Docker Swarm Guide

Battle-Tested for 2026

JANUARY 10, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

V1: BATTLE-TESTED PRODUCTION KNOWLEDGE

TL;DR: I've been running Docker Swarm in production on AWS for years and I'm sharing everything I've learned - from basic concepts to advanced production configurations. This isn't theory - it's battle-tested knowledge that kept our services running through countless deployments.

What's in V1:

- Complete Swarm hierarchy explained
- VPS requirements and cost planning across providers
- DNS configuration (the #1 cause of Swarm issues)
- Production-ready compose files and multi-stage Dockerfiles
- Prometheus + Grafana monitoring stack
- Platform comparison (Portainer, Dokploy, Coolify, CapRover, Dockge)
- CI/CD versioning and deployment workflows
- GitHub repo with all configs

WHY DOCKER SWARM IN 2026?

Before the Kubernetes crowd jumps in - yes, I know K8s exists. But here's the thing: **Docker Swarm is still incredibly relevant in 2026**, especially for small-to-medium teams who want container orchestration without the complexity overhead.

Swarm advantages:

- Native Docker integration (no YAML hell beyond compose files)
- Significantly lower learning curve
- Perfect for 2-20 node clusters
- Built-in service discovery and load balancing
- Rolling updates out of the box
- Works with your existing Docker Compose files (mostly)

If you're not running thousands of microservices across multiple data centers, Swarm might be exactly what you need.

UNDERSTANDING THE DOCKER SWARM HIERARCHY

Before diving into configs, you need to understand how Swarm is organized. Think of it as concentric circles moving inward:

```
Swarm → Nodes → Stacks → Services → Tasks (Containers)
```

Swarm

The outer ring - it's your entire cluster. As long as you have one Manager node, you have a Swarm. Swarm only works with **pre-built images** - there's no `docker build` in production. Images must be pushed to a registry (Docker Hub, ECR, etc.) beforehand. This is by design - production deployments need to be fast.

Nodes

Physical or virtual hosts in your cluster. Two types:

- **Managers**: Handle cluster state and scheduling
- **Workers**: Run your containers

Pro tip: For high availability, use 3 or 5 managers (odd numbers for quorum). We run a 2-node setup (1 manager, 1 worker) which works fine but has no manager redundancy.

Stacks

Groups of related services defined in a compose file. Think of a Stack as a "deployment unit" - when you deploy a stack, all its services come up together.

Services

The workhorse of Swarm. A service manages multiple container replicas and handles:

- Load balancing between replicas
- Rolling updates

- Health monitoring
- Automatic restart on failure

Tasks

This trips people up. In Swarm terminology, a **Task = Container**. When you scale a service to 6 replicas, you have 6 tasks. The scheduler dispatches tasks to available nodes.

VPS REQUIREMENTS & COST PLANNING

Before spinning up servers, here's what you actually need. Docker Swarm is lightweight - the overhead is minimal compared to Kubernetes.

Infrastructure Presets

PRESET	NODES	LAYOUT	MIN SPECS (PER NODE)	USE CASE
Minimal	1	1 manager	1 vCPU, 1GB RAM, 25GB	Dev/testing only
Basic	2	1 manager + 1 worker	1 vCPU, 2GB RAM, 50GB	Small production
Standard	3	1 manager + 2 workers	2 vCPU, 4GB RAM, 80GB	Standard production
HA	5	3 managers + 2 workers	2 vCPU, 4GB RAM, 80GB	High availability
Enterprise	8	3 managers + 5 workers	4 vCPU, 8GB RAM, 160GB	Large scale

Why these numbers?

- **1GB RAM minimum**: Swarm itself uses ~100-200MB, but you need headroom for containers
- **3 or 5 managers for HA**: Raft consensus requires odd numbers for quorum
- **2 vCPU for production**: Single core gets bottlenecked during deployments

Approximate Monthly Costs (2025/2026)

PROVIDER	BASIC (2 NODES)	STANDARD (3 NODES)	HA (5 NODES)	NOTES
Hetzner	~€8-12	~€20-30	~€40-60	Cheapest, EU-focused
Vultr	~$12-20	~$30-50	~$60-100	Good global coverage
DigitalOcean	~$16-24	~$40-60	~$80-120	Great UX, pricier
Linode	~$14-22	~$35-55	~$70-110	Solid middle ground
AWS EC2	~$20-40	~$50-100	~$100-200	Most expensive, most features

Prices based on comparable instance types. Actual costs depend on specific configurations.

My Recommendation

For most small-to-medium teams:

1. **Start with Basic (2 nodes)** - 1 manager + 1 worker on Vultr or Hetzner
2. **Budget ~$20-40/month** for a production-ready setup
3. **Add nodes as needed** - Swarm makes scaling easy

If you need HA from day one, the **Standard (3 nodes)** preset gives you redundancy without breaking the bank at ~$30-50/month on Vultr/Hetzner.

What About AWS/GCP/Azure?

