Domain 1: Orchestration (25% of exam)

Complete the setup of a swarm mode cluster, with managers and worker nodes

Note: A swarm is a cluster of docker nodes that includes manager and worker nodes, on which containers can be deployed.

1. Ports used by docker swarm:

   ```
   TCP 2377/2376 for cluster management communications
   UDP 4789 for overlay network traffic
   TCP/UDP 7946 for communication among nodes

   The following commands will whitelist the docker swarm ports in firewalld:

   # firewall-cmd --add-port=2376/tcp --permanent
   # firewall-cmd --add-port=2377/tcp --permanent
   # firewall-cmd --add-port=7946/tcp --permanent
   # firewall-cmd --add-port=7946/udp --permanent
   # firewall-cmd --add-port=4789/udp --permanent
   ```

2. New multi node swarm (for deploying multiple containers over multiple nodes):

   ssh to a machine that has docker, that you want to run as swarm manager, and run the following command.

   ```
   # docker swarm init --advertise-addr <IP-ADDRESS-OF-MANAGER>
   ```

   Note: this command creates a swarm and provides a swarm token, which can be used to add worker nodes to swarm.

3. Autolock a swarm.

   ```
   # docker swarm init --autolock --advertise-addr <IP-ADDRESS-OF-MANAGER>
   ```

   Note: If autolock is enabled, when the docker daemon on a swarm manager is restarted, the swarm must be unlocked first using the swarm unlock key. This command will generate a key which will be later needed to unlock swarm.

4. Unlock a swarm

   ```
   # docker swarm unlock
   ```

   Note: This command will prompt for the unlock key.

5. New single node swarm (for deploying multiple containers on a single node):

   ```
   # docker swarm init
   ```

   Note: This command provides a swarm token, which can be used to add worker nodes to swarm.

6. Check status of swarm:

   ```
   # docker info
   ```

7. List all nodes in a swarm:

   ```
   # docker node ls
   ```

   Note: In the output, * next to *node id* indicates you are currently connected to that node.

8. Display join token (used to add a new worker to swarm)

   Run the command on a manager node.

```
# docker swarm join-token worker
```
Note: this command provides a swarm token, which can be used to add worker nodes to swarm.

9. Display join token (used to add a new worker to swarm)
   Run the command on manager node.
   ```
   # docker swarm join-token manager
   ```
   Note: this command provides a swarm token, which can be used to add manager nodes to swarm.

10. Join a new worker or manager to existing swarm.
    Run the command on worker node.
    ```
    # docker swarm join --token <manager OR worker token> <manager-ip>:2377
    ```
    Note: worker nodes are added to swarm to increase swarm capacity, to handle tasks.
    Whereas, Manager nodes are added to increase fault tolerance. The first manager node remains the leader and it is responsible for orchestration and cluster management. When the leader goes down, new manager is elected as leader to maintain swarm state. Manager nodes can also act as worker nodes to run tasks.

11. Promote a worker to manager.
    Run the command on manager node.
    ```
    # docker node promote <worker_nodename>
    ```

12. Demote a manager to worker.
    Run the command on manager node.
    ```
    # docker node demote <worker_nodename>
    ```

13. Remove a node from swarm.
    ```
    # docker swarm leave
    ```
    Note: to remove a manager node from swarm, demote the manager to worker and run same command. To remove manager directly, add '--force' to the same command. This however will disrupt the quorum.

14. Drain a node.
    ```
    # docker node update --availability drain <nodename>
    ```
    Note: drained node can no longer accept tasks. Existing tasks will be reassigned.

15. Activate a node.
    ```
    # docker node update --availability active <nodename>
    ```

16. Delete a node from swarm.
    ```
    # docker node rm <nodename>
    ```

17. Some interesting notes on swarm.
    When **'docker swarm join'** command is run on a node,
    - Docker Engine on the node switches to swarm mode.
    - requests a TLS certificate from the manager.
    - names the node with the machine hostname
    - joins the node to the swarm at the manager listen address based upon the swarm token.
    - sets the current node to Active availability, meaning it can receive tasks from the scheduler.
    - Swarm Allows specifying an overlay network for services that was manually created earlier, or creates its own.
    - extends the ingress overlay network to the current node. (inbound communication)
    - Swarm has its own DNS server for service discovery
    - Swarm does ingress load balancing itself, but also allows external load balancer.

