

Docker Swarm cheat sheet

The commands to set up, run, and recover a Docker Swarm cluster, with what each one does and when to use it. Swarm mode is Docker Engine's built-in container orchestration.

[DOCKER ENGINE 29](#) · [UBUNTU SERVER 26.04 LTS](#) · [SETUP](#) · [SERVICES](#) · [RECOVERY](#) · [TROUBLESHOOTING](#)

General educational guidance, not affiliated with or endorsed by Docker, Inc. or Canonical. Try commands on a test cluster before production. The security items follow the CIS Docker Benchmark v1.8.0, section 7 (Docker Swarm Configuration).
Published 2026-09-07.

HOW A SWARM FITS TOGETHER

A swarm is a group of Docker hosts acting as one. You declare the services you want; the managers keep the cluster matched to that declaration and restart or move containers when a node fails.

```
+-- managers: keep an odd number -----+
| mgr-1 (leader)   mgr-2   mgr-3         |
+-----+-----+
|                               |
|         | schedules tasks    |
+-- workers -----+-----+
| wkr-1 [web.1] [api.1]   wkr-2 [web.2] |
+-----+-----+
client -> any node:80 -> routing mesh -> a web task
```

manager schedules work and stores the cluster state; a majority of managers has to agree (Raft) before you can change anything, so you run an odd number.

worker runs containers and holds no cluster state; lose one and its tasks restart elsewhere.

service the state you declare: image, replica count, ports, secrets, limits.

task one running container of a service. web.2 is the second task of service web.

stack a group of services deployed together from one Compose file.

INSTALL AND OPEN PORTS, UBUNTU SERVER 26.04

Run the same steps on every node, managers and workers alike.

```
# add Docker's apt repo: docs.docker.com/engine/install/ubuntu
sudo apt install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
# leave the nftables firewall backend off; Swarm needs iptables
# open to the other nodes only, never at a perimeter firewall
# <subnet> is the network the nodes reach each other on
# ufw is inactive on a fresh install and these rules do
#   nothing until you allow SSH, then run: sudo ufw enable
sudo ufw allow from <subnet> to any port 2377 proto tcp
sudo ufw allow from <subnet> to any port 7946
sudo ufw allow from <subnet> to any port 4789 proto udp
#   2377 managers; 7946 discovery, tcp and udp; 4789 overlay
#   data (VXLAN, unauthenticated)
# an encrypted overlay also needs IP protocol 50 (ESP), which
#   ufw writes only in this longer form
sudo ufw allow from <subnet> proto esp
```

These rules cover the cluster's own ports. They do not cover a port you publish with `-p`: Docker routes that traffic in the `nat` table, ahead of the chains `ufw` filters. Restrict those at the cloud security group or the load balancer in front. With neither, put the rules in the `DOCKER-USER` chain that Docker's install page names, and test them against a published port first: swarm sends that traffic through its own `DOCKER-INGRESS` chain.

CREATE AND LOCK THE CLUSTER

One init on the first manager; every other node joins with a token. Treat the token like a password.

```
# on the first manager
docker swarm init --advertise-addr <manager-ip>
# harden: --listen-addr <ip>:2377 --data-path-addr <ip2>
# print the join command, token included, for each role
docker swarm join-token worker
docker swarm join-token manager
# on each new node, paste what it printed
docker swarm join --token <token> <manager-ip>:2377
# new token; the old one no longer joins nodes, and the
# nodes already in the swarm are unaffected
docker swarm join-token --rotate worker
```

Without autolock, Docker stores the cluster state's encryption keys unencrypted on the manager's disk. With autolock on, a manager will not rejoin after a restart until someone types the unlock key, so decide who holds it and where before you turn it on: an unattended reboot leaves the cluster down until they do.

```
docker swarm update --autolock=true # save the key it prints
# a new swarm: docker swarm init --autolock starts locked
# print the current key again, on any unlocked manager
docker swarm unlock-key
docker swarm unlock # after every Docker restart, on a manager
docker swarm unlock-key --rotate # new key; update your copy
# keep the old key a few minutes: only it unlocks a manager
# that went down before the new key reached it
```

SERVICES

You create a service once; scaling, upgrades, and rollbacks are edits to its declared state, and the managers roll every edit out task by task. Run this and everything after it on a manager, unless a comment says otherwise.

```
# --name web          the name every other command uses
# --replicas 3        run three copies
# -p 80:80            reachable on port 80 of every node
# --network appnet    the overlay from Networks; without it the
#                     service joins only ingress
# nginx:stable        a moving tag; pin a version in production
docker service create --name web --replicas 3 -p 80:80 \
  --network appnet nginx:stable
```