Cloud giants work fine with Swarm, but:

- **More expensive** for equivalent specs
- **More complexity** (VPCs, security groups, IAM)
- **Better if** you need other AWS services (RDS, S3, etc.)

We run Swarm on AWS EC2 because we're already deep in the AWS ecosystem. If you're starting fresh, a dedicated VPS provider is simpler and cheaper.

SETTING UP YOUR PRODUCTION ENVIRONMENT

Step 1: Install Docker (The Right Way)

On Ubuntu (tested on 20.04/22.04/24.04):

```
# Clean up any old installations
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker
    sudo apt-get remove $pkg
done

# Add Docker's official GPG key
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -c
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugir

# Add your user to docker group (logout/login after)
sudo groupadd docker
sudo usermod -aG docker $USER
```

Important: Use `docker compose` (with a space), not `docker-compose`. The latter is deprecated.

Step 2: Initialize the Swarm

This is where people mess up on AWS/cloud environments. You have multiple network interfaces, so you MUST specify the advertise address:

```
# Get your internal IP
ip addr

# Initialize Swarm on the manager (replace with your internal IP)
docker swarm init --advertise-addr 10.10.1.141:2377 --listen-addr 10.10.1.141:2
```

The output gives you a join token. Save it! Workers use this to join:

```
# On worker nodes
docker swarm join --token SWMTKN-1-xxxxx... 10.10.1.141:2377
```

Critical for HA: Use a fixed IP address for the advertise address. If the whole swarm restarts and every manager node gets a new IP address, there's no way for any node to contact an existing manager.

DNS CONFIGURATION (THIS WILL SAVE YOU HOURS OF DEBUGGING)

CRITICAL: DNS issues cause 90% of Swarm networking problems. Docker runs its own DNS server at `127.0.0.11` for container-to-container communication.

For internal service discovery (especially important on AWS), set up an internal DNS server. We use Bind9 on a dedicated host:

On Each Swarm Node

Edit `/etc/systemd/resolved.conf`:

```
[Resolve]
DNS=10.10.1.122 8.8.8.8
Domains=~yourdomain.io
```

Then reboot the node.

Why This Matters

Without proper DNS:

- Containers can't resolve other services by name
- You'll see random connection timeouts
- Round-trip to external DNS adds latency
- Service discovery breaks silently

Rule of thumb: Never hardcode IP addresses in Swarm. Services come and go - let Docker handle routing via service names.

Docker's Internal DNS (127.0.0.11)

Docker runs its own DNS server at `127.0.0.11` for container-to-container resolution. Some applications (like Postfix) need this explicitly configured:

```
# In your Dockerfile - for apps that need DNS config
RUN echo "nameserver 127.0.0.11" >> /var/spool/postfix/etc/resolv.conf
```

This is especially important for services that chroot or have their own resolv.conf handling.

NETWORK CONFIGURATION: THE SECRET SAUCE

Create a user-defined overlay network. This is **mandatory** for multi-node communication:

```
docker network create \
  --opt encrypted \
  --subnet 172.240.0.0/24 \
  --gateway 172.240.0.254 \
  --attachable \
  --driver overlay \
  awsnet
```

Let me break down these flags:

FLAG	WHY IT'S IMPORTANT
`--opt encrypted`	Enables IPsec encryption for inter-node traffic. Optional but recommended for security. **Note:** Can cause issues in AWS/cloud with NAT - use internal VPC IPs if you enable this
`--subnet`	Prevents conflicts with AWS VPC ranges and default Docker networks
`--attachable`	Allows standalone containers (like monitoring agents) to connect
`--driver overlay`	Required for Swarm networking across nodes

Pro tip: If you're using Postfix for email relay, whitelist your Docker subnet (e.g., `172.240.0.0/24`) in the relay configuration.

Required Ports for Swarm Communication

Ensure these ports are open between nodes:

- **TCP 2377**: Cluster management communications
- **TCP/UDP 7946**: Communication among nodes
- **TCP/UDP 4789**: Overlay network traffic

THE COMPOSE FILE DEEP DIVE

Here's a production-ready compose file with explanations:

```
version: "3.8"

services:
  nodeserver:
    # ALWAYS specify your DNS server for internal resolution
    dns:
      - 10.10.1.122

    # Use init for proper signal handling and zombie process cleanup
    init: true

    environment:
      - NODE_ENV=production
      # Reference .env variables with ${VAR}
      - API_KEY=${API_KEY}
      - NODE_OPTIONS=--max-old-space-size=300

    deploy:
      mode: replicated
      replicas: 6

      placement:
        # Spread across nodes - max 3 per node means 6 replicas need 2+ nodes
        max_replicas_per_node: 3

      update_config:
        # Rolling updates: 2 at a time with 10s delay
        parallelism: 2
        delay: 10s
        # Rollback on failure
        failure_action: rollback
        order: start-first  # New containers start before old ones stop

      rollback_config:
        parallelism: 2
        delay: 10s

      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s

      resources:
        limits:
          cpus: '0.50'
          memory: 400M
        reservations:
          cpus: '0.20'
          memory: 150M

    # Build is IGNORED in Swarm - image must be pre-built
    build:
      context: ./nodeserver

    image: "yourregistry/nodeserver:latest"

    # Only container port - Docker handles host port assignment
    ports:
      - "61339"

    volumes:
      - nodeserver-logs:/var/log

    networks:
      awsnet:

    secrets:
      - app_secrets

# Must declare volumes at root level
volumes:
  nodeserver-logs:

# External secrets (created in Portainer or via CLI)
secrets:
  app_secrets:
    external: true

# External network (pre-created with our custom config)
networks:
  awsnet:
    external: true
    name: awsnet
```