- Communication between swarm nodes are secured using TLS mutual authentication and encryption
- Allows usage of self-signed root certificate or certificate from root CA
- Swarm supports applying updates and rollbacks.
- A swarm node also allows to run standalone containers along with swarm service.

Describe and demonstrate how to extend the instructions to run individual containers into running services under swarm

Note: Every swarm node has a load balancer which can relay requests to every instance running across itself and other swarm nodes using a routing mesh (network for ingress communication between nodes). This is what makes deployment of services over multiple nodes and load balancing between them possible in a swarm.

1. Deploy a service on a swarm, with 2 replicas
```
# docker service create --replicas 2 -p 1213:80 --name webservice
nginx:1.16.1
```
Note: Creates a docker service named *webservice* running nginx version 1.16.1, with 2 replicas (also known as instances) & container port 80 mapped to host port 1213.

cli reference – docker service create

2. Deploy a service on a swarm, in global mode
```
# docker service create -p 1213:80 --name webservice --mode global
nginx:1.16.1
```
Note: Creates a docker service named *webservice* running nginx version 1.16.1, container port 80 mapped to host port 1213. With '`--mode global`' service is placed on every available node including the new nodes that become available in future.

3. Deploy a service on a swarm
```
# docker service create --replicas 2 -p 1213:80 --name webservice
nginx:1.16.1
```
Note: Creates a docker service named *webservice* running nginx version 1.16.1, with 2 replicas & docker host port 1213 mapped to container port 80. Instead of '`-p 1213:80`', one can also use '`--publish published=1213,target=80`'. It's good to remember that, port on the right is for container.

4. List running services
```
# docker service ls
```

5. List a service
```
# docker service ps webservice
```
Note: lists a service and its replicas, along with its state and the node on which it is running etc.

6. Inspect a service
```
# docker service inspect webservice --pretty
```
Note: displays all details of a service

7. Scale service
```
# docker service scale webservice=5
```
Note: Changes the number of replicas of *webservice* to 5

8. Update service with newer version

```
# docker service update --image nginx:1.17.4 webservice
```

Note: Updates nginx version to 1.17.4 for webservice. Update mechanism is run as per *UpdateConfig* policy, which is a configurable parameter. Current configuration can be found using 'docker service inspect' command

9. Delete a service

```
# docker service rm webservice
```

Describe the importance of quorum in a swarm cluster.

Note: Multiple manager nodes in swarm need to have information and state of swarm synced in between them. Every time a task (such as node addition, removal, service creation etc.) is initiated by one of the manager nodes, this state change is proposed to the other manager nodes, by means of log replication.

'Quorum' in docker is the minimum number of manager nodes that need to be in 'available' state. so that they can replicate the information among them and agree for scheduling the proposed task. When the number of manager nodes are less than required, tasks can no longer be scheduled (however, existing tasks continue to run on worker nodes). This minimum number of manager nodes is calculated using (N/2)+1, N being the number of manager nodes. Example, in a swarm containing 13 managers, minimum of 7 managers need to be available to maintain quorum (i.e more than half the number of total manager nodes).

This way of requiring manager nodes to agree on proposed state changes before it gets scheduled, ensures all managers are storing the same consistent state. This is implemented by using 'Raft consensus algorithm'. The algorithm ensures, one of the manager nodes acts as a leader and logs all changes taking place in the cluster. It also replicates those logs to every other available manager node. In case this manager node goes down, another manager node is elected as leader to resume this operation. This way swarm exhibits the characteristics of a 'distributed system' such as,

- Agreement on values in a fault tolerant system, mutual exclusion through the leader election process
- cluster membership management.
- globally consistent object sequencing and CAS (compare-and-swap) primitives.

