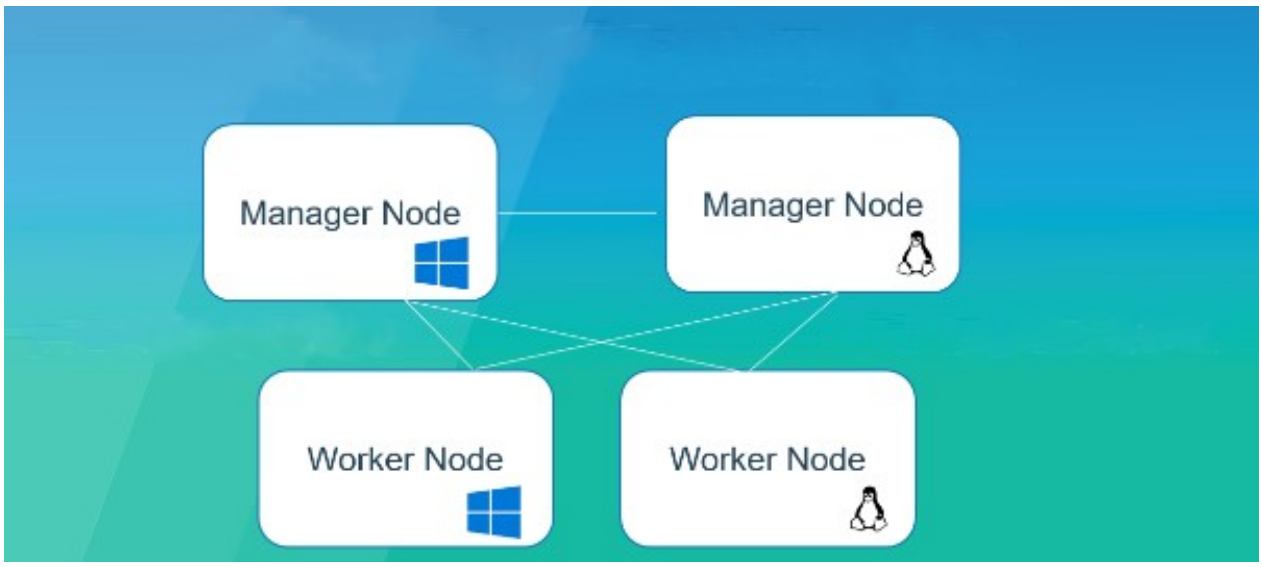
What is Docker Swarm ?

Docker Swarm is a **container orchestration platform** provided by Docker, Inc. It enables you to manage and scale containerized applications across a cluster of machines. In other words, it allows you to deploy, manage, and scale Docker containers seamlessly, providing high availability, load balancing, and ease of management for your applications.

What are the Worker Nodes and Docker Swarm Manager??

Manager nodes handle the orchestration, management, and control of a Docker Swarm, while **worker nodes execute the tasks** associated with containerized services. This division of roles and responsibilities enables Docker Swarm to efficiently distribute and manage containerized applications across a cluster of machines.



Docker Swarm Manager Nodes:

Manager nodes are the control plane of a Docker Swarm cluster. They are responsible for orchestrating and managing the cluster, making global decisions, and ensuring the desired state of services and tasks. Here's what you need to know about manager nodes:

1. **Roles and Responsibilities:**

- Managers are the entry point for managing the swarm. They handle user commands and interact with the Docker API.
- Managers maintain the global cluster state, which includes information about services, tasks, and nodes.
- They orchestrate the scheduling of tasks across worker nodes, considering factors like resource availability, placement constraints, and service configurations.
- Managers handle service scaling, updates, and rollbacks.
- They monitor the health of the swarm, including nodes and services, and take corrective actions when needed.
- Managers facilitate consensus among themselves to make decisions about the state of the swarm. They elect a leader among them to coordinate actions.

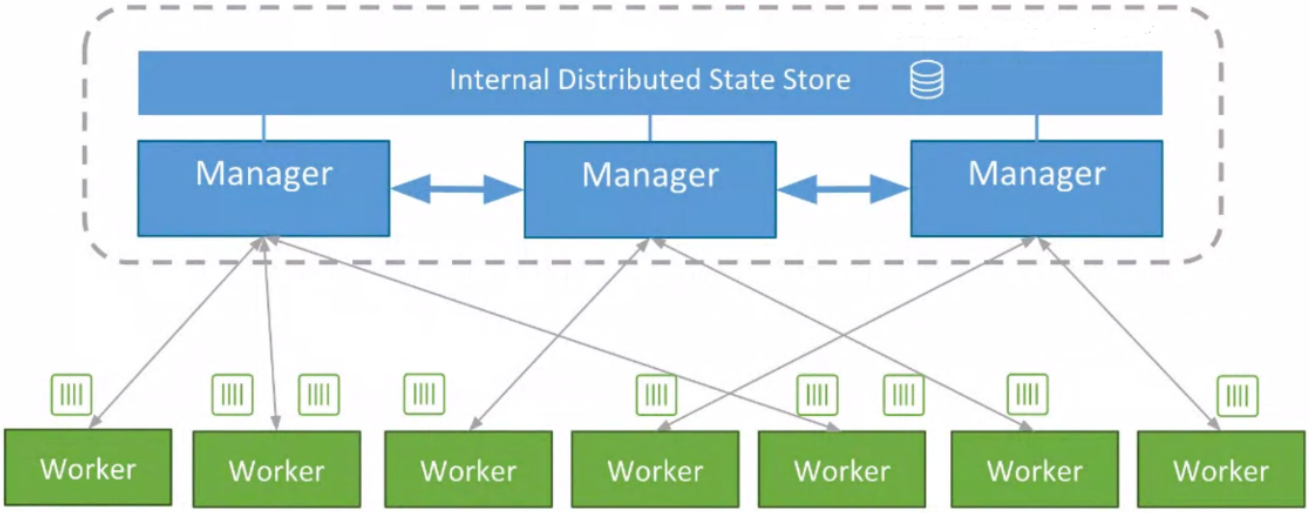
2. **High Availability:**

    - To ensure high availability, it's recommended to have multiple manager nodes. Typically, a swarm should have an odd number of managers (e.g., 1, 3, 5) to prevent split-brain scenarios.

- In case a manager node becomes unavailable, another manager takes over the leadership role.

3. **Security:**

- Manager nodes generate and manage the cryptographic keys required for secure communication within the swarm.
- They authenticate nodes that want to join the swarm.



Docker Swarm Worker Nodes:

Worker nodes are responsible for executing tasks assigned to them by manager nodes. They run the containers that make up your applications. Here's an overview of worker nodes:

1. **Roles and Responsibilities:**

- Workers execute the tasks that are scheduled by manager nodes. A task corresponds to running a container instance as part of a service.
- They report their state and resource availability to manager nodes.
- Workers don't participate in the decision-making process of the swarm; they follow the instructions provided by manager nodes.

2. **Scaling and Load Balancing:**

- Workers play a crucial role in ensuring the desired number of service replicas are running across the swarm.
- They balance the workload by distributing containers among themselves, based on the scheduling decisions made by manager nodes.

3. **Health Checks:**

    - Workers execute health checks on containers and report the results to manager nodes.
    - Manager nodes use this information to decide whether to reschedule tasks on other nodes if a container's health deteriorates.

Interactions and Collaboration:

- Manager nodes and worker nodes communicate using encrypted channels.
- Worker nodes periodically report their state and available resources to manager nodes.
- Manager nodes use information from worker nodes to make informed decisions about task scheduling and service management.

What differs Docker compose from Docker swarm ?

Choosing the Right Tool:

- **Docker compose**

Docker Compose is ideal for local development and smaller projects where you want to define and manage the configuration of your containers in a straightforward way.

- **Docker swarm**

Docker Swarm is suitable for orchestrating containers at scale across a cluster, providing features for service management, load balancing, and high availability.

Key Concepts

1. **Nodes:**

- Nodes are the individual machines that make up a Docker Swarm cluster.
- There are two types of nodes: Manager Nodes and Worker Nodes.
- Manager Nodes manage the swarm and make global decisions, while Worker Nodes execute tasks.

2. **Services:**

- A service is the fundamental unit of work in Docker Swarm.
- It defines the desired state of a containerized application, including the container image, number of replicas, network settings, and more.
- Services ensure that a specified number of replicas (containers) are running across the swarm cluster.

3. **Tasks:**

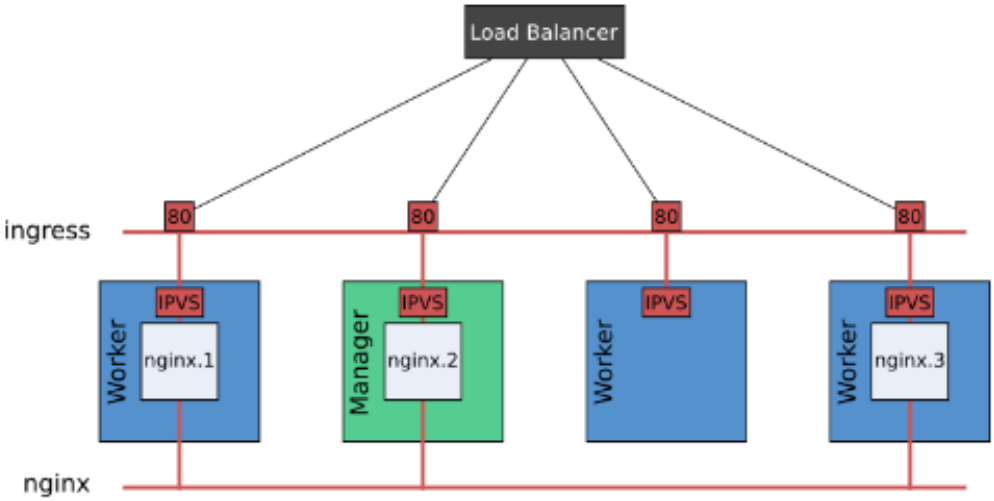
- A task is a running instance of a service on a worker node.
- Swarm schedules tasks onto worker nodes based on the service's specifications.
- Each task corresponds to a single container instance.

4. **Replication and Global Modes:**

- Replicated Services: A service with a specified number of replicas.
- Global Services: A service that runs one instance on each available node, ensuring one instance per node.

5. **Overlay Networks:**

- Swarm uses overlay networks to enable communication between containers across different nodes.
- Overlay networks provide isolation, automatic DNS resolution, and encryption by default.



6. **Load Balancing:**

- Swarm automatically distributes incoming traffic across the replicas of a service using built-in load balancing.
- Load balancing ensures efficient distribution of requests and fault tolerance.

7. **Scaling:**

- You can scale a service up or down by changing the number of replicas.
- Swarm automatically distributes replicas across available worker nodes.

8. **Rolling Updates:**

- Swarm supports rolling updates to services, allowing you to update containers without downtime.
- New replicas are gradually introduced while old replicas are removed.

9. **Health Checks:**

- Swarm provides health checks to monitor the health of containers.
- Health checks can be customized to determine container health and take actions accordingly.

10. **Service Discovery:**

- Swarm includes a built-in DNS server for service discovery using service names instead of IP addresses.

11. **Secrets and Configs:**

- Secrets allow you to manage sensitive data separately from application code.
- Configs allow you to manage configuration data separately from the codebase.

12. **High Availability:**

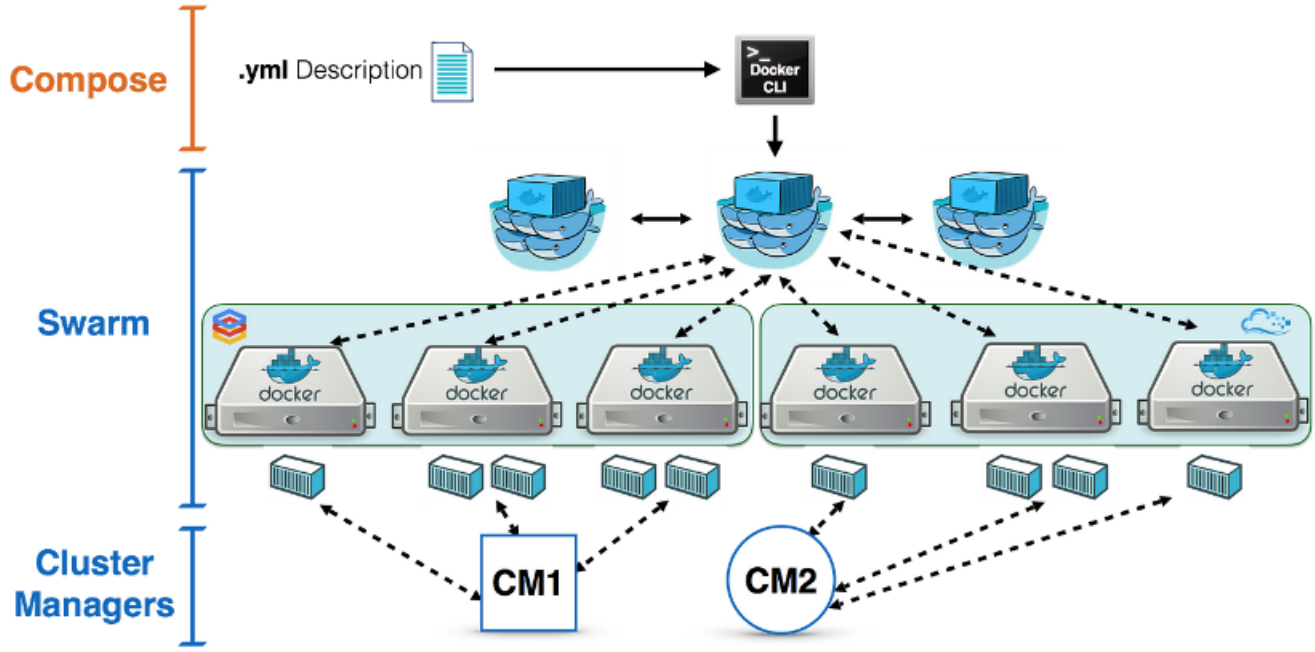
- Manager nodes operate in an active-passive mode to ensure high availability.
- If the active manager fails, a passive one takes over.

13. **Integration with Docker Compose:**

- Swarm can deploy services defined in Docker Compose files, simplifying the transition from local development to production.

14. **API and CLI:**

- Docker Swarm can be managed through the Docker CLI and RESTful API, enabling both manual and automated management.



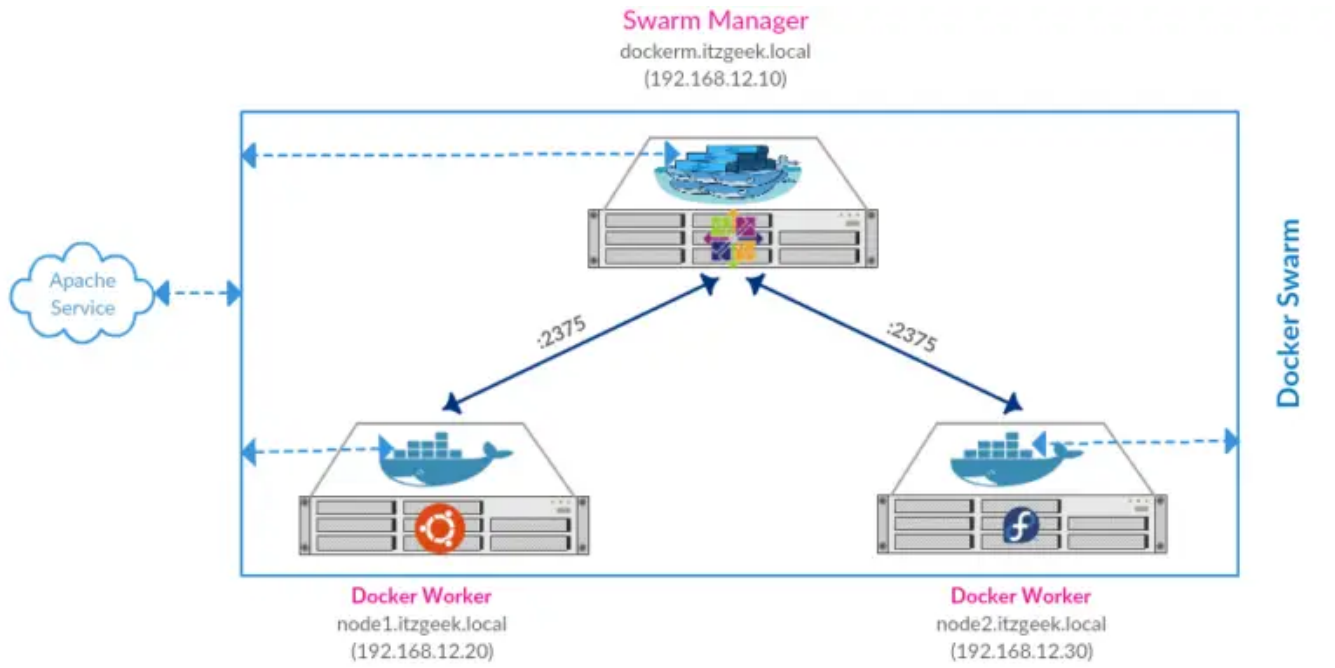
Use Cases and Considerations:

- Docker Swarm is suitable for smaller teams and projects looking for an easy-to-use orchestration solution.
- It's a good choice for applications that need basic orchestration capabilities and quick setup.
- For more complex deployments, Kubernetes might offer a broader feature set and more advanced capabilities.

Docker Swarm simplifies the process of managing and scaling containerized applications by providing essential orchestration features. While it may not be as feature-rich as some other orchestration platforms, it strikes a balance between ease of use and functionality, making it a great choice for many scenarios.

Docker Swarm Setup

Setting up a Docker Swarm cluster involves creating a manager node and one or more worker nodes to manage and distribute containers. Here's a step-by-step guide to setting up a Docker Swarm cluster with the specified prerequisites:



Providing additional steps to follow after setting up the prerequisites for your Docker Swarm cluster. These steps involve configuring the hosts file with hostnames and testing connectivity using those hostnames. Here's a breakdown of the provided steps:

Step 1: Edit /etc/hosts File

You're updating the `/etc/hosts` file on both the manager and worker nodes to associate hostnames with IP addresses. This is a common practice to simplify communication between nodes using human-readable names.

- Log in to the manager node:

  ```
  ssh ubuntu@manager_ip
  ```

- Edit the hosts file:

  ```
  sudo vim /etc/hosts
  ```

- Add the following Example lines (replace IP addresses with your actual ones):

  ```
  18.222.159.153 manager
  13.58.74.39 worker01
  18.220.166.15 worker02
  ```

- Save and exit the file.

Repeat the same steps on worker nodes.

Step 2: Test Connectivity

After updating the hosts file, you can use the hostnames instead of IP addresses to test connectivity between nodes.

- Ping the manager node from the manager itself:

  ```
  ping -c 3 manager
  ```

- Ping the worker01 node from the manager:

```
ping -c 3 worker01
```

- Ping the worker02 node from the manager:

```
ping -c 3 worker02
```

Make sure that each ping command is successful and you're able to reach all nodes using their respective hostnames.

By adding these entries to the hosts file, you make it easier to communicate between nodes using recognizable names rather than IP addresses. This can be especially useful when managing a cluster with multiple nodes. Just make sure the entries match the actual IP addresses of your manager and worker nodes.

Install Docker-ce on all the machines

Instructions for installing **Docker CE (Community Edition)** on Ubuntu machines. These steps will help you set up Docker on each of your nodes in the Docker Swarm cluster. Here's a breakdown of the provided commands:

Step 1: Install Prerequisites:

You're installing necessary prerequisites that are required for adding external repositories and installing packages.

```
sudo apt install apt-transport-https software-properties-common ca-certifica
tes -y
```

This command installs `apt-transport-https` to support secure connections, `software-properties-common` for managing software properties, and `ca-certificates` for managing SSL certificates.

Step 2: Add Docker Key and Repository:

You're adding the Docker GPG key and the Docker CE repository to your server's sources list.

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
sudo echo "deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial
stable" > /etc/apt/sources.list.d/docker-ce.list
```

- The first command downloads the Docker GPG key and adds it to the system.
- The second command adds the Docker CE repository to the sources list.

Step 3: Update Repository and Install Docker:

You're updating the package repository and then installing Docker CE.

```
sudo apt update
sudo apt install docker-ce -y
```

- The first command updates the package repository to include the Docker packages.
- The second command installs Docker CE on your system.

Step 4: Start Docker and Enable Auto-Start:

Finally, We're starting the Docker service and enabling it to start automatically on boot.

```
sudo systemctl start docker
sudo systemctl enable docker
```

- The first command starts the Docker service.
- The second command enables Docker to start automatically on system boot.

These steps will successfully install Docker CE on your manager and worker nodes. Make sure to repeat these steps on all the machines in your Docker Swarm cluster to ensure that Docker is properly installed and configured on each node.

- These Steps to configure Docker to run as a non-root user. This is a security best practice that helps reduce the potential attack surface of your system.

Step 5: Create a New User:

You're creating a new user named "Prudhvi" using the `useradd` command.

```
useradd -m -s /bin/bash Prudhvi
```

- `-m` creates the user's home directory if it doesn't exist.
- `-s /bin/bash` sets the user's default shell to Bash.

Step 6: Add User to Docker Group:

You're adding the newly created user "Prudhvi" to the "docker" group using the `usermod` command.

```
sudo usermod -aG docker Prudhvi
```

- `-aG docker` adds the user to the "docker" group.

- The user needs to be part of the "docker" group to be able to interact with Docker without requiring root privileges.

Step 7: Test Docker as the New User:

You're logging in as the "Prudhvi" user and testing Docker using the "hello-world" image.

```
su - Prudhvi
docker run hello-world
```

- su - Prudhvi` switches to the "Prudhvi" user's environment.
- `docker run hello-world` runs the "hello-world" container to verify that Docker is working correctly for the non-root user.

By adding the "Prudhvi" user to the "docker" group, you allow the user to interact with Docker without requiring root privileges. This enhances security and follows the principle of least privilege.

Ensure that you test the Docker commands thoroughly to confirm that the "Prudhvi" user has the necessary permissions to work with Docker containers and images.

Create the Swarm Cluster

The process of setting up a Docker Swarm cluster by adding a worker node to the cluster. Here's a step-by-step breakdown of the provided instructions:

Step 1: Initialize Swarm on Manager Node:

On your manager node, you initialize the Docker Swarm using the `docker swarm init` command. This command generates the necessary tokens and configuration to manage the swarm.

```
docker swarm init --advertise-addr 13.232.148.157
```

- `--advertise-addr 13.232.148.157` : Specifies the IP address used for communication with other nodes in the swarm.

```
mohammad@ip-172-31-31-143:~$ docker swarm init --advertise-addr 13.232.148.157
Swarm initialized: current node (zgffk5in71vbts2lbvds776tk) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4x90xojqkcidrqzt1emut1bgwrqqulzs3lcnj555j9eu20sm9w-510uatr4x0cqdocam1d7h86t9 13.232.148.157:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Step 2: Generate Join Token:

After initializing the swarm on the manager node, the `join-token` is generated. This token is required to allow worker nodes to join the swarm.

Step 3: Join Worker Node to Swarm:

On the worker node (`worker01` in this case), you'll use the generated join token to join the node to the swarm.

- SSH into the worker01 node:

  ```
  ssh ubuntu@worker01_ip
  ```

- Run the following command to join the worker node to the swarm using the generated join token:

  ```
  docker swarm join --token <worker_token> <manager_ip>:2377
  ```

- Replace `<worker_token>` with the actual token generated on the manager node.
- Replace `<manager_ip>` with the IP address of the manager node.

```
mohammad@ip-172-31-24-107:~$ docker swarm join --token SWMTKN-1-4x90xojqkcidrqzt1emut1bgwrqqulzs31cnj555j9eu20sm9w-510uatr4x0cqdocam1d7h86t9 13.232.148.157:2377
This node joined a swarm as a worker.
```

After executing this command, the worker node (`worker01`) will successfully join the Docker Swarm cluster managed by the manager node.

This process allows you to create a multi-node Docker Swarm cluster, where the manager node handles orchestration and management tasks, and the worker nodes execute tasks assigned by the manager. This setup provides scalability, load balancing, and high availability for your containerized applications.

where we can find worker token? Explain in detail

The worker join token is generated on the manager node when you initialize the Docker Swarm using the `docker swarm init` command. This token is required for worker nodes to join the swarm. Here's how you can find and retrieve the worker join token from the manager node:

1. **On the Manager Node:**

   After running the `docker swarm init` command on the manager node, the terminal output will include the generated worker join token. It will look something like this:

   ```
   To add a worker to this swarm, run the following command:

   docker swarm join --token <worker-token> <manager-ip>:2377
   ```

The `<worker-token>` is the token you need to provide to worker nodes to allow them to join the swarm.

2. **Regenerating Worker Token (If Needed):**

If you somehow missed the initial token output or if you need to regenerate a new token, you can do so using the following command on the manager node:

```
docker swarm join-token worker
```

This command will output the join command, including the worker token, which you can use to add worker nodes to the swarm.

3. **Viewing Worker Tokens Later:**

   If you need to view the worker join token again at a later time, you can use the same `docker swarm join-token worker` command on the manager node.

Remember that the worker join token is sensitive information and should be kept secure. Treat it like a password, as anyone with access to the token can join worker nodes to the swarm. If you're concerned about security, you can also rotate or regenerate tokens periodically.

Check it by running the following command on the 'manager' node.

- **docker node ls**

```
root@ip-172-31-44-230:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
00plmr34714ok62zkmuvromrr     ip-172-31-37-61     Ready               Active                                  18.09.0
ptschibesmy2yun518akrjatx *   ip-172-31-44-230    Ready               Active              Leader              18.09.0
ry5gs6phuilbfbkvcjg80i0xr     ip-172-31-47-13     Ready               Active                                  18.09.0
```

- Now you see the 'worker01‘ and ‘worker02’ nodes are joined to the swarm clusterm

Deploying First Service to the Cluster

Create and deploy your first service in the Docker Swarm cluster. Let's break down the steps you've provided:

Step 1: Create and Deploy Nginx Service:

You're using the `docker service create` command to create and deploy an Nginx web server service named "my-web" in the Swarm cluster.

```
docker service create --name my-web --publish 8080:80 nginx:1.13-alpine
```

- `--name my-web` : Sets the name of the service to "my-web".
- `--publish 8080:80` : Publishes port 8080 on the host to port 80 in the container.
- `nginx:1.13-alpine` : Specifies the image to use for the service, which is Nginx version 1.13 based on the Alpine Linux distribution.

```
mohammad@ip-172-31-31-143:~$ docker service create --name my-web --publish 8080:80 nginx:1.13-alpine
sgotbz7ea4tltp7usallwvodm
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
```

Step 2: Check the Created Service:

You're using the `docker service ls` command to list the services in the swarm and verify that the Nginx service ("my-web") has been created and deployed.

```
docker service ls
```

```
mohammad@ip-172-31-31-143:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
sgotbz7ea4tl        my-web              replicated          1/1                 nginx:1.13-alpine   *:8080->80/tcp
mohammad@ip-172-31-31-143:~$
```

Step 3: Check on Which Node Service is Running:

To determine on which node the service is running, you need to check the service tasks. Each service task represents a running container instance of the service.

- Use the `docker service ps` command to list the tasks for the "my-web" service:

  ```
  docker service ps my-web
  ```

- This command will show you information about the tasks, including the node on which they are running.

```
root@ip-172-31-44-230:~# docker service ps my-web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE               ERROR               PORTS
53g8rd5fmhk3        my-web.1            nginx:1.13-alpine   ip-172-31-44-230    Running             Running about a minute ago
```

- Check on the node whether it is running or not, go to the node and run the command

```
root@ip-172-31-44-230:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
90c1ec259400        nginx:1.13-alpine   "nginx -g 'daemon of…"   4 minutes ago       Up 4 minutes        80/tcp              my-web.1.53g8rd5fmhk3iojlaemrbj48x
```

You've successfully created and deployed an Nginx service named "my-web" in the Docker Swarm cluster. The service is accessible via port 8080 on the host, and you've confirmed its status and the node on which it's running. Keep in mind that Docker Swarm takes care of managing the service's replicas and distributing them across available nodes in the swarm.

Change the load from manager to worker node

It seems like you're explaining how to change the load from a manager node to a worker node in a Docker Swarm cluster. You're draining a manager node to stop it, and you want to ensure that the service running on that node is moved to another node in the cluster. Here's a summary of the steps you're describing:

Step 1: Draining the Manager Node:

You're using the `docker node update` command with the `--availability drain` option to mark the manager node as drained. This prevents new tasks from being scheduled on the node, allowing existing tasks to finish.

```
docker node update --availability drain ip-172-31-44-230
```

```
root@ip-172-31-44-230:~# docker node update --availability drain  ip-172-31-44-230
ip-172-31-44-230
```

```
root@ip-172-31-44-230:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
00plmr34714ok62zkmuvromrr     ip-172-31-37-61     Ready               Active                                  18.09.0
ptschibesmy2yun518akrjatx *   ip-172-31-44-230    Ready               Drain               Leader              18.09.0
ry5gs6phuilbfbkvcjg80i0xr     ip-172-31-47-13     Ready               Active                                  18.09.0
```

Step 2: Moving the Service:

Once the manager node is drained, Docker Swarm will automatically redistribute the service's tasks to other available nodes in the cluster, ensuring that the desired number of replicas is maintained.

Step 3: Verifying the Service:

You can use the `docker service ps` command to monitor the status of the service's tasks and their distribution across nodes:

```
docker service ps <service-name>
```

Replace `<service-name>` with the actual name of your service.

```
root@ip-172-31-44-230:~# docker service ps my-web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
n5eai8dois5q        my-web.1            nginx:1.13-alpine   ip-172-31-37-61     Running             Running 4 minutes ago
53g8rd5fmhk3         \_ my-web.1        nginx:1.13-alpine   ip-172-31-44-230    Shutdown            Shutdown 4 minutes ago
root@ip-172-31-44-230:~#
```

By draining the manager node, Docker Swarm ensures that tasks are moved away from the node before it's taken offline for maintenance or other purposes. This process helps maintain the availability and reliability of your services.

Scale up the service

Steps involving scaling a Docker service, managing node availability, and observing how containers are distributed among nodes in a Docker Swarm cluster. Let's break down the steps you've provided:

Step 1: Scale the Service:

You're scaling the "my-web" service to have a total of 10 replicas running across the cluster. This command will distribute the containers among available nodes.

```
docker service scale my-web=10
```

- Check the service info

```
root@ip-172-31-44-230:~# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
kuy5o0bu3lmm        my-web              replicated          10/10               nginx:1.13-alpine   *:8080->80/tcp
```

Step 2: Check Service Information:

You can verify the distribution of replicas among nodes by inspecting the service's tasks:

```
docker service ps my-web
```

This command will show the status of the tasks and their placement on different nodes.

```
root@ip-172-31-44-230:~# docker service ps my-web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
n5eai8dois5q        my-web.1            nginx:1.13-alpine   ip-172-31-37-61     Running             Running 23 minutes ago
53g8rd5fmhk3         \_ my-web.1        nginx:1.13-alpine   ip-172-31-44-230    Shutdown            Shutdown 23 minutes ago
j83csmp2fra8        my-web.2            nginx:1.13-alpine   ip-172-31-37-61     Running             Running 4 minutes ago
cm47pdolovt7        my-web.3            nginx:1.13-alpine   ip-172-31-47-13     Running             Running 4 minutes ago
fjp7tzk2qwf2        my-web.4            nginx:1.13-alpine   ip-172-31-47-13     Running             Running 4 minutes ago
x6xc6lgwfsve        my-web.5            nginx:1.13-alpine   ip-172-31-37-61     Running             Running 4 minutes ago
885ft9y30z9l        my-web.6            nginx:1.13-alpine   ip-172-31-37-61     Running             Running 4 minutes ago
led8uqmoikhy        my-web.7            nginx:1.13-alpine   ip-172-31-47-13     Running             Running 4 minutes ago
jsa0wkjl6ugp        my-web.8            nginx:1.13-alpine   ip-172-31-47-13     Running             Running 4 minutes ago
pt9fbdeo64pa        my-web.9            nginx:1.13-alpine   ip-172-31-47-13     Running             Running 4 minutes ago
x5bv7w4n03pw        my-web.10           nginx:1.13-alpine   ip-172-31-37-61     Running             Running 4 minutes ago
```

Step 3: Bring Back the Manager Node Online:

You're using the `docker node update` command to change the availability state of the manager node back to "active" after it was drained.

```
docker node update --availability active ip-172-31-44-230
```

```
root@ip-172-31-44-230:~# docker node update --availability drain  ip-172-31-44-230
ip-172-31-44-230
```

```
root@ip-172-31-44-230:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
00plmr34714ok62zkmuvromrr     ip-172-31-37-61     Ready               Active                                  18.09.0
ptschibesmy2yun518akrjatx *   ip-172-31-44-230    Ready               Drain               Leader              18.09.0
ry5gs6phuilbfbkvcjg80i0xr     ip-172-31-47-13     Ready               Active                                  18.09.0
```

Step 4: Shutdown One of the Worker Nodes:

You're using the `docker node update` command again to change the availability state of a worker node (ip-172-31-44-231) to "active." This node was presumably shut down earlier.

```
docker node update --availability active ip-172-31-44-231
```

```
root@ip-172-31-44-230:~# docker ps
```

Step 5: Check Containers on Manager Node:

On the manager node, you can check the list of containers running on it using the following command:

```
docker ps
```

Step 6: Check Containers on Worker Node:

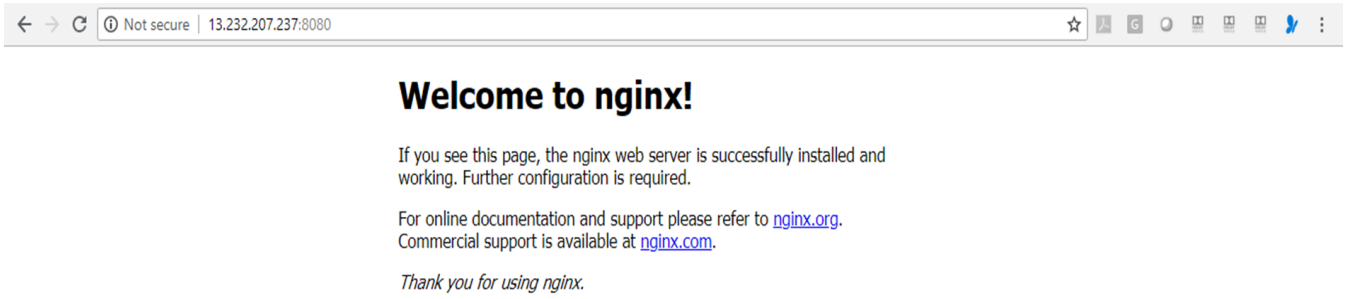
On the worker02 node (ip-172-31-44-231), you can check the list of containers running on it using the following command:

```
docker ps
```

```
root@ip-172-31-37-61:~# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS              PORTS               NAMES
9bbfe50f0dee        nginx:1.13-alpine    "nginx -g 'daemon of…"   24 minutes ago      Up 24 minutes       80/tcp              my-web.10.x5bv7w4n03pwe5wv8r2e4olqa
ed80e132f221        nginx:1.13-alpine    "nginx -g 'daemon of…"   24 minutes ago      Up 24 minutes       80/tcp              my-web.6.885ft9y30z9lub3wpp410a1r9
c61547f7f3fa        nginx:1.13-alpine    "nginx -g 'daemon of…"   24 minutes ago      Up 24 minutes       80/tcp              my-web.5.x6xc6lgwfsvewtx85k02pz1mf
c6b4940b170f        nginx:1.13-alpine    "nginx -g 'daemon of…"   24 minutes ago      Up 24 minutes       80/tcp              my-web.2.j83csmp2fra8f7rveycel5i5h
9c9f50bf30ac        nginx:1.13-alpine    "nginx -g 'daemon of…"   43 minutes ago      Up 43 minutes       80/tcp              my-web.1.n5eai8dois5qmx8kjfl8lyje2
```

By performing these steps, you're managing the scaling and distribution of containers across nodes in your Docker Swarm cluster. You're also observing how Docker Swarm handles the distribution of containers when nodes are brought back online or taken offline. This orchestration and load balancing are key features of Docker Swarm, helping to ensure the high availability and efficiency of your containerized applications.

Open your web browser and type the worker node IP address with port 8080



What is Rolling Update in Docker swarm ?

A rolling update is a strategy for updating services in a Docker Swarm cluster while ensuring that the service remains available to users during the update process. This strategy is designed to **minimize downtime and maintain service availability** by gradually updating tasks in a controlled manner. Here's how a rolling update works in detail:

Step-by-Step Rolling Update Process:

1. **Prepare a New Image:**
    - Before starting a rolling update, you need to have a new version of the image that you want to deploy. This could be a newer version of your application code or configuration.
2. **Update Service with New Image:**

- Use the `docker service update` command to initiate the rolling update. You'll provide the name of the service you want to update and specify the new image version.

```
docker service update --image new-image-version service-name
```

3. **Rolling Update Strategy:**
   - Docker Swarm uses a rolling update strategy by default. This means that during the update, Swarm replaces tasks with the new version of the image gradually, one task at a time, across different nodes in the swarm.
4. **Control Over Replicas:**
   - You can control the pace of the update by specifying the number of replicas that should be updated at once. This can be done with the `--update-parallelism` option.

```
docker service update --update-parallelism <num> service-name
```

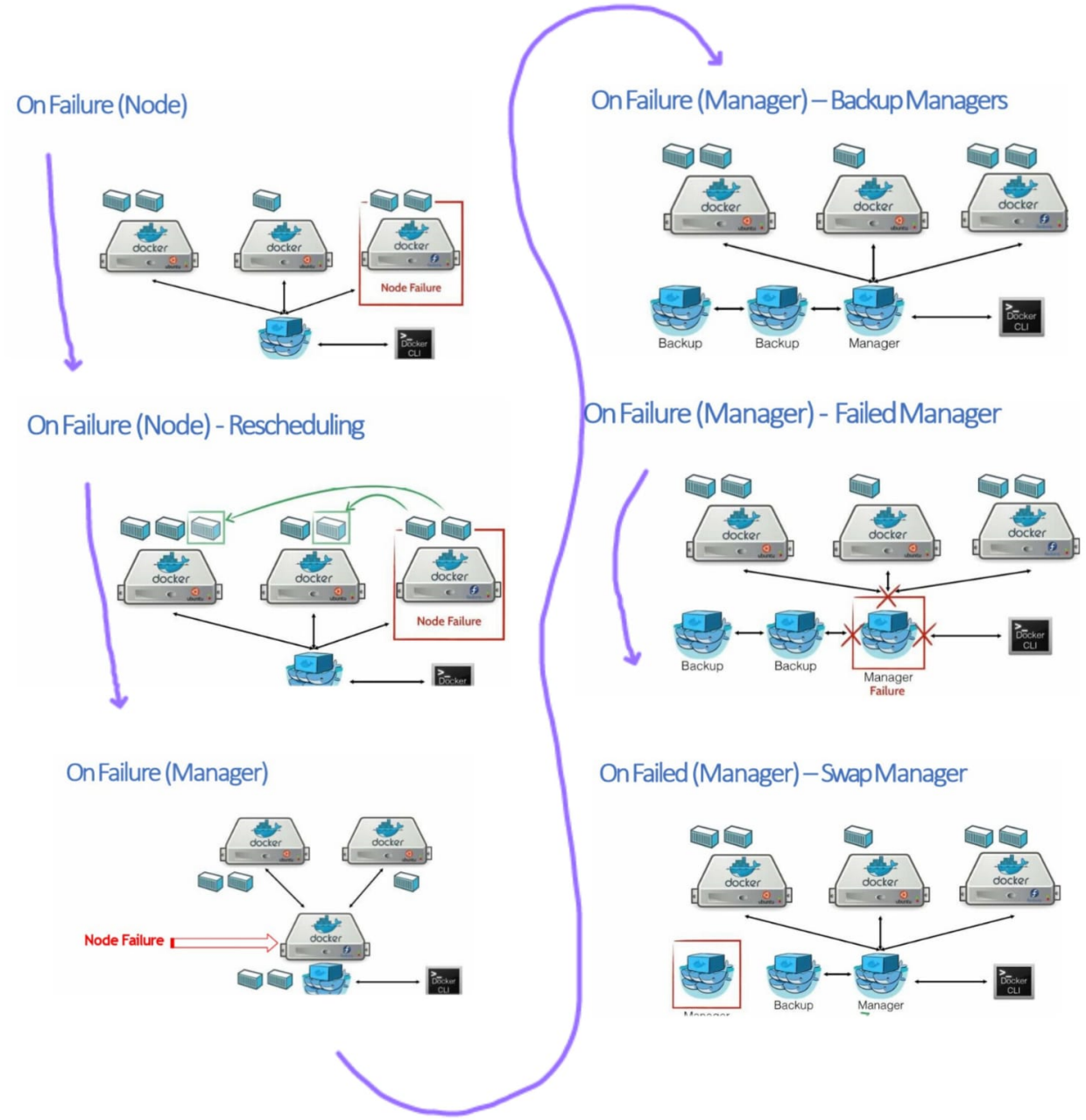
5. **Health Checks:**
   - Docker Swarm monitors the health of updated tasks. Before removing old tasks, it ensures that the new tasks are healthy and have passed their health checks.
6. **Scaling Up and Down:**
   - While the rolling update is in progress, you can scale the service up or down as needed. Swarm maintains the desired replica count while applying the update.
7. **Monitoring:**
   - You can monitor the progress of the rolling update using the `docker service ps` command. This shows the status of tasks, including if they are "updating," "running," or "complete."
8. **Completion and Rollback:**
   - Once all tasks have been updated successfully, the rolling update is complete, and the service is running the new version of the image.

- In case of issues, you can roll back to the previous version using the same `docker service update` command with the previous image version.

Advantages of Rolling Updates:

- **Minimal Downtime:** The rolling update strategy helps ensure that a certain percentage of the service remains available at all times, minimizing downtime during the update process.
- **Controlled Process:** You have control over the pace of the update, parallelism, and the ability to monitor the update progress.
- **Automatic Rollback:** In case of issues, Docker Swarm supports automatic rollback to the previous version.

By using rolling updates in Docker Swarm, you can keep your services up-to-date with minimal service disruption and maintain a high level of availability for your applications.




```
-- Prudhvi Vardhan ( LinkedIN)
```