Key Deploy Settings Explained

Parallelism & Updates:

```
update_config:
  parallelism: 2
  delay: 10s
```

With 6 replicas and parallelism of 2, Swarm updates 2 containers at a time. If they come up healthy, it proceeds to the next 2. This ensures zero downtime and automatic rollback if the new image fails.

Resource Limits:

```
resources:
  limits:
    cpus: '0.50'
    memory: 400M
```

Always set these! Without limits, a misbehaving container can starve the entire node.

Init Process:

```
init: true
```

This runs a tiny init system (tini) as PID 1. It handles:

- Signal forwarding to your application
- Zombie process reaping
- Proper shutdown sequences

Without this, orphaned processes accumulate and `SIGTERM` might not reach your app.

DOCKERFILE BEST PRACTICES FOR SWARM

Since Swarm only works with pre-built images, your Dockerfile quality matters even more. Let me share real production Dockerfiles I've refined over years.

Multi-Stage Builds (The Right Way)

Multi-stage builds keep your final image small and secure. Here's a standard Node.js example:

```
# syntax=docker/dockerfile:1

# ============================================
# STAGE 1: Base with build dependencies
# ============================================
FROM node:20-bookworm-slim AS base

WORKDIR /app

# Install build tools needed for native npm packages
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    python3 \
    make \
    g++ \
    && apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Copy package files first (layer caching optimization)
COPY package.json package-lock.json ./

# ============================================
# STAGE 2: Install dependencies
# ============================================
FROM base AS compiled

RUN npm ci --omit=dev

# ============================================
# STAGE 3: Final production image
# ============================================
FROM node:20-bookworm-slim AS final

# Set timezone
RUN ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime \
    && echo America/New_York > /etc/timezone

WORKDIR /app

# Copy ONLY the compiled node_modules from build stage
COPY --from=compiled /app/node_modules /app/node_modules

# Copy application code
COPY . .

EXPOSE 3000
```

```
ENTRYPOINT ["node", "--trace-warnings", "./server.js"]
```

Why multi-stage?

- Build tools (python3, make, g++) stay in the `base` stage
- Final image is clean `node:20-bookworm-slim` without build dependencies
- Significantly smaller image size
- Security: No compilers/build tools for attackers to exploit

Hybrid Python + Node.js Dockerfile

Sometimes you need both Python and Node.js - for example, when your app requires Chromium for PDF generation, Python-based build tools, or data processing scripts. Here's a real production example:

```
# syntax=docker/dockerfile:1

# ============================================
# STAGE 1: Python base with Node.js installed
# ============================================
FROM python:bookworm AS base

LABEL org.opencontainers.image.authors="Your Name <you@example.com>"

# Create non-root user
RUN groupadd --gid 1000 node \
  && useradd --uid 1000 --gid node --shell /bin/bash --create-home node

# Install Node.js, Yarn, and Python tools
RUN \
  echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg] https://deb.nodesource
  wget -qO- https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | gpg --c
  apt-get update && \
  apt-get install -y --no-install-recommends \
  chromium \
  nodejs && \
  pip install -U pip && pip install pipenv && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*

# Set timezone
RUN ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime \
    && echo America/New_York > /etc/timezone

WORKDIR /app

COPY package.json package-lock.json ./

# ============================================
# STAGE 2: Install Node dependencies
# ============================================
FROM base AS compiled

RUN npm install --omit=dev

# ============================================
# STAGE 3: Final production image
# ============================================
FROM base AS final

# Copy compiled node_modules from build stage
```

```
COPY --from=compiled /app/node_modules /app/node_modules

# Copy application code
COPY . .

EXPOSE 3000

ENTRYPOINT ["node", "--trace-warnings", "./server.js"]
```

When to use Python + Node hybrid:

- PDF generation with Puppeteer/Chromium
- Data processing pipelines mixing Python and Node
- Build systems requiring Python tools (Poetry, pipenv)
- ML/AI features alongside a Node.js web server

