Domain 3: Installation and Configuration (15% of exam)

Demonstrate the ability to upgrade the Docker engine
Complete setup of repo, select a storage driver, and complete installation of Docker engine on multiple platforms

Recommendations for installing/upgrading/uninstalling docker engine:

1. remove older packages that provided docker engine (docker, docker-engine, docker.io, containerd, runc). "`docker-ce`" along with "`docker-ce-cli`", "`containerd.io`" can be installed. The contents of `/var/lib/docker` are preserved.
2. Storage drivers supported by docker engine: `overlay2 (default), aufs, btrfs`.
3. Installation methods:
   - setup docker repositories and install from them using yum or apt-get etc (*recommended*).
   - Packages can also be downloaded and installed manually using yum or apt-get etc. This is useful on air-gapped systems
   - Dev/Testing environments can use convenience script "`get-docker.sh`" from docker portal and run it for automated installation. This script does not perform upgrades. If an upgrade is needed, packages need to be upgraded using package managers.
4. If non-root user needs to run docker, add the user to "`docker`" group.
5. Enable and start the service.
6. Test installation by running "`docker run hello-world`"
7. Uninstall needs to be done using package managers. Contents such as images and containers is stored under `/var/lib/docker`, `/var/lib/containerd` needs to removed manually if no longer needed.

Configure logging drivers (splunk, journald, etc)

Logging drivers can be configured to extract information about running containers and services. Docker has built in plugin drivers. Other plugins are available in docker hub in the form of images, which can be installed using,

```
# docker plugin install <image>
# docker plugin ls
```
NOTE: list available plugins

The current logging driver in use can be found by running "`docker info --format '{{.LoggingDriver}}'`". Different log drivers have different options that can be configured. A log driver and its corresponding options can be configured in /etc/docker/daemon.json. Example,

```
{
 "log-driver": "json-file",
 "log-opts": {
  "max-size": "10m",
  "max-file": "3",
```

```
    "labels": "production_status",
    "env": "os,customer"
   }
}
# numbers must be quoted and be presented as strings.
```

Different Docker drivers can be used for different containers/services by setting the value `to --log-driver` flag. Default logging driver used by docker is "json-file". If no logging driver is specified this is used. The logging used by a specific container or service can be found by running "`docker inspect <container or service>`" command and checking the corresponding section in output.

Log message delivery modes (decides how delivery of messages is handled from container to log driver). The "`mode`" parameter can be set in "`log-opts`" section for the log-driver in `/etc/docker/daemon.json,` or it can also be set at container level during "`docker run`" by passing the `–log-opt mode=<blocking OR non-blocking>` flag.

- *Blocking mode*, also known as direct mode: This is the default mode. In this mode application is interrupted when a message needs to be delivered to the logging driver, until it is delivered. This may cause latency in the container application in cases where logging needs to happen to a remote server instead of local file system or if logging driver is busy etc.

- *Non-blocking mode*, also known as delivery mode: In this mode log messages are stored in an intermediate per-container in memory ring buffer for consumption by driver. The log hand off is completed immediately and application task is resumed. However, if application is emitting logs frequently the log buffer might run of space causing its deletion even before the messages are consumed by the driver. `max-buffer-size` (default value 1MB) can be increased or decreased in `/etc/docker/daemon.json.`

NOTE: Some logging drivers supports logging of environmental variables and labels in the output when displaying details of a container in log output's "`attrs`" section.

Example, "attrs":{"production_status":"testing","os":"ubuntu"}

```
Commands that fetches logs in docker:
# docker logs <container-name> OR
# docker container logs <container-name>
# docker service logs <service-name>
```

- Regardless of the logging driver used, the above commands should provide logs in the recent versions of docker.
- Dual logging is supported only Docker Enterprise and is enabled by default. This does not mean two logging drivers can be configured. This logging driver configured in docker will work along with the local logging.
- "`tag`" option passed to `--log-opt` is used to set the tag for log messages (at the beginning of every log entry after the date, time & hostname, the tag specified is also appended. The tag can be one of the many available inbuilt templates such as

container's short id `{{.ID}}`, full id `{{.FullID}}`, name of image `{{.ImageName}}` etc. Container's short ID is default, where in it prints the container ID's first 12 characters. NOTE: `{{.Name}}` represents the container's name. This can also be used. However, if "docker rename" OR "docker container rename" command is used to rename the container, the tag will not update itself. It will still log with old name.

Setup swarm, configure managers, add nodes, and setup backup schedule

NOTE: Detailed notes can be found on docker swarm in "Docker Certified Associate - Part1 – Orchestration", which has a document of its own. This document is available as a separate download.

The following points walks us through some of the key points and best practices with respect to managing a docker swarm.

