exposes the `my-deployment` deployment on port 8080, forwarding traffic to port 80 of the pods.

14. `kubectl create namespace my-namespace`:
   - Output: The `my-namespace` namespace is created.
   - Explanation: Creates a new namespace called `my-namespace`.

15. `kubectl get namespaces`:
   - Output: A list of namespaces in the cluster.
   - Explanation

: Lists all the namespaces available in the cluster.

16. `kubectl describe namespace my-namespace`:
   - Output: Detailed information about the `my-namespace` namespace.
   - Explanation: Retrieves detailed information about the `my-namespace` namespace, including its status and resource limits.

17. `kubectl delete namespace my-namespace`:
   - Output: The `my-namespace` namespace is deleted.
   - Explanation: Deletes the `my-namespace` namespace and all its associated resources.

18. `kubectl create configmap my-config --from-file=config.ini`:
   - Output: The `my-config` ConfigMap is created.
   - Explanation: Creates a ConfigMap called `my-config` using the contents of the `config.ini` file.

19. `kubectl get configmaps`:
   - Output: A list of ConfigMaps in the default namespace.
   - Explanation: Lists all the ConfigMaps available in the default namespace.

20. `kubectl describe configmap my-config`:
- Output: Detailed information about the `my-config` ConfigMap.
- Explanation: Retrieves detailed information about the `my-config` ConfigMap, including its data and associated pods.

21. `kubectl delete configmap my-config`:
- Output: The `my-config` ConfigMap is deleted.
- Explanation: Deletes the `my-config` ConfigMap.

22. `kubectl create secret generic my-secret --from-literal=password=abc123`:
- Output: The `my-secret` Secret is created.
- Explanation: Creates a Secret called `my-secret` with the key `password` and the value `abc123`.

23. `kubectl get secrets`:
- Output: A list of secrets in the default namespace.
- Explanation: Lists all the secrets available in the default namespace.

24. `kubectl describe secret my-secret`:
- Output: Detailed information about the `my-secret` Secret.
- Explanation: Retrieves detailed information about the `my-secret` Secret, including its type and data.

25. `kubectl delete secret my-secret`:
- Output: The `my-secret` Secret is deleted.
- Explanation: Deletes the `my-secret` Secret.

26. `kubectl create ingress my-ingress --rule=my-domain.com/path=my-service:8080`:
- Output: The `my-ingress` Ingress is created.
- Explanation: Creates an Ingress called `my-ingress` that routes traffic from `my-domain.com/path` to the `my-service` service on port 8080.

27. `kubectl get ingresses`:
- Output: A list of ingresses in the default namespace.
- Explanation: Lists all the ingresses available in the default namespace.

28. `kubectl describe ingress my-ingress`:
- Output: Detailed information about the `my-ingress` Ingress.

- Explanation: Retrieves detailed information about the `my-ingress` Ingress, including its rules and backend services.

29. `kubectl delete ingress my-ingress`:
- Output: The `my-ingress` Ingress is deleted.
- Explanation: Deletes the `my-ingress` Ingress.

30. `kubectl create serviceaccount my-serviceaccount`:
- Output: The `my-serviceaccount` ServiceAccount is created.
- Explanation: Creates a ServiceAccount called `my-serviceaccount`.

31. `kubectl get serviceaccounts`:
- Output: A list of service accounts in the default namespace.
- Explanation: Lists all the service accounts available in the default namespace.

32. `kubectl describe serviceaccount my-serviceaccount`:
- Output: Detailed information about the `my-serviceaccount` ServiceAccount.
- Explanation: Retrieves

detailed information about the `my-serviceaccount` ServiceAccount, including its associated tokens.

33. `kubectl delete serviceaccount my-serviceaccount`:
- Output: The `my-serviceaccount` ServiceAccount is deleted.
- Explanation: Deletes the `my-serviceaccount` ServiceAccount.