Nginx with ModSecurity WAF

Here's a real Nginx Dockerfile with ModSecurity WAF compiled in:

```
# syntax=docker/dockerfile:1
ARG NGINX_VERSION=1.27.0

FROM nginx:$NGINX_VERSION as base

# Install build dependencies
RUN apt update && \
    apt install -y git dos2unix apt-utils autoconf automake \
    build-essential libcurl4-openssl-dev libgeoip-dev \
    liblmdb-dev libpcre3 libpcre3-dev libtool libxml2-dev \
    libyajl-dev pkgconf wget tar zlib1g-dev && \
    ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime

# Clone and build ModSecurity
RUN git clone --depth 1 -b v3/master --single-branch https://github.com/SpiderL

WORKDIR /ModSecurity

RUN git submodule init && git submodule update && \
    ./build.sh && ./configure && make && make install

# Build Nginx ModSecurity module
RUN git clone --depth 1 https://github.com/SpiderLabs/ModSecurity-nginx.git &&
    wget http://nginx.org/download/nginx-$NGINX_VERSION.tar.gz && \
    tar zxvf nginx-$NGINX_VERSION.tar.gz

WORKDIR /ModSecurity/nginx-$NGINX_VERSION

RUN ./configure --with-compat --add-dynamic-module=../ModSecurity-nginx && \
    make modules && \
    cp objs/ngx_http_modsecurity_module.so /usr/lib/nginx/modules

# ============================================
# Final stage - clean image with just the module
# ============================================
FROM nginx:$NGINX_VERSION AS final

# Copy the compiled module from build stage
COPY --from=base /usr/lib/nginx/modules/ngx_http_modsecurity_module.so /usr/lib
COPY --from=base /usr/local/modsecurity/ /usr/local/modsecurity/

COPY nginx/ /etc/nginx/

RUN mkdir -p /var/cache/nginx_cache && \
    ln -s /etc/nginx/sites-available/* /etc/nginx/sites-enabled/
```

```
EXPOSE 80 443
```

The key insight: **Build ModSecurity in a temp stage, copy only the compiled `.so` module to the final image.**

Key Dockerfile Rules:

1. Always Run in Foreground

```
# When building nginx from a base OS image (debian, ubuntu, etc.):
# WRONG - daemon mode, container exits immediately
CMD ["nginx"]

# RIGHT - foreground mode
CMD ["nginx", "-g", "daemon off;"]

# NOTE: The official nginx Docker image already includes "daemon off;"
# so you don't need to specify it when using FROM nginx:x.x.x

# For Postfix:
CMD ["/usr/sbin/postfix", "start-fg"]
```

Containers need a foreground process to stay alive. If your process daemonizes, the container thinks "my job is done" and exits.

2. Handle Cross-Platform Line Endings

If you develop on Windows but deploy to Linux, line endings can break everything:

```
# Install dos2unix and convert files
RUN apt-get install -y dos2unix && \
    dos2unix /etc/myapp/config.conf && \
    dos2unix /scripts/entrypoint.sh
```

This has saved me hours of debugging "file not found" errors that were actually `\r\n` vs `\n` issues.

3. Pin Your Base Images

```
# BAD - "latest" changes without warning
FROM ubuntu:latest

# BETTER - version pinned
FROM ubuntu:22.04

# BEST - SHA pinned (immutable)
FROM ubuntu@sha256:abc123...
```

4. Include Health Checks

```
HEALTHCHECK --interval=30s --timeout=10s --start-period=60s --retries=3 \
    CMD curl -f http://localhost/health || exit 1
```

Swarm uses health checks for:

- Deciding when a container is ready for traffic
- Triggering restarts on failure
- Rolling update decisions

5. Use .dockerignore

Create a `.dockerignore` file to exclude sensitive/unnecessary files:

```
# Secrets - NEVER include in images
**/secrets.json
**/.env
**/sasl_passwd

# Development files
**/.git
**/.gitignore
**/node_modules
**/*.log
**/npm-debug.log

# Source maps (if you don't need debugging)
**/*.js.map

# IDE files
**/.vscode
**/.idea

# Test files
**/test
**/tests
**/__tests__
```

This keeps your images small and prevents accidental secret exposure.

ADVANCED COMPOSE PATTERNS

Build Cache Optimization

Speed up builds with `cache_from`:

```
build:
  context: ./aws-nodeserver
  cache_from:
    - "yourregistry/nodeserver:latest"
  args:
    - BUILD_VERSION=${BUILD_VERSION}
    - GIT_COMMIT=${LONG_COMMIT}
image: "yourregistry/nodeserver:${DOCKER_BUILD_VERSION:-latest}"
```

Docker will use layers from the cached image when possible. This can cut build times from 10 minutes to 30 seconds.