1. Adding more manager nodes increase fault tolerance

2. Too many manager nodes however can cause reduced IO performance and increased network usage due to more log replication.

3. Odd number of managers are recommended to maintain Quorum.
4. A fixed or static IP should be used as advertised IP in swarm. Dynamic IP is OK for worker nodes.

5. If the number of manager nodes is down to 1, it is not possible to demote that manager to worker.

6. Distribution of manager nodes across a minimum of 3 availability zones is recommended, for optimal fault tolerance.

7. Draining a manager node's availability (`--availability drain`) will ensure it does not accept worker tasks and existing tasks are reassigned to other nodes. (`--availability active`) sets it back to accept worker tasks. (`--availability pause`) makes the worker unavailable for new tasks but existing tasks continue to run.

8. Ensure enough worker nodes are available for load balancing. Ensure node labels are updated as needed in case the service has `node constraints` set.

9. "docker node inspect <nodename>" provides the health status of node. Worker nodes reveal a section named `"Status"` in which it indicates weather the worker's `ready status`. Manager nodes reveal a section named `"ManagerStatus"` in which it indicates if the manager 's `reachability status` . If a node functions as both manager and worker, then both the sections are outputted.

10. If a manager is unreachable for unknown reasons:
    - Restart daemon and see if it becomes available.

- If not, restart node.
- If not, demote node as worker and remove the node from swarm.

11. Backup swarm regularly. Swarm stores, swarm states and logs in `/var/lib/docker/swarm`. This location also stores the keys used to encrypt the logs. This is essential to restore the swarm from backup. The backup can be done from any one of the manager nodes. Backup and restore can be done by following these steps,

    - In case swam `auto-lock` is enabled, ensure that you have the correct `unlock key`.
    - Stopping the docker daemon on manager node is recommended as opposed to hot backup, since data does not change when backup is done. Backup is done by copying the contents of /var/lib/docker/swarm. Note that when docker is stopped in a manager node for backup, other nodes continue to function and generate logs, which will not be a part of this backup.
    - Restart the manager after backup.

12. To recover from disaster using backup, following these steps
    - Stop the docker daemon on manager node and replace contents of /var/lib/docker/swarm from backup
    - Unlock the swarm using swarm unlock key, if auto lock was enabled
    - Re-initialize the swarm using "docker swarm init --force-new-cluster"
    - Rotate the unlock key if auto-lock is used.
    - At this point all running tasks are intact and worker nodes are still a part of swarm, but other manager nodes need to be re added to achieve quorum.

13. When swarm is reinitialized, and new nodes are added OR when new manager nodes are generally added existing tasks are not redistributed. However, a service can be forced to have its tasks rebalanced by running "docker service update --force". However, this restarts service. Another option used in to rebalance is to upscale and downcale back using "docker service scale" which rebalances the tasks. Again this will also disrupts running tasks.

Interpret errors to troubleshoot installation issues without assistance

The following can be used as guidelines for troubleshooting docker daemon.
Docker daemon config location: /etc/docker/daemon.json in linux and c:\ProgramData\docker\config\daemon.json. dockerd is the binary that is responsible for running docker daemon. dockerd is generally launched on system start up using a system utility like systemd, 'dockerd' can also be run manually by passing config in the form of command flags OR the default way is to have the daemon.json in which config is specified, as in an actual setup.

1. Set "debug" to "true" in /etc/docker/daemon.json OR start dockerd with -D flag if starting manually.

2. Running "sudo kill -SIGUSR1 $(pidof dockerd)" will force stack trace to be sent to daemon logs, wherein all threads will be logged.
3. If startup scripts are used to start dockerd and if any flags were configured to be passed, it might conflict with what is specified in daemon.json. Ensure that there are no conflicts.
4. System like utility starts dockerd by passing the -H option which is used to specify the default socket. If daemon.json has an entry for hosts, it conflicts with the start up script. To overcome this, ensure that the startup script does not pass the -H option.
5. OS Kernel will kill containers or docker daemon when containers try to consume more memory that what is available in system. To avoid such problems and out of memory exceptions, ensure there is sufficient memory available in host.
6. Read daemon logs. The daemon log location depends on the OS. Example: /var/log/messages
7. Daemon can be reloaded by sending SIGHUP to docker's PID. Example, "sudo kill -SIGHUP $(pidof dockerd)"
8. Check if docker is running by using system's default utilities such as systemd or by running "docker info" command.

Understand namespaces, cgroups, and configuration of certificates

The following walks through the underlying technologies that is docker built on.

namespaces – namespace is the technology that enables creation of isolated workspaces (called as containers). Every container runs on its own layer of isolated workspace and its access is limited to that container.

cgroups or Control Groups – cgroups enables limiting resources to an application. Docker engine uses this technology to share or limit resources such as CPU, Memory etc to containers.

Union Filesystem or UFS – Filesystems such as aufs, btrfs, devicemapper, vfs use the UFS. UFS uses layers for storing information, which is very light weight.

Note: The combination of the above three technologies namely namespaces, cgroups and UFS is called a container format called libcontainer, used by docker.

Some notes on how communication happens between docker daemon and docker client with TLS.

1. Docker communicates using an IPC Unix socket or optionally using an HTTP socket.
2. If docker needs to safely communicate over network for two way communication, TLS authentication can be configured.

A. Create Root Private key
   (Now that you have a Root Private Key, this can be used to sign certificate for yourself and others. When you sign it for yourself, its called a self signed certificate)
B. Create Root Certificate using the Root Private Key created in previous step.
   (This Root Certificate is the self-signed certificate which will be presented to anyone for trust verification)
C. Now every docker host can create a Server private key and a CSR (certificate signing request). The CSR will be signed by the Root Private Key created previously. As a result, we will have a signed server certificate, for docker host. Docker clients (docker CLI users) who wish to communicate with this docker host can be provided with a client certificate generated using a client CSR. We can change the type of certificate between client and server by changing the *extendedKeyUsage* to "clientAuth" or "serverAuth" in the extfile used to sign certificate.

   Now "docker daemon" can be started with TLS by passing the CA certificate, Server certificate and Server private key.

```
Sudo docker daemon
--<Other options>
--tlsverify
  --tlscacert=<RootCACertificate.crt> \
  --tlscert=<ServerCertificate.crt> \
  --tlskey=<ServerPrivateKey.pem>  \
--<Other options>
```

   The step C can be repeated for Docker Client so that we will have a client key and client certificate. The TLS Client key, TLS Signed Client Certificate and CA certificate can be sent to docker client nodes. The client is not a docker daemon but a docker cli client that one can use to connect to remote docker daemons. The docker cli can be started like docker daemon (in the above example), except that instead of a ServerPrivateKey and ServerCertificate, it will use a ClientPrivateKey and ClientCertificate along with RootCACertificate.

   Instead of passing the above options everytime when starting docker, one can place the files in $HOME/.docker and run "export DOCKER_HOST=tcp://$HOST:2376 DOCKER_TLS_VERIFY=1" to automatically have TLS working every time

   Therefore, with the above setup docker daemon will only accept connections from clients that present a client certificate signed by the same Root CA that signed its server certificates.

Use certificate-based client-server authentication to ensure a Docker daemon has the rights to access images on a registry

Some notes on how communication happens between docker daemon and docker client with TLS.

In The section titled "Domain 2: Image Creation, Management, and Registry" which has a document of its own (This document is available as a separate download), discusses about setting up and using TLS certificates to be configured for Registry container or service. This is an extension of that section.

In case Registry is started with TLS authentication that also requires client authentication (requires users of registry to also provide a client certificate), in this case, client-certificate, client-privatekey and CA certificate can be provided to clients and be placed in /etc/docker/certs.d/myregistryhostname:registryport/

Consistently repeat steps to deploy Docker engine, UCP, and DTR on AWS and on premises in an HA config. Docker, DTR, UCP,, Docker on AWS and possibly on premises HA config

Docker EE (licenced) provides all features as in Docker CE and additionally the following.

1. UCP (universal control plane) - GUI for managing docker swarm/kubernetes and also provides interface to create and manage users/teams/access control over docker cluster objects.
2. DTR (Docker trusted registry) - GUI for managing Enterprise registry
3. Vulnerability scanning of images stored in DTR is a security feature which makes use of a vulnerability DB which will can be optionally installed in DTR.

Installing docker Enterprise Edition:

NOTE: Sizing requirements depends on the containers.
Docker EE licence requries a docker hub account(hub.docker.com). From docker hub, docker EE url can be obtained which will be used to setup repo in yum or apt-get in linux to further download and install Docker-EE.

If non root user needs to run docker, add that user to docker group
`docker version` (gives the details of docker installation)

Installing UCP:

Sizing requirements for - Manager nodes:

- Minimum: 8GB RAM, 2 CPU
- Recommended: 16GB RAM, 4 CPU

Sizing requirements for - worker nodes:

- Minimum: 4GB RAM

```
# docker image pull docker/ucp
```

```
# docker run --rm -it --name ucp -v
/var/run/docker.sock:/var/run/docker.sock docker/ucp install --host-address
<docker-host's-IP> --interactive

#note that socket is mounted here so that installation will directly interact with docker's
socket on node
# username and password is prompted for choosing. This will be used to login to UCP.
# After first login, license key from docker hub needs to be uploaded.
```

Note: the above image is just a container that deploys the actual UCP container. The installer runs a container temporarily and installs the actual UCP. The same installer container can be user to perform other maintenance operations on the actual UCP instance

Teams and access using UCP web interface (terminologies):

User - People added to be provided with access.
Team - Group of users
Organization - A group of users.
Subject - A subject can be user(s) or a team(s) or Organization(S)
Collection - Cluster objects like containers, services, nodes etc
Role - Defines the level of control a subject can have over collections.
Grant - "Subject" + "Role" + "Collections". Example: "X people or teams" can have "Full control" over "all containers".

Backup and restore of UCP:

To backup UCP, docker swarm also needs to be backed up (as discussed in earlier sections). After docker swarm backup is done, following steps is done on UCP to backup UCP's metadata such as information about teams, access control etc.

1. Get UCP instance ID.
```
# docker container run --rm --name ucp -v
/var/run/docker.sock:/var/run/docker.sock docker/ucp id
```

2. Perform back up.
```
# docker container run --log-driver none -rm --interactive --name ucp -v
/var/run/docker.sock:/var/run/docker.sock docker/ucp backup --passphrase
"backup-passwd" --id <UCP instance ID> > /path/to/my-UCP-backup.tar
# the passphrase will be needed for restoring from backup
# UCP container is stopped when backup is done. Hence there will be downtime
```

3. Uninstall running UCP first before backup, if backup needs to be restored:
```
# docker run --rm -it --name ucp -v
/var/run/docker.sock:/var/run/docker.sock docker/ucp uninstall --
interactive
```

4. Restore from backup

```
# docker run --rm -i --name ucp -v
/var/run/docker.sock:/var/run/docker.sock docker/ucp restore --passphrase
"backup-passwd" < /path/to/my-UCP-backup.tar
```

NOTE: UCP uses self-signed certificates by default, however one can upload their custom root CA, Server cert and private key instead on the certificate settings page. One can also generate a client bundle and have one downloaded to be shared to clients for secured access to UCP.

Installing DTR (Docker Trusted Registry):

Sizing requirements:

- Minimum: 16G RAM, 2 CPU, 10G Data Disk
- Recommended: 16G RAM, 4CPU, 25G-100G Data disk

```
# docker run --rm -it docker/dtr install --ucp-node <node-on-which-dtr-is-
installed> --ucp-username admin --ucp-url https://somedockerhost-IP
```

NOTE: The above command will prompt for UCP password that was setup during installation of UCP. After this, DTR can be accessed by viewing the IP address of the docker node on which DTR was installed. The credentials to login is same as UCP username and password.

Steps to Backup DTR:

1. Get the source directory of volume from the following command
   ```
   # docker container inspect <registry-container-name>
   ```
   The "source" key under the "mounts" section should provide the path to directory name containing the source of the container. Create a tar backup of that directory

   The above step will create a backup of all images. The next step is to backup the DTR meta data, which is also needed.

2. Backup DTR Metadata:
   ```
   # docker run --log-driver none --rm -i --env UCP_PASSWORD='myucppw'
   docker/dtr backup --ucp-username admin --ucp-url https://somedockerhost-IP
   --existing-replica-id > DTR-meta-data-backup.tar
   ```

3. Destroy existing replica before restoring from backup.
   ```
   # docker run -it --rm docker/dtr destroy --ucp-username admin --ucp-url
   https://my-ucp-url
   ```
   Next, untar the archive containing the contents of volume back to same location from where it was backed up.

4. Restore from Backup
   ```
   # docker run -i -rm --env UCP_PASSWPRD="myucppassword" docker/dtr restore -
   -dtr-use-default-storage --ucp-url https://my-ucp-url --ucp-username admin
   --ucp-node mydtrnodename --replica-id myreplicaid --dtr-external-url <dtr-
   url> < DTR-meta-data-backup.tar
   ```

   DTR Security features:

- From the DTR UI under security settings, one can turn ON or OFF image security scanning that will download a security DB in DTR (this can be installed online or manually using file upload).
  Once this is enabled, after an image is pushed, the tag that has been pushed will get an option under vulnerabilities section, called "start a scan". This scans for any vulnerabilities and updates status. One can also have images pushed and scanned automatically during the time it's being pushed. Manual scan OR Automatic scan upon push can be set from under settings for image scanning.

- There is an option called "immutability" for tags, which will prevent tags from being overwritten when pushed again. This option can also be toggled on the settings page.

NOTE: DTR allows using custom root CA, Server cert and private key to be uploaded on the domain/proxy settings page.