Describe the difference between running a container and running a service.

Note: Below are the key differences between a container and a service

Running as a container:

- A container is a standalone instance that can be built and run from a docker image.
- A container has its own filesystem and processes running in it but uses the host OS's kernel and runs in user space of the host.

- Changes require container to be restarted manually.

Running as a service:

- A docker service allows running multiple containers (also known as tasks) as replicas over multiple nodes.
- Docker service is deployed on a swarm (a cluster of worker and manager nodes).
- The tasks run independently of each other.
- Swarm allows to modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service
- manager node in a swarm contains the following modules.
    - API - accepts commands and creates service object
    - orchestrator - create tasks for service objects (task creation)
    - allocator - allocates IP addresses to tasks (containers)
    - dispatcher - assigns tasks to nodes
    - scheduler - instructs a worker to run a task. (Containers are spawned)

Interpret the output of "docker inspect" commands.

Note: The output of "docker inspect" or "docker service inspect" can be interpreted by understanding the following options. `# docker inspect <id-of-any-docker-object>` can be used to inspect the object.

cli reference – docker inspect
cli reference – docker service inspect

Convert an application deployment into a stack file using a YAML compose file with "docker stack deploy"

Note: docker stack deploy uses a docker-compose.yml which can be used to configure multiple services in swarm mode. (`docker-compose` also uses a docker-compose.yml file to bring up multiple services, but it cannot have replicas running in swarm mode.) Depending on which one is used, some parameters are parsed or ignored in docker-compose.yml.

For example, build option in docker-compose.yml is used to build an image from a Dockerfile. This is supported by docker-compose only This will be ignored by '`docker stack deploy`' command. It does not support building of an image from a docker file. It will expect the image to be already present. . In this scenario, image can be built before deployment using '`docker image build -t <imageName>:version <path/to/dir/containing/Dockerfile>`'

1. Deploy a new stack or update an existing stack

```
# docker stack deploy --compose-file docker-compose.yml MyStack
```

cli reference – docker stack deploy

2. List stacks
```
# docker stack ls
```

cli reference – docker stack ls

3. List tasks in stack
```
# docker stack ps MyStack
```

cli reference – docker stack ps

4. Remove a stack
```
# docker stack rm MyStack
```

cli reference – docker stack rm

5. List Services running in stack.
```
# docker stack services MyStack
```

cli reference – docker stack rm

Describe and demonstrate orchestration activities

Note: Docker supports two orchestrators namely swarm and Kubernetes. Orchestrators help in managing lifecycle of a container such as deployment, scaling, networking etc.

We have previously discussed managing services using swarm.
This section does not cover or explain Kubernetes orchestrator. However to get a gist, below are some of the basic Kubernetes commands for deploying an application and checking its status.

1. Deploy with Kubernetes using, Kubernetes yaml file
```
# kubectl apply --filename MyApp.yaml
```

2. List all Kubernetes pods
```
# kubectl get pods
```

3. List Kubernetes deployments
```
# kubectl get deployments
```

4. List Kubernetes services
```
# kubectl get services
```

Increase number of replicas

Note: The command to increase or decrease replicas is the same.

1. Examples to change the number of replicas for services.
```
# docker service scale myservice=4 anotherservice=5
# docker service update --replicas 50 myservice
```

Note: the above commands can be used to increase or decrease the number of replicas. Both commands can be used alternatively. The second command does not allow scaling multiple services at once like the first one.

Add networks, publish ports

Note: Network is a pluggable component in docker, which can be created and attached to a container.

1. Create a network

```
# docker network create --driver drivertype MyNetwork
```

Note: Docker supports different network drivers. Drivers can be chosen depending on the requirement and the type of deployment.
drivertype can be any of the following types of network as arguments.

cli reference – docker network create

- *bridge* – This is the default network type for containers. 'bridge' is also the name of the default network created by default for communication between containers of same node, and 'docker_gwbridge' for a docker swarm. 'docker_gwbridge' is for communication docker hosts participating in the swarm. 'docker_gwbridge' is created automatically when swarm is initialized or when a node is joined to swarm. The default bridge network does not provide name resolution.