Environment Variable Defaults

Use `${VAR:-default}` syntax for fallback values:

```
environment:
  - NGINX_RESOLVER=${NGINX_RESOLVER:-127.0.0.11}
  - NODE_ENV=${NODE_ENV:-production}

image: "yourregistry/app:${DOCKER_BUILD_VERSION:-latest}"
```

Placement Constraints

Control where services run:

```
deploy:
  placement:
    # Run only on workers (not managers)
    constraints: [node.role == worker]

    # Or only on managers
    constraints: [node.role == manager]

    # Or specific nodes by label
    constraints:
      - node.role == manager
      - node.labels.monitoring == true
```

Useful for:

- Running databases only on nodes with SSDs
- Keeping CPU-intensive work off the manager
- Pinning monitoring services to labeled nodes

To add a label to a node:

```
docker node update --label-add monitoring=true docker1.yourdomain.io
```

Global Mode Deployment

For monitoring agents that need to run on **every node**:

```
cadvisor:
  image: gcr.io/cadvisor/cadvisor:v0.47.0
  deploy:
    mode: global  # Runs ONE instance on EVERY node
    resources:
      limits:
        memory: 128M
      reservations:
        memory: 64M
```

Use `mode: global` for:

- Monitoring agents (cAdvisor, node-exporter)
- Log collectors
- Security agents
- Anything that needs host-level access on all nodes

Full Rollback Configuration

Production-ready update and rollback config:

```
deploy:
  rollback_config:
    parallelism: 1
    delay: 20s
    monitor: 10s  # Watch for this long before considering update successful
  update_config:
    parallelism: 2
    delay: 70s
    failure_action: rollback  # Auto-rollback on failure
  restart_policy:
    condition: on-failure
    delay: 70s
    max_attempts: 30  # Higher for production stability
    window: 120s
```

Key settings:

- `monitor: 10s` - How long to watch the new container before proceeding
- `failure_action: rollback` - Automatically rollback if update fails
- `max_attempts: 30` - More retries for transient failures in production
- `delay: 70s` - Longer delays give services time to stabilize

Docker Configs (Non-Sensitive Configuration)

Secrets are for sensitive data. **Configs** are for non-sensitive configuration files:

```
services:
  nginx:
    configs:
      - nginx_blocked_ips
    # ...

configs:
  nginx_blocked_ips:
    external: true  # Created in Portainer or via CLI
```

Create a config:

```
docker config create nginx_blocked_ips ./blockips.conf
```

Configs appear in the container at `/config_name` by default, or you can specify a path:

```
configs:
  - source: nginx_blocked_ips
    target: /etc/nginx/conf.d/blocked_ips.conf
    mode: 0440
```

Use configs for:

- Nginx config snippets (blocked IPs, rate limits)
- Application config files
- Feature flags
- Anything non-sensitive that changes independently of the image

Long-Form Volume Syntax

For more control over mounts, use the long-form syntax:

```
volumes:
  # Named volume (Docker-managed)
  - type: volume
    source: grafana-data
    target: /var/lib/grafana

  # Bind mount (host path)
  - type: bind
    source: /docker/swarm/aws-nginx
    target: /var/log

  # Read-only system mounts (for monitoring)
  - type: bind
    source: /proc
    target: /host/proc
    read_only: true
```

Host Path Volumes for Persistent Logs

For logs that need to survive container restarts AND be accessible from the host:

```
volumes:
  # Docker volume (isolated, Docker-managed)
  - aws-nginx:/var/log

  # Host path (accessible from host, persists across deploys)
  - /docker/swarm/aws-nginx:/var/log
```

Host path volumes are useful when:

- You need to access logs from the host for shipping
- External tools need to read container logs
- You want logs to survive `docker system prune`

Network Share Volumes (CIFS/SMB)

Mount a Windows/Samba network share as a Docker volume:

```
docker volume create \
  --driver local \
  --opt type=cifs \
  --opt device=//nas-server/share-name \
  --opt o=username=USER,password=PASS,domain=DOMAIN,uid=1000,gid=1000 \
  my-network-volume
```

Then use it in your compose file:

```
volumes:
  my-network-volume:
    external: true
```

Use cases:

- Shared storage across multiple Swarm nodes
- Accessing existing NAS storage
- Shared uploads/exports directories

Note: For production, use Docker secrets or environment variables for credentials instead of hardcoding them.

Ulimits for Memory-Hungry Services

Elasticsearch and similar services need memory locking:

```
elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:8.8.0
  ulimits:
    memlock:
      soft: -1
      hard: -1
  deploy:
    resources:
      limits:
        memory: 4096M
      reservations:
        memory: 1024M
```

Health Checks in Compose

```
services:
  visualizer:
    image: yandeu/visualizer:dev
    healthcheck:
      test: curl -f http://localhost:3500/healthcheck || exit 1
      interval: 10s
      timeout: 2s
      retries: 3
      start_period: 5s
```

COMPLETE MONITORING STACK (PROMETHEUS + GRAFANA)