34. `kubectl create role my-role --verb=get,list --resource=pods,pods/log`:
- Output: The `my-role` Role is created.
- Explanation: Creates a Role called `my-role` with permissions to perform the `get` and `list` operations on pods and pod logs.

35. `kubectl get roles`:
- Output: A list of roles in the default namespace.
- Explanation: Lists all the roles available in the default namespace.

36. `kubectl describe role my-role`:
- Output: Detailed information about the `my-role` Role.
- Explanation: Retrieves detailed information about the `my-role` Role, including its rules and permissions.

37. `kubectl delete role my-role`:
   - Output: The `my-role` Role is deleted.
   - Explanation: Deletes the `my-role` Role.

38. `kubectl create rolebinding my-rolebinding --role=my-role --serviceaccount=my-namespace:my-serviceaccount`:
   - Output: The `my-rolebinding` RoleBinding is created.
   - Explanation: Creates a RoleBinding called `my-rolebinding` that binds the `my-role` Role to the `my-serviceaccount` ServiceAccount in the `my-namespace` namespace.

39. `kubectl get rolebindings`:
   - Output: A list of role bindings in the default namespace.
   - Explanation: Lists all the role bindings available in the default namespace.

40. `kubectl describe rolebinding my-rolebinding`:
   - Output: Detailed information about the `my-rolebinding` RoleBinding.
   - Explanation: Retrieves detailed information about the `my-rolebinding` RoleBinding, including its role and subjects.

41. `kubectl delete rolebinding my-rolebinding`:
   - Output: The `my-rolebinding` RoleBinding is deleted.
   - Explanation: Deletes the `my-rolebinding` RoleBinding.

42. `kubectl create namespace my-namespace`:
   - Output: The `my-namespace` namespace is created.
   - Explanation: Creates a new namespace called `my-namespace`.

43. `kubectl create deployment my-deployment --image=my-image:v1 --namespace=my-namespace`:
   - Output: The `my-deployment` deployment is created in the `my-namespace` namespace.
   - Explanation: Creates a deployment called `my-deployment` in the `my-namespace` namespace using the `my-image:v1` container image.

44. `kubectl scale deployment my-deployment --replicas=3 --namespace=my-namespace`:
   - Output: The `my-deployment` deployment in the `my-namespace` namespace is scaled to 3 replicas.

- Explanation: Adjusts the number of replicas for the `my-deployment` deployment in the `my-namespace` namespace to 3.

45. `kubectl get deployments --namespace=my-namespace`:
- Output: A list of deployments in the `my-namespace` namespace.
- Explanation: Lists all the deployments available in the `my-namespace` namespace.

46. `kubectl get pods --field-selector=status.phase=Running --namespace=my-namespace`:
- Output: A list of running pods in the `my-namespace` namespace.
- Explanation: Lists all the pods in the `my-namespace` namespace that are in the running phase.

47. `kubectl rollout status deployment/my-deployment --namespace=my-namespace`:

- Output: The rollout status of the `my-deployment` deployment in the `my-namespace` namespace.
- Explanation: Checks and displays the status of the rollout process for the `my-deployment` deployment in the `my-namespace` namespace.

48. `kubectl set image deployment/my-deployment my-container=my-image:v2 --namespace=my-namespace`:
- Output: The image for the `my-container` container in the `my-deployment` deployment is updated to `my-image:v2`.
- Explanation: Updates the image of the `my-container` container in the `my-deployment` deployment in the `my-namespace` namespace to `my-image:v2`.

49. `kubectl delete deployment my-deployment --namespace=my-namespace`:
- Output: The `my-deployment` deployment in the `my-namespace` namespace is deleted.
- Explanation: Deletes the `my-deployment` deployment in the `my-namespace` namespace, terminating its execution and removing all associated resources.

50. `kubectl delete namespace my-namespace`:
- Output: The `my-namespace` namespace is deleted.
- Explanation: Deletes the `my-namespace` namespace and all its associated resources, including deployments, pods, and services.