COMMAND	WHAT IT DOES
<code>docker service ls</code>	every service; 3/3 means all replicas run
<code>docker service ps web</code>	web's tasks, the node each runs on, and errors
<code>docker service logs -f web</code>	follow the logs of all web's tasks at once; needs the json-file or journald log driver
<code>docker service inspect web --pretty</code>	the full declared state, readable

COMMAND	WHAT IT DOES
<code>docker service scale web=5</code>	change the replica count
<code>docker service rm web</code>	remove it, tasks and all

```
# rolling update: replace 2 tasks at a time, 10s apart
docker service update --image <image:tag> \
  --update-parallelism 2 --update-delay 10s web
# start each new task before stopping the old one
# (default is stop-first: the old task goes down first)
docker service update --update-order start-first web
# roll back automatically if an update fails
# the default, pause, stops the rollout half-updated
# it only fires inside --update-monitor (default 5s), so
# set that past startup and give the image a HEALTHCHECK
docker service update --update-failure-action rollback web
# put back the state before the last update
docker service update --rollback web
# recreate every task, config unchanged (rolling restart)
docker service update --force web
```

PLACEMENT AND LIMITS

Flags for service create that control which nodes run a task and how much memory and CPU it gets.

```
--mode global           # one task on every node (agents)
--constraint node.role==worker # keep off the managers
--constraint node.labels.zone==a # only nodes you labeled
--replicas-max-per-node 1 # spread copies across nodes
--reserve-memory 256M    # scheduler plans around this
--limit-memory 512M      # hard cap; over it the task is
#                        OOM-killed and restarted
--reserve-cpu 0.5        # the same pair for CPU, in cores
--limit-cpu 1
# on service update the memory flags and
# --replicas-max-per-node are the same, --constraint becomes
# --constraint-add and --constraint-rm, and --mode does not
# exist: to change the mode, remove the service and recreate
```

SECRETS AND CONFIGS

The swarm stores a secret encrypted in the cluster state and mounts it in an in-memory filesystem, in only the services you grant it to, so the value stays out of the image, the environment, and the shell history. Turn autolock on as well, or the key that encrypts it sits unencrypted on the manager's disk.

A config carries values that are not sensitive. Docker leaves a config unencrypted at rest and mounts it as an ordinary file, so keep every credential in a secret.

```
# from a file, which keeps the value out of shell history
# (stdin works too: docker secret create db_pass -, but
# an echo pipeline writes it into your shell history)
docker secret create db_pass ./db_pass.txt
# grant it; the task reads /run/secrets/db_pass
# services only: plain docker run cannot use a secret
docker service create --secret db_pass ...
# a secret cannot be edited: make a new one and swap it in
# keep the target name; the task still reads db_pass
docker service update --secret-rm db_pass \
  --secret-add source=new_pass,target=db_pass web
docker secret rm db_pass # the swap left the old one in place
docker secret ls        # docker config has ls and rm too
docker config create app_conf ./app.conf
docker service create \
  --config source=app_conf,target=/etc/app.conf ...
```

STACKS: SERVICES FROM A COMPOSE FILE

You write service flags once, under `deploy:` in a Compose file, and deploy the whole application as a stack. Underneath, a stack is the same services, so every command above still reads one. A later `docker stack deploy` puts the file's values back, so change the file rather than the running service.

```
# compose.yml
services:
  web:
    image: registry.example.com/web:1.4
    ports: ["80:80"]
    secrets: [db_pass]
    deploy:
      replicas: 3
      update_config:
        parallelism: 2
        delay: 10s
        order: start-first
      restart_policy: { condition: on-failure }
      placement: { constraints: [node.role==worker] }
secrets:
  db_pass:
    external: true    # created with docker secret create
```

```
# create, and later update, the whole stack
docker stack deploy -c compose.yml app --with-registry-auth
docker stack ls          # every stack on the cluster
docker stack services app # its services, with replica counts
docker stack ps app      # every task in the stack
docker stack rm app
```

`docker stack deploy` does not build images: build, push to a registry the nodes can reach, then deploy. Re-run it after editing the file; `--prune` also removes services you deleted.