Here's a production-ready monitoring stack for Docker Swarm:

```
version: "3.8"

services:
  grafana:
    image: portainer/template-swarm-monitoring:grafana-9.5.2
    ports:
      - target: 3000
        published: 3000
        protocol: tcp
        mode: ingress
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == manager
          - node.labels.monitoring == true
    volumes:
      - type: volume
        source: grafana-data
        target: /var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_USER=${GRAFANA_USER}
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
      - GF_USERS_ALLOW_SIGN_UP=false
    networks:
      - monitoring

  prometheus:
    image: portainer/template-swarm-monitoring:prometheus-v2.44.0
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--log.level=error'
      - '--storage.tsdb.path=/prometheus'
      - '--storage.tsdb.retention.time=7d'
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == manager
          - node.labels.monitoring == true
    volumes:
      - type: volume
```

```
        source: prometheus-data
        target: /prometheus
    networks:
      - monitoring

  # Container metrics - runs on ALL nodes
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.47.0
    command: -logtostderr -docker_only
    deploy:
      mode: global  # One instance per node
      resources:
        limits:
          memory: 128M
        reservations:
          memory: 64M
    volumes:
      - type: bind
        source: /
        target: /rootfs
        read_only: true
      - type: bind
        source: /var/run
        target: /var/run
        read_only: true
      - type: bind
        source: /sys
        target: /sys
        read_only: true
      - type: bind
        source: /var/lib/docker
        target: /var/lib/docker
        read_only: true
      - type: bind
        source: /dev/disk
        target: /dev/disk
        read_only: true
    networks:
      - monitoring

  # Host metrics - runs on ALL nodes
  node-exporter:
    image: prom/node-exporter:v1.5.0
    command:
      - '--path.sysfs=/host/sys'
      - '--path.procfs=/host/proc'
      - '--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)(
      - '--no-collector.ipvs'
    deploy:
      mode: global  # One instance per node
      resources:
        limits:
          memory: 128M
        reservations:
          memory: 64M
    volumes:
      - type: bind
        source: /
        target: /rootfs
        read_only: true
      - type: bind
        source: /proc
        target: /host/proc
        read_only: true
      - type: bind
        source: /sys
        target: /host/sys
        read_only: true
    networks:
      - monitoring

volumes:
  grafana-data:
  prometheus-data:

networks:
  monitoring:
    driver: overlay
```

What each service does:

SERVICE	PURPOSE	MODE
Grafana	Visualization dashboards	1 replica on manager
Prometheus	Metrics collection & storage	1 replica on manager
cAdvisor	Container resource metrics	Global (all nodes)
Node Exporter	Host system metrics	Global (all nodes)

Setup steps:

1. Label your monitoring node: `docker node update --label-add monitoring=true docker1`
2. Deploy: `docker stack deploy -c monitoring.yaml monitoring`
3. Access Grafana at `http://your-manager:3000`

This gives you visibility into CPU, memory, disk, and network usage for both containers and hosts.

MODULAR COMPOSE FILES WITH EXTENDS

> *Important: The `extends` keyword only works with `docker compose up` for local development. It does **not** work with `docker stack deploy`. For Swarm deployments, use multiple `-c` flags instead: `docker stack deploy -c base.yml -c production.yml mystack`*

For large projects in **local development**, split your compose files and use `extends`:

```yaml
# docker-compose.yaml (main file)
version: "3.8"
services:
  nginx:
    extends:
      file: docker-compose_nginx.yaml
      service: nginx

  nodeserver:
    extends:
      file: docker-compose_node.yaml
      service: nodeserver

  mailserver:
    extends:
      file: docker-compose_mail.yaml
      service: mailserver

# Shared definitions
volumes:
  nginx-logs:
  nodeserver-logs:
  mailserver-logs:

networks:
  awsnet:
    external: true
    name: awsnet

secrets:
  # File-based (for development)
  app_secrets:
    file: ./.secrets/secrets.json

  # SSL certificates
  nginx_server_pem:
    file: ./.secrets/ssl/server.pem
  nginx_server_key:
    file: ./.secrets/ssl/server.key
```

Then each service has its own compose file:

```
# docker-compose_node.yaml
version: "3.8"
services:
  nodeserver:
    image: yourregistry/nodeserver:${VERSION:-latest}
    dns:
      - 10.10.1.122
    init: true
    # ... full service definition
```

Benefits:

- Easier to manage large stacks
- Teams can own their service configs
- Cleaner git diffs
- Reusable service definitions

SECRET MANAGEMENT (STOP USING ENVIRONMENT VARIABLES!)

Docker secrets are encrypted at rest and in transit. They appear as files in `/run/secrets/SECRET_NAME`.

Development vs Production Secrets

```
secrets:
  app_secrets:
    # DEVELOPMENT: Load from local file
    file: ./.secrets/secrets.json

  app_secrets_prod:
    # PRODUCTION: Reference pre-created secret
    external: true
```

```
# Create external secret for production
docker secret create app_secrets ./secrets.json

# Or via Portainer's GUI
```

Creating Secrets Properly

Method 1: From a file (common but has risks)

```
docker secret create my_secret ./secret.txt
```

Risk: The file still exists on disk. Delete it after creating the secret, or use Method 2.

Method 2: From stdin (more secure)

```
# Single value
echo "my_super_secret_password" | docker secret create db_password -

# Or use printf to avoid newline issues
printf "my_api_key_here" | docker secret create api_key -

# From password manager or environment (never type secrets in shell history)
cat /dev/stdin | docker secret create api_key -
# Then paste and press Ctrl+D
```

Why stdin? No file on disk, no shell history (if you pipe from another command).

Method 3: Multi-line secrets (JSON, certificates, etc.)

```
# JSON config
cat << 'EOF' | docker secret create app_config -
{
  "database": "mongodb://...",
  "api_key": "sk-...",
  "jwt_secret": "..."
}
EOF

# Or from existing file, then delete
docker secret create ssl_cert ./cert.pem && rm ./cert.pem
```

Managing Secrets

```
# List all secrets
docker secret ls

# Inspect secret metadata (NOT the value - that's the point!)
docker secret inspect my_secret

# See which services use a secret
docker service inspect --format '{{json .Spec.TaskTemplate.ContainerSpec.Secret

# Delete a secret (must not be in use)
docker secret rm my_secret
```

Updating Secrets (They're Immutable!)

Common mistake: Trying to update a secret in place. Docker secrets are **immutable** - you can't change them.

The correct workflow:

```
# 1. Create new secret with versioned name
echo "new_password_value" | docker secret create db_password_v2 -

# 2. Update your compose file to reference the new secret
# secrets:
#   - db_password_v2   # was: db_password

# 3. Redeploy the stack
docker stack deploy -c docker-compose.yml mystack

# 4. Remove old secret (once no services use it)
docker secret rm db_password
```

Pro tip: Use a naming convention like `secret_name_v1`, `secret_name_v2` or `secret_name_20260116` for easier rotation tracking.

Common Mistakes to Avoid

MISTAKE	WHY IT'S BAD	FIX
Creating from file, not deleting file	Secret sits on disk in plaintext	Use stdin or delete file immediately
Putting secret in shell command	Saved in `.bash_history`	Pipe from stdin or use `read -s`
Using same secret across environments	Compromised staging = compromised production	Separate secrets per environment
Not versioning secrets	Can't rollback if new secret breaks things	Use `_v1`, `_v2` suffix
Committing `.secrets/` folder	Secrets end up in git history forever	Add to `.gitignore` FIRST

Multiple Secret Types Example

```
secrets:
  # Application secrets
  gg_secrets:
    file: ./.secrets/secrets.json

  # Mail server credentials
  mail_sasl_passwd:
    file: ./.secrets/mail_sasl_passwd

  # SSL certificates (yes, these can be secrets!)
  nginx_dhparams_pem:
    file: ./.secrets/nginx_ssl_certificates/dhparams.pem
  nginx_server_pem:
    file: ./.secrets/nginx_ssl_certificates/server.pem
  nginx_server_key:
    file: ./.secrets/nginx_ssl_certificates/server.key
```

In your Dockerfile, set proper permissions:

```
# For sensitive files like SASL passwords
RUN chown root:root /etc/postfix/sasl_passwd && \
    chmod 0600 /etc/postfix/sasl_passwd
```

In your application, read `/run/secrets/app_secrets` instead of using env vars for sensitive data.

Why secrets > env vars:

- Not visible in `docker inspect`
- Not in image layers
- Encrypted in the Raft log
- Only sent to nodes that need them
- Proper file permissions can be set

Why Environment Variables Aren't Actually "Safe"

Common misconception: "It's not hardcoded, it's an environment variable, so it's safe."

Reality: Any process running inside the container can read environment variables. A compromised dependency, a debug endpoint, a log statement that dumps `process.env`, or a memory dump can expose them all.

Better approach - use env vars to point to secret files:

```
// BAD: Secret value directly in environment
// const apiKey = process.env.API_KEY

// GOOD: Environment variable points to a filename
const fs = require('fs');

function getSecret(secretName) {
  // Check for _FILE suffix convention, fallback to Docker secrets path
  const secretPath = process.env[`${secretName}_FILE`] || `/run/secrets/${secre
  return fs.readFileSync(secretPath, 'utf8').trim();
}

// Usage
const apiKey = getSecret('API_KEY');
const dbPassword = getSecret('DB_PASSWORD');
```

This pattern:

1. **Adds a layer of abstraction** - even if env vars leak, attackers only get file paths
2. **Works with Docker secrets** - reads from `/run/secrets/` by default

3. **Supports the `_FILE` convention** - used by many official Docker images (MySQL, PostgreSQL, etc.)
4. **Keeps secrets out of process memory dumps** - secret is read on-demand, not stored in `process.env`

In your compose file:

```
environment:
  - API_KEY_FILE=/run/secrets/api_key
  - DB_PASSWORD_FILE=/run/secrets/db_password
secrets:
  - api_key
  - db_password
```

DOCKER MANAGEMENT & DEPLOYMENT PLATFORMS

Managing Docker Swarm via CLI is powerful, but GUI tools can significantly improve visibility and reduce operational overhead. Here's a comparison of the top platforms in 2026.

Portainer

What it is: Container management UI for Docker, Docker Swarm, and Kubernetes.

Best for: Teams wanting visual management without changing their workflow.

Swarm Support: Full native support

Key Features:

- Visual stack/service management
- Built-in templates for common deployments
- User management and RBAC
- Real-time container logs and stats
- Secret and config management via GUI

Installation:

```
# Deploy Portainer agent on each Swarm node
docker service create \
  --name portainer_agent \
  --publish mode=host,target=9001,published=9001 \
  --mode global \
  --mount type=bind,src=//var/run/docker.sock,dst=/var/run/docker.sock \
  --mount type=bind,src=//var/lib/docker/volumes,dst=/var/lib/docker/volumes \
  portainer/agent:latest

# Deploy Portainer server on manager
docker service create \
  --name portainer \
  --publish 9443:9443 \
  --publish 8000:8000 \
  --replicas=1 \
  --constraint 'node.role == manager' \
  --mount type=volume,src=portainer_data,dst=/data \
  portainer/portainer-ce:latest
```

Pricing: Portainer CE (Community Edition) is completely free with no node limits. Business Edition adds enterprise features, support, and RBAC.

Dokploy

What it is: Self-hosted PaaS alternative to Heroku/Vercel/Netlify, built on Docker and Traefik.

Best for: Teams wanting push-to-deploy workflows with Docker Swarm clustering.

Swarm Support: Full - uses Docker Swarm for clustering

Key Features:

- Git-based deployments (GitHub, GitLab, Bitbucket)
- Automatic SSL via Let's Encrypt
- Built-in Traefik for routing
- Docker Compose support
- Server/service metrics out of the box
- Volume backups to S3

- Multi-server clustering via SSH

Installation:

```
curl -sSL https://dokploy.com/install.sh | sh
```

Pricing: Free self-hosted, $4.50/month managed option

Limitations:

- Documentation lags behind development
- UI for Swarm node management is still maturing
- Requires external registry for multi-node deployments

Coolify

What it is: Open-source, self-hostable PaaS with 280+ one-click templates.

Best for: Developers wanting a polished Heroku-like experience with maximum flexibility.

Swarm Support: Experimental

Key Features:

- 280+ one-click application templates
- Remote build server support (offload builds)
- Multi-server deployments
- Automatic SSL certificates
- Git integration with PR previews
- Beautiful, intuitive UI
- Self-healing deployments

Installation:

```
curl -fsSL https://cdn.coollabs.io/coolify/install.sh | bash
```

Pricing: Free self-hosted, $4/month/server managed

Limitations:

- Docker Swarm support is experimental
- SSH configuration can be tricky with strict firewalls
- Multi-server = multiple instances, not true clustering

CapRover

What it is: Battle-tested PaaS built natively on Docker Swarm with automatic Nginx load balancing.

Best for: Teams wanting proven Swarm-based PaaS with one-click apps.

Swarm Support: Full - native Swarm architecture

Key Features:

- Native Docker Swarm clustering
- Built-in Nginx load balancing
- One-click apps marketplace
- Free SSL with Let's Encrypt
- Docker Compose and Dockerfile support
- CLI and web dashboard

Installation:

```
# On manager node
docker run -p 80:80 -p 443:443 -p 3000:3000 \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /captain:/captain \
  caprover/caprover
```

Pricing: Free, open-source

Limitations:

- UI can feel dated

- Documentation could be better
- Less active development than Coolify/Dokploy

Dockge

What it is: Lightweight Docker Compose stack manager with beautiful UI.

Best for: Simple Compose management in home labs or single-server setups.

Swarm Support: None - Docker Compose only

Key Features:

- Clean, modern UI
- Real-time log viewer
- Direct compose.yaml editing
- Interactive container terminal
- Lightweight (minimal resources)

Installation:

```
mkdir -p /opt/stacks /opt/dockge
cd /opt/dockge
curl -O https://raw.githubusercontent.com/louislam/dockge/master/compose.yaml
docker compose up -d
```

Limitations:

- No Docker Swarm support
- Single-server only
- No built-in SSL or routing

Platform Comparison Matrix

FEATURE	PORTAINER	DOKPLOY	COOLIFY	CAPROVER	DOCKGE
Swarm Support	Full	Full	Experimental	Full	None
Multi-Node	Yes	Yes	Yes	Yes	No
Git Deploy	No	Yes	Yes	Yes	No
Auto SSL	No	Yes	Yes	Yes	No
One-Click Apps	Templates	Limited	280+	Yes	No
Traefik Built-in	No	Yes	Yes	No (Nginx)	No
Volume Backups	No	S3	Limited	No	No
Resource Usage	Medium	Medium	Medium-High	Medium	Low
Learning Curve	Low	Low	Medium	Low	Very Low

Multiple instances, not true clustering

Recommendations

For Production Swarm Clusters:

1. **Portainer** - If you want visibility without changing workflows
2. **Dokploy** - If you want Heroku-style deployments on Swarm
3. **CapRover** - If you want proven, native Swarm PaaS

For