Setting up Docker Swarm High Availability in Production

Link: https://betterstack.com/community/guides/scaling-docker/ha-docker-swarm/

DockerHigh Availability

Ayooluwa Isaiah
Updated on January 14, 2026

Contents

- Prerequisites
- Explaining Docker Swarm terminology
- Docker Swarm requirements for high availability
- Step 1 — Installing Docker
- Step 2 — Executing the Docker command without sudo
- Step 3 — Initializing the Swarm Cluster
- Step 4 — Adding worker nodes to the cluster
- Step 5 — Adding manager nodes to the cluster
- Step 6 — Draining a node on the swarm
- Step 7 — Deploying a Highly Available NGINX service
- Step 8 — Draining the other manager nodes
- Step 9 — Testing the failover mechanism
- Monitoring your Docker Swarm cluster
- Final thoughts

Docker Swarm is a container orchestration tool that makes it easy to manage and scale your existing Docker infrastructure. It consists of a pool of Docker hosts that run in Swarm mode with some nodes acting as managers, workers, or both. Using Docker Swarm mode to manage your Docker containers brings the following benefits:

- It allows you to incrementally apply updates with zero downtime.
- It increases application resilience to outages by reconciling any differences between the actual state and your expressed desired state.
- It eases the process of scaling your applications since you only need to define the desired number of replicas in the cluster.
- It is built into the `docker` CLI, so you don't need additional software to get up and running.
- It enables multi-host networking such that containers deployed on different nodes can communicate with each other easily.

In this tutorial, you will learn key concepts in Docker Swarm and set up a highly available Swarm cluster that is resilient to failures. You will also learn some best practices and recommendations to ensure that your Swarm setup is fault tolerant.

https://www.youtube.com/embed/wQKjCDD7nfk

Prerequisites

Before proceeding with this tutorial, ensure that you have access to five Ubuntu 22.04 servers. This is necessary to demonstrate a highly available set up, although it is also possible to run Docker Swarm on a single machine. You also need to configure each server with a user that has administrative privileges.

The following ports must also be available on each server for communication purposes between the nodes. On Ubuntu 22.04, they are open by default:

- TCP port 2377 for cluster management communications,
- TCP and UDP port 7946 for communication among nodes,
- TCP and UDP port 4789 for overlay network traffic.

Centralize & visualize your logs. Query everything with SQL.

Explaining Docker Swarm terminology

Before proceeding with this tutorial, let's examine some terms and definitions in Docker Swarm so that you have enough understanding of what each one means when they are used in this article and in other Docker Swarm resources.

- **Node**: refers to an instance of the Docker engine in the Swarm cluster.
- **Manager nodes**: they are tasked with handling orchestration and cluster management functions, and dispatching incoming tasks to worker nodes. They can also act as worker nodes unless placed in Drain mode (recommended).
- **Leader**: this is a specific manager node that is elected to perform orchestration tasks and management/maintenance operations by all the manager nodes in the cluster using the Raft Consensus Algorithm.
- **Worker nodes**: are Docker instances whose sole purpose is to receive and execute Swarm tasks from manager nodes.
- **Swarm task**: refers to a Docker container and the commands that run inside the container. Once a task is assigned to a node, it can run or fail but it cannot be transferred to a different node.
- **Swarm service**: this is the mechanism for defining tasks that should be executed on a node. It involves specifying the container image and commands that should run inside the container.
- **Drain**: means that new tasks are no longer assigned to a node, and existing tasks are reassigned to other available nodes.

Docker Swarm requirements for high availability

A highly available Docker Swarm setup ensures that if a node fails, services on the failed node are re-provisioned and assigned to other available nodes in the cluster. A Docker Swarm setup that consists of one or two manager nodes is not considered highly available because any incident will cause operations on the cluster to be interrupted. Therefore the minimum number of manager nodes in a highly available Swarm cluster should be three.

The table below shows the number of failures a Swarm cluster can tolerate depending on the number of manager nodes in the cluster:

Manager Nodes	Failures tolerated
1	0
2	0
3	1

Manager Nodes	Failures tolerated
4	1
5	2
6	2
7	3

As you can see, having an even number of manager nodes does not help with failure tolerance, so you should always maintain an odd number of manager nodes. Fault tolerance improves as you add more manager nodes, but Docker recommends no more than seven managers so that performance is not negatively impacted since each node must acknowledge proposals to update the state of the cluster.