NETWORKS AND PUBLISHING

Services on the same overlay network reach each other by name: `web` resolves to a virtual IP (VIP) that balances across web's tasks, on any node. The routing mesh publishes a port cluster-wide: every node answers on it, then forwards the request to a task.

```
# app network; --attachable lets plain docker run join it
docker network create -d overlay --attachable appnet
# encrypt task-to-task traffic between nodes (ESP); it costs
#   throughput. Swarm reports no error on a Windows node,
#   which then cannot reach Linux containers on the network
#   and sends its own traffic unencrypted
docker network create -d overlay --opt encrypted private
# the next two are docker service create flags
# publish on the task's own node only, skip the mesh
--publish published=8080,target=80,mode=host
# DNS round robin, for use behind your own load balancer;
#   cannot be combined with a port published in ingress mode
--endpoint-mode dnsrr
```

`--opt encrypted` covers only a network you create. The routing mesh runs on the ingress network, which Docker creates for you and leaves unencrypted, so traffic to every port published with `-p` crosses between nodes in cleartext.

To encrypt it, check that every node runs the same engine version, remove the services that publish ports, which takes them offline until you put them back, then run `docker network rm ingress` and create it again with `-d overlay --ingress --opt encrypted`.

The traffic that runs the cluster itself is always encrypted with mutual TLS; this note is about your application's data.

NODES

COMMAND	WHAT IT DOES
<code>docker node ls</code>	every node, its state, and which manager leads
<code>docker node ps <node></code>	the tasks running on one node
<code>docker node inspect <node> --pretty</code>	labels, resources, and engine version, readable
<code>docker node promote <node></code>	make a worker a manager
<code>docker node demote <node></code>	make a manager a worker; always demote before removing
<code>docker node update --label-add zone=a <node></code>	tag a node, for placement constraints
<code>docker node rm <node></code>	remove a node that left or failed

```
# node maintenance, in this order
docker node update --availability drain <node>
#   its service tasks move to other nodes; patch and reboot
docker node update --availability active <node>
# pause: the node keeps its tasks but takes no new ones
docker node update --availability pause <node>
# in production, keep application tasks off the managers
docker node update --availability drain <manager>
```

RETIRE A NODE, DISMANTLE THE CLUSTER

Demote a manager before it leaves. It cannot leave a live swarm without `--force`, which removes it without reconfiguring quorum.

```
# retire a node: drain it first (see Nodes), then
docker swarm leave          # on the leaving node
docker node rm <node>       # then on a manager
# dissolve: rm stacks; workers leave; demote+leave managers;
docker swarm leave --force   # last manager; close the ports
```

BACK UP AND RECOVER

Docker stores the cluster state (services, secrets, TLS certificates, membership) in `/var/lib/docker/swarm` on every manager. Back it up from one manager; if you lose every manager, that backup is the only way to rebuild the cluster.

The archive holds the swarm's root CA private key and the join tokens, so encrypt it and keep it where you keep the cluster's secrets. Restore it to a spare host every quarter and check that `docker node ls` and `docker service ls` come back.

```
# back up on a manager; with 3, one more failure loses quorum
sudo systemctl stop docker.socket docker
sudo tar -czf /root/swarm-backup.tgz -C /var/lib/docker swarm
sudo systemctl start docker
# restore on a manager: stop Docker, empty that directory,
# then unpack one level above it, since the archive holds
# swarm/. Start Docker again, then run the init below
sudo tar -xzf /root/swarm-backup.tgz -C /var/lib/docker
# (with autolock on, the restore also needs the unlock key)
docker swarm init --force-new-cluster --advertise-addr <ip>
# then re-add every manager and worker; membership is stale,
# and rotate the unlock key, which the backup still carries
# lost quorum instead? same command, and the workers stay
docker swarm ca --rotate      # new CA; join tokens change
# waits for every node; one that is down blocks it
docker swarm update --cert-expiry 720h # 30d; default 2160h
```

Run an odd number of managers: 3 tolerate the loss of 1, 5 tolerate 2. Stop there. Every manager can change the cluster and its security settings, so each extra manager is one more host worth attacking. Running services keep running while quorum is lost (a majority of managers), but you cannot scale or update them until it returns or you force a new cluster.

TROUBLESHOOTING

COMMAND	WHAT IT ANSWERS
<code>docker info</code>	is this node in a swarm, its role, manager count; run it on the node in question
<code>docker node ls</code>	a node Down or Drain? no leader elected?
<code>docker service ps web --no-trunc</code>	full task errors: pull failures, "no suitable node"
<code>docker service logs web</code>	is the application itself failing? json-file or journald driver only
<code>sudo journalctl -u docker</code>	the Docker daemon's own log (systemd); run it on the failing node

A task stuck in Pending means no node qualifies: drained, short of reserved memory, or excluded by a constraint. A service at 0/3 with a pull error means the nodes cannot fetch the image; check the registry, and pass `--with-registry-auth` on the create, update, or stack deploy that pulls it.