  User-defined Bridge network:

  ```
  # docker network create --driver bridge my-custom-bridge-network
  ```

  The features are same as the default bridge network, however, when a custom network like this is created and used, DNS between containers.

- *host* – Used when container shares the same network as host and port range. (Can be used only with swarm services). When using this network, port mapping options will be ignored. The limitation here is that only once service container can bind to a specific host port in a docker node. Multiple containers cannot bind to same host port. Host Networking driver is useful when NAT is not preferred and many ports are needed for container. Docker communication between containers in swarm does not use this network. Only data traffic between containers uses this.

- *overlay* – This is used for communication between swarm services across different nodes. When a swarm service is created, a bridge network and an overlay network is created at the same time to implement this communication.

- *macvlan* – Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

- *none* – For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.

Note: The following argument is passed to '`docker service create`'

1. Some examples for publishing ports

TCP only:

```
‘-p 1213:80/tcp’ OR
‘--publish published=1213,target=80,protocol=tcp’
```

UDP only:

```
‘-p 1213:80/tcp’ OR
‘--publish published=1213,target=80,protocol=udp’
```

TCP and UDP:

```
‘-p 1213:80/tcp -p 1213:80/udp’ OR
‘--publish published=1213,target=80,protocol=tcp --publish
published=1213,target=80,protocol=udp’
```

```
Default protocol is TCP, if no protocol is specified.

Published port can be left empty, in which case swarm manager assigns one
in range 30000-32767
```

2. Publish port in host mode.

```
# docker service create --name webservice--publish
published=1213,target=80,protocol=tcp,mode=host --mode global
```

Note: ‘mode=host’ ensures that any request coming to 1213 is forwarded to port 80 of the instance running on that node only. If the node stops running the service, the request is not forwarded to a different node in swarm. Also, along with this ‘--mode global’ should be used else it is difficult to know which nodes are running the service to route work to them.

3. Publish new port on an existing service.

```
# docker service update --publish-add published=1145,target=8080 webservice
```

Mount volumes

Note: Docker provides three types of mounts namely bind mounts, volumes, tmpfs (linux only).

Mount can be specified using -v (--volume) or --mount.
--mount is supported by both standalone containers and swarm services
-v (--volume) is a legacy option available only for containers and it does not support a lot of parameters that is supported by --mount, hence –mount is always recommended.

Bind mount:

- Bind mount: In host machine, any file or directory located in any filesystem can be mounted in docker container.
- Data can be shared between containers
- Data persists even after container is stopped
- Examples (mount in standalone containers):
  - Using -v (--volume)

    ```
    # docker run -v /path/in/host:/path/in/container -d nginx
    ```

    Following command does same as the previous one, but container mount point path is created similar to host.

    ```
    # docker run -v /path/in/host -d nginx
    ```

  - Using --mount

```
# docker run --mount
type=bind,source=/path/in/host,target=/path/in/container,mountoptions -d
nginx
```

- Examples (mount in swam services, -v or --volume is not supported, only --mount is supported):

```
# docker service create --name myservice --mount
type=bind,source=/path/in/host,target=/path/in/container,mountoptions
nginx
```

Volume:

- Volume: One can only choose a volume name. Docker creates a directory for the volume on host machine under /var/lib/docker/volumes/ and mounts
- Data can be shared between containers
- Data persists even after container is stopped
- Examples (mount in standalone containers):
  - Using -v (--volume)

```
docker run -v volumename:/path/in/container -d nginx
```

  - Using --mount

```
docker run --mount
source=volumename,target=/path/in/container,mountoptions -d nginx
```

- Examples (mount in swam services, -v or --volume is not supported, only --mount is supported):

```
# docker service create --name myservice --mount
source=volumename,target=/path/in/container,mountoptions nginx
```

Recommended read: Troubleshooting errors

tmpfs:

- tmpfs: Data is stored in host machine's memory.
- Data cannot be shared between containers
- Examples (mount in standalone container):
  - `docker run --tmpfs /mytmp -d nginx`
- Examples (mount in swam services, -v or --volume is not supported, only --mount is supported)

```
docker service create --name myservice --mount type=tmpfs,target=/app nginx
```

More on *volume* mount:

1. Create a volume

```
# docker volume create MyVolume
```

cli reference – docker volume create

2. Display detailed information on one or more volumes

```
# docker volume inspect MyVolume
```

cli reference – docker volume inspect

3. List volumes
```
# docker volume ls
```

cli reference – docker volume ls

4. Remove all unused local volumes
```
# docker volume prune
```

cli reference – docker volume prune

5. Remove one or more volumes
```
# docker volume rm MyVolume
```

cli reference – docker volume rm

Describe and demonstrate how to run replicated and global services

Note: *Replicated* services is to have one or more replicas of the container service running across the nodes, whereas *Global* refers to running a service on every available node (including new nodes in the future).

1. Deploy a service on a swarm, with 2 replicas
```
# docker service create --replicas 2 -p 1213:80 --name webservice
nginx:1.16.1
```
Note: Creates a docker service named *webservice* running nginx version 1.16.1, with 2 replicas (also known as instances) & container port 1213 mapped to host port 80.

cli reference – docker service create

2. Deploy a service on a swarm, in global mode
```
# docker service create -p 1213:80 --name webservice --mode global
nginx:1.16.1
```
Note: Creates a docker service named *webservice* running nginx version 1.16.1, container port 1213 mapped to host port 80. With '`--mode global`' service is placed on every available node including the new nodes that become available in future.

Apply node labels to demonstrate placement of tasks

Note: When swam services are started, one can control on which nodes a service can be placed. This can be controlled by two options '`--constraint`' and '`--placement-pref`'

labels can be updated on nodes, which can be used to decide whether to place a service on that node.

1. Add label to node
```
# docker node update --label-add environment=test MyNode
```
Note: environment=test is a key value pair that is updated as metadata on MyNode

cli reference – docker node update

2. Example: Create a service with constraint
```
# docker service create --name web –replicas 3 --constraint
node.labels.environment==test nginx
```
Note: Service is placed only on nodes that satisfies label 'environment==test'. If none of the node satisfies this, deployment of service will fail.

3. Example: Create a service with a placement preference.
```
# docker service create --name web --placement-pref
"spread=node.labels.environment" nginx
```
Note: Service is placed on nodes that has an environment key label. Spread is an algorithm that tries to evenly distribute service amount nodes considering preference and other constraints requested. Currently 'spread' is the only option available. In case the no nodes are available to satisfy the preference, it still proceeds to place tasks on nodes (unlike a constraint, where in it would fail it does not satisfies).

Create services using templates

Note: Templates can be compared to pre-defined inbuilt variables which's value can be substituted and used as a template while creating services. A template is enclosed with double curly braces. For example, one can set the hostname of the container to contain the name of the service or task or node etc, by using the respective template for service or task or the node respectively, and the value is populated accordingly. Templates are supported by '--hostname', '--env', '--mount' flags only

1. Create a service to demonstrate usage of templates.
```
# docker service create --name web --hostname="{{.Node.Hostname}}-
{{.Node.ID}}-{{.Service.Name}}" --env "myTask={{.Task.Name}}" --mount
“source=myvolume,target=/path/in/container/{{.Task.Slot}}” nginx
```

Recommended read: List of available templates

Identify the steps needed to troubleshoot a service not deploying

Note: The following commands can be used to troubleshoot docker services.

1. check if service is initiated and listed
```
# docker service ls
```

2. check the state of service and which nodes are running it
```
# docker service ps service_name
```

3. Verify configuration for service is as expected for the service using inspect.
```
# docker service inspect service_name
```

4. Check logs for service
```
# docker service logs service_name
```

Additionally, following are the common solutions to consider (use appropriately)

1. Restart a container
```
# docker container stop container_name
# docker container start container_name
```

2. Restart a service.
```
# docker service scale myservice=0
# docker service scale myservice=1
```
Note: Service cannot be restarted directly. It can only be removed and recreated. However, by scaling the service to 0 and back does the job of restart.

3. Unjoin and join a node from swarm
```
# docker swarm leave
# docker swarm join --token <manager OR worker token> <manager-ip>:2377
```

4. Restart docker daemon on host
```
# systemctl restart docker
```

Describe how a Dockerized application communicates with legacy systems

Note: Regardless of the type of network and system it connects to, container uses the network interface, IP address, subnet mask and gateway of the network and DNS to communicate.

- Each docker network has a subnet mask and gateway
- Docker daemon acts as DHCP server and assigns a ip to the container from the pool, when container is started.
- A container can be started with a single network, but additional networks can be added to it after it is started using using '' docker network connect network-name container-name"
  cli reference – docker network connect
- Containers using the default network (bridge), inherits /etc/resolv.conf from host
- Containers using a custom network will forward the external DNS lookup to host
- DNS can be customized in container's /etc/resolv.conf by using the below flags while creating a container or service.
  `--dns` Custom DNS server
  `--dns-option` Set DNS options (DNS Options are as in options officially supported by resolv.conf)
  `--dns-search` Set custom DNS search domains

Recommended read: Add an entry to /etc/host in container

Describe how to deploy containerized workloads as Kubernetes pods and deployments

Note: Understanding the following is essential to be able to use Kubernetes to handle deployments.

- *Kubernetes cluster* - A set of nodes containing *Kubernetes master server(s)* and *Kubernetes nodes*
- *Kubernetes Pods* - Used to run a single set of one or more containers on same node. Example, an application pod can contain a db-container and a web-container. Practically pods are not directly deployed in production. Instead a deployment is used

  Recommended read: Kubernetes Pods
- *Kubernetes Deployments* - Used to run replicas of pods (scaling) on one or more nodes and to maintain its configuration and states.

  Recommended read: Kubernetes Deployments
- *Kubernetes Services - Provides networking in a cluster*

  Recommended read: Kubernetes Services
- *Kubernetes objects* – Any persistent entity that can be defined in a Kubernetes YAML file is called a Kubernetes object. Thus Pods, Deployments, Services are Kubernetes objects that can be defined in a Kubernetes YAML file and created by running using 'kubectl apply -f <path-to-YAML-file>. The configuration to use for deploying workloads is provided in this YAML file.

  The YAML file requires the following fields to be specified for deployment.

  1. *apiVersion* - Kubernetes API version to use
  2. *kind* - (Pods or Deployments or Service)
  3. *metadata* - Metadata such as name, UID, and labels
  4. *spec* - Image to run, no. of replicas, port mapping etc

     Recommended read: Kubernetes YAML file

Describe how to provide configuration to Kubernetes pods using ConfigMaps and secrets

Note: ConfigMaps and secrets are two different ways of injecting information into a container when the container is started. The difference between them is that secrets are base64 encrypted whereas ConfigMaps are not. Secrets and ConfigMaps can be used in pods as data volumes and environment variables.

Secrets & ConfigMaps can be created using 'kubectl create secret' and 'kubectl create configmap' commands respectively.

Recommended read:

Creating a secret manually
Creating a ConfigMap manually

Like Pods, Secrets and ConfigMaps are also objects and therefore can be defined in a YAML file and created by using 'kubectl apply -f <path-to-YAML-file>

Secrets and ConfigMaps can be created from various data sources.

Recommended Read: Creating and using Secrets

- Generating a Secret from string literals
- Generating a Secret from files
- Decoding a Secret
- Editing a Secret
- Using Secret(s)

Recommended Read: Creating and using ConfigMaps

- Create ConfigMaps from directories
- Create ConfigMaps from files
- Create ConfigMaps from literal values
- Using ConfigMap(s)

Recommended Read: Creating Secrets and ConfigMaps using *Kustomize* tool