You should also distribute your manager nodes in separate locations so they are not affected by the same outage. If they run on the same server, a hardware problem could cause them all to go down. The high availability Swarm cluster that you will be set up in this tutorial will therefore exhibit the following characteristics:

- 5 total nodes (2 workers and 3 managers) with each one running on a separate server.
- 2 worker nodes (`worker-1` and `worker-2`).
- 3 manager nodes (`manager-1`, `manager-2`, and `manager-3`).

Step 1 — Installing Docker

In this step, you will install Docker on all five Ubuntu servers. Therefore, execute all the commands below (and in step 2) on all five servers. If your host offers a snapshot feature, you may be able to run the commands on a single server and use that server as a base for the other four instances.

Let's start by installing the latest version of the Docker Engine (20.10.18 at the time of writing). Go ahead and update the package information list from all configured sources on your system:

```
sudo apt update
```

Afterward, install the following packages to allow `apt` use packages over HTTPS:

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Next, add the GPG key for the official Docker repository to the server:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

Once the GPG key is added, include the official Docker repository in the server's apt sources list.

```
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

Finally, update apt once again and install the Docker Engine:

```
sudo apt update
```

```
sudo apt install docker-ce
```

Once the relevant packages are installed, you check the status of the `docker` service using the command below:

```
sudo systemctl status docker
```

If everything goes well, you should observe that the container engine is active and running on your server:

Output

```
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2022-09-21 10:18:15 UTC; 58s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 25355 (dockerd)
      Tasks: 7
     Memory: 22.2M
        CPU: 346ms
     CGroup: /system.slice/docker.service
             └─25355 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

In the next step, we will add the current user to the `docker` group so that you can use the `docker` command without escalating to administrative privileges (using `sudo`) which can lead to security issues.

Step 2 — Executing the Docker command without sudo

By default, the `docker` command can only be executed by the root user or any user in the `docker` group (auto created on installation). If you execute a `docker` command without prefixing it with `sudo` or running it through a user that belongs to the `docker` group, you will get a permission error that looks like this:

Output

```
Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json":
dial unix /var/run/docker.sock: connect: permission denied
```

As mentioned earlier, using `sudo` with `docker` is a security risk, so the solution to the above error is to add the relevant user to the `docker` group which can be achieved through the command below:

```
sudo usermod -aG docker ${USER}
```

Next, run the following command and enter the user's password when prompted for the changes to take effect:

```
su - ${USER}
```

You should now be able to run `docker` commands without prefixing them with `sudo`. For example, when you run the command below:

```
docker ps
```

You should observe the following output:

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Before proceeding to the next step, ensure that all the commands in step 1 and step 2 have been executed on all five servers.

Step 3 — Initializing the Swarm Cluster

At this point, each of your five Docker instances are acting as separate hosts and not as part of a Swarm cluster. Therefore, in this step, we will initialize the Swarm cluster on the `manager-1` server and add the hosts to the cluster accordingly.

Start by logging into one of the Ubuntu servers (`manager-1`), and retrieve the private IP address of the machine using the following command:

```
hostname -I | awk '{print $1}'
```

Output

```
<manager_1_server_ip>
```

Copy the IP address to your clipboard and replace the `<manager_1_server_ip>` placeholder in the command below to initialize Swarm mode:

```
docker swarm init --advertise-addr <manager_1_server_ip>
```

Output

```
Swarm initialized: current node (9r83zto8qpqiazt6slxfkjypq) is now a manager.

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

docker swarm join --token <token> <manager_1_server_ip>:<port>

```


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

```
instructions.
```

The command above enables Swarm mode on the node and configures it as the first manager of the cluster. Ensure to copy the entire command to your clipboard (and replace the placeholders) as it will be utilized in the next section.

You can view the current state of the Swarm using the command below:

```
docker info
```

Output

```
. . .
 Swarm: active
  NodeID: 9r83zto8qpqiazt6slxfkjypq
  Is Manager: true
  ClusterID: q6laywz9u8xlis9wlkbzap8i0
  Managers: 1
  Nodes: 1
. . .
```

You can also use the command below to view information regarding the nodes on the cluster:

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Active         Leader           20.10.18
```

The `*` next to the node ID indicates that you're currently connected to this node. Here's the meaning of the other values:

- **Ready**: the node is recognized by the manager and can participate in the cluster.
- **Active**: the node is used as a worker also to run Docker containers.
- **Leader**: the node is a manager node and is also the cluster's leader.

In the next section of this tutorial, we will add two workers to our cluster bringing the total nodes to three.

Step 4 — Adding worker nodes to the cluster

Adding worker modes to a cluster can be done by running the command copied from step 3 above (the `docker swarm init` output) on the `worker-1` and `worker-2` servers:

```
docker swarm join --token <token> <manager_1_server_ip>:<port>
```

Output

```
This node joined a swarm as a worker.
```

If you forgot to copy the `join` command for workers, use the command below on the `manager-1` server to retrieve it:

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

Output

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

```

docker swarm join --token <token> <manager_1_server_ip>:<port>

```

```

After executing the `join` command on each worker node, you should be able to see the updated list of Swarm nodes when you run the command below on your `manager-1` server:

```
 docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Active         Leader           20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

The empty `MANAGER` column for the `worker-1` and `worker-2` nodes identifies them as worker nodes. Note that you can promote a worker node to a manager mode by using the command below:

```
docker node promote <worker_node_id>
```

Output

```
Node <worker_node_id> promoted to a manager in the swarm.
```

Step 5 — Adding manager nodes to the cluster

In this section, you will add the remaining two nodes to the cluster as managers. This time, you need to retrieve the `join` command for manager nodes by running the command below on the `manager-1` server:

```
docker swarm join-token manager
```

Output:

Output

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

    docker swarm join --token <token> <manager_1_server_ip>:<port>
```

Copy the generated command above and execute it on each additional manager nodes (`manager-2` and `manager-3`) as shown below:

```
docker swarm join --token <token> <manager_1_server_ip>:<port>
```

Output

```
This node joined a swarm as a manager.
```

Finally, verify the status of the cluster nodes using the following command:

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Active         Leader           20.10.18
uspt9qwqnzqwl78gbxc7omja7     manager-2   Ready     Active         Reachable        20.10.18
txrdxwuwjpg5jjfer3bcmtc5r     manager-3   Ready     Active         Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

Note that you can also demote a manager node to a worker as follows:

```
docker node demote <manager_node_id>
```

Output

```
Manager <manager_node_id> demoted in the swarm.
```

Step 6 — Draining a node on the swarm

At the moment, all five nodes in the swarm cluster are running with `Active` availability. This means they are all available to accept new tasks from the swarm manager (including the leader). If you want to avoid scheduling tasks on a node, you need to `drain` it such that no new tasks are assigned to it, and existing tasks are stopped and relaunched on a replica node with `Active` availability.

In this section, you will drain the manager node so that it is no longer able to receive new tasks, which should help to improve its performance since no resources will be allocated towards running Docker containers.

Before you can drain a node, you need to figure out ID of the node to be drained using the following command on the `manager-1` server:

```
docker node ls
```

Copy the ID of the `Leader` node:

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Active         Leader           20.10.18
. . .
```

Next, update the node availability using the below command:

```
docker node update  --availability drain <leader_node_id>
```

Output

```
<leader_node_id>
```

When you run `docker node ls` once more, you should observe that the `AVAILABILITY` of the `Leader` node has been changed to `Drain`:

```
docker node ls
```

Output:

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Drain          Leader           20.10.18
uspt9qwqnzqwl78gbxc7omja7     manager-2   Ready     Active         Reachable        20.10.18
txrdxwuwjpg5jjfer3bcmtc5r     manager-3   Ready     Active         Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

You can also use the `docker node inspect` command to check the availability of a node:

```
docker node inspect --pretty <node_id>
```

Output

```
ID:                     <node_id>
Hostname:               manager-1
```

```
Joined at:              2022-09-21 11:07:08.730840341 +0000 utc
Status:
 State:                 Ready
```

Availability: Drain

```
 Address:               116.203.21.130
Manager Status:
 Address:               <ip_address>
 Raft Status:           Reachable
 Leader:                Yes


. . .
```

If you change your mind about draining a node, you can return it to an active state by executing the following command:

Output

```
docker node update  --availability active <node_id>
```

Step 7 — Deploying a Highly Available NGINX service

In this section, you will deploy a service to the running cluster using a High Availability Swarm configuration. We'll be utilizing the official NGINX docker image for demonstration purposes, but you can use any Docker image you want.

Start by creating the compose file for the `nginx` service on the `manager-1` server with all the necessary configurations for High Availability mode.

```
nano nginx.yaml
```

nginx.yaml

```
version: '3.7'

networks:
```

```
  nginx:
    external: false

services:

  # --- NGINX ---
  nginx:
      image: nginx:latest
      ports:
        - '8088:80'
      deploy:
        replicas: 4
        update_config:
          parallelism: 2
          order: start-first
          failure_action: rollback
          delay: 10s
        rollback_config:
          parallelism: 0
          order: stop-first
        restart_policy:
          condition: any
          delay: 5s
          max_attempts: 3
          window: 120s
      healthcheck:
        test: ["CMD", "service", "nginx", "status"]
      networks:
        - nginx
```

The above file configures an `nginx` service with four replicas. Updates to the containers will be carried out in batches (two at a time) with a wait time of 10 seconds before updating the next batch. If an update failure is detected, it will roll back to the previous configuration. Please see the Compose file reference for more information.

Go ahead and deploy the NGINX stack on the `manager-1` node using the command below:

```
docker stack deploy -c nginx.yaml nginx
```

Output

```
Creating network nginx_nginx
Creating service nginx_nginx
```

Once you deploy the stack, you will be able to see a list of running services on the cluster using the command below:

```
docker service ls
```

Output

```
ID             NAME          MODE         REPLICAS   IMAGE          PORTS
xh24wj31z4ml   nginx_nginx   replicated   4/4        nginx:latest   *:8088->80/tcp
```

You can also see which nodes are running the service:

```
docker service ps <service_id>
```

Output

```
ID             NAME            IMAGE          NODE        DESIRED STATE   CURRENT
STATE           ERROR     PORTS
inel921owq9b   nginx_nginx.1   nginx:latest   manager-2   Running         Running 2 minutes
ago
uk18jo6wwlq6   nginx_nginx.2   nginx:latest   manager-3   Running         Running 2 minutes
ago
ijbalagf7isy   nginx_nginx.3   nginx:latest   worker-1    Running         Running 2 minutes
ago
drgu007baw8z   nginx_nginx.4   nginx:latest   worker-2    Running         Running 2 minutes
ago
```

Each replica runs on the four `Active` nodes in this case. The `DESIRED STATE` and `CURRENT STATE` columns lets you determine the tasks are running according to the service definition.

If you want to see details about the container for a task, run `docker ps` on the relevant node. For example, on `manager-3`:

```
docker ps
```

Output

```
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS                    PORTS     NAMES
62e396145307   nginx:latest   "/docker-entrypoint.…"   7 minutes ago   Up 7 minutes
(healthy)   80/tcp    nginx_nginx.2.uk18jo6wwlq6qfrv2lr6zk1xz
```


Monitor your Docker Swarm cluster with Better Stack

While Docker Swarm handles container orchestration, Better Stack provides comprehensive monitoring for your distributed services. Track container health, collect logs from all nodes, and get alerted when services fail with infrastructure monitoring that checks your cluster every 30 seconds.

Predictable pricing and up to 30x cheaper than Datadog. Start free in minutes.

Step 8 — Draining the other manager nodes

As mentioned earlier, manager nodes should ideally be only responsible for management-related tasks. Currently, two of the three manager nodes are running replicas of the NGINX service which could hamper management operations under certain conditions. To prevent such interference, it is best to mark them as unavailable for running tasks by draining them as follows:

```
docker node update --availability drain <manager_node_id>
```

Once you've done so for both the `manager-2` and `manager-3` nodes, you'll observe that their `AVAILABILITY` has been updated to drain:

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Drain          Leader           20.10.18
uspt9qwqnzqwl78gbxc7omja7     manager-2   Ready     Drain          Reachable        20.10.18
txrdxwuwjpg5jjfer3bcmtc5r     manager-3   Ready     Drain          Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

The NGINX replicas that were running on both manager nodes are subsequently stopped and reassigned to each worker node. You can confirm that four replicas are still running using the command below:

```
docker service ls
```

Output

```
ID             NAME           MODE         REPLICAS   IMAGE          PORTS
xh24wj31z4ml   nginx_nginx    replicated   4/4        nginx:latest   *:8088->80/tcp
```

Now, confirm which nodes are running each replica through the command below:

```
docker service ps <service_id>
```

Output

```
ID             NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE              ERROR     PORTS
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 19
minutes ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 19
minutes ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 19
minutes ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 19
minutes ago
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running about an
hour ago
drgu007baw8z   nginx_nginx.4       nginx:latest   worker-2    Running         Running about an
hour ago
```

As you can see, the `manager-2` and `manager-3` tasks were shut down 19 minutes ago and subsequently reassigned to `worker-2` and `worker-1` respectively. Your manager nodes are now only responsible for cluster management activities and maintaining high availability of the cluster.

Step 9 — Testing the failover mechanism

Before concluding this tutorial, let's test the availability of our cluster by causing the current leader (`manager-1`) to fail. Once this happens, the other managers should detect the failure and elect a new leader.

You can cause the `manager-1` node to become unavailable by stopping the `docker` service on the server:

```
sudo systemctl stop docker
```

Afterward, run the `docker node ls` command on any of the other manager nodes:

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq     manager-1   Ready     Drain          Unreachable      20.10.18
uspt9qwqnzqwl78gbxc7omja7     manager-2   Ready     Drain          Leader           20.10.18
txrdxwuwjpg5jjfer3bcmtc5r *   manager-3   Ready     Drain          Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

Notice that `manager-1` is deemed unreachable and `manager-2` has been elected the new leader. If you check the NGINX service status, you'll observe that the replicas running on the worker nodes were all restarted afterward:

```
docker service ps <service_id>
```

Output

```
ID            NAME                IMAGE          NODE        DESIRED STATE   CURRENT
STATE             ERROR     PORTS
ikjo7su8ooo6   nginx_nginx.1       nginx:latest   worker-2    Running         Running 3
minutes ago
inel921owq9b    \_ nginx_nginx.1   nginx:latest   manager-2   Shutdown        Shutdown 34
minutes ago
y0y5mvj5n44r   nginx_nginx.2       nginx:latest   worker-1    Running         Running 3
minutes ago
uk18jo6wwlq6    \_ nginx_nginx.2   nginx:latest   manager-3   Shutdown        Shutdown 34
minutes ago
ijbalagf7isy   nginx_nginx.3       nginx:latest   worker-1    Running         Running 3
minutes ago
drgu007baw8z   nginx_nginx.4       nginx:latest   worker-2    Running         Running 3
minutes ago
```

When you start the `docker` service on `manager-1` once again, it will be marked as `Reachable`, but `manager-2` will remain the `Leader` of the cluster.

```
sudo systemctl start docker
```

```
docker node ls
```

Output

```
ID                            HOSTNAME    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
VERSION
9r83zto8qpqiazt6slxfkjypq *   manager-1   Ready     Drain          Reachable        20.10.18
uspt9qwqnzqwl78gbxc7omja7     manager-2   Ready     Drain          Leader           20.10.18
txrdxwuwjpg5jjfer3bcmtc5r     manager-3   Ready     Drain          Reachable        20.10.18
kaq8r9gec4t58yc9oh3dc0r2d     worker-1    Ready     Active                          20.10.18
vk1224zd81xcihgm1iis2703z     worker-2    Ready     Active                          20.10.18
```

Monitoring your Docker Swarm cluster

You've set up a highly available Docker Swarm cluster that can withstand node failures and automatically redistribute workloads. However, high availability means nothing without proper monitoring to detect issues before they impact your services.

Better Stack provides unified monitoring for your entire Swarm cluster:

- Infrastructure monitoring checks all your nodes every 30 seconds from global locations
- Collect logs from all containers across manager and worker nodes in one place
- Track container health, resource usage, and service availability
- Get instant alerts via email, SMS, or phone when nodes become unreachable
- Monitor NGINX and other services with automatic health checks
- Create status pages to communicate cluster status to users
- Incident management tools coordinate your team during outages

Instead of manually checking `docker service ps` and `docker node ls`, Better Stack gives you real-time visibility into your entire cluster. When a manager node fails like in Step 9, you'll get alerted immediately rather than discovering it later.

The platform combines infrastructure monitoring, log management, and incident response in one place, eliminating the need to piece together multiple monitoring tools for your Swarm cluster.

If you want comprehensive monitoring for your Docker Swarm setup, check out Better Stack.

Final thoughts

A highly available setup is one of the essential requirements for any production system, but building such systems used to be a tedious and complex task. As demonstrated in this tutorial, using Docker and Docker Swarm makes this task much easier and also takes fewer resources when compared to other technologies (such as Kubernetes) used to accomplish the same type of high availability setup.

There's a lot more pertaining to Docker Swarm (security, scaling, secrets management, etc) that cannot be covered in one tutorial, so ensure to check out the official Swarm guide and the rest of our scaling docker tutorial series. If you would like to read more on Docker, feel free to also explore our Docker logging guide.

Thanks for reading!

Revision #1
Created 2026-03-22 19:41:00 UTC by Administrador

Updated 2026-03-22 19:43:10 UTC by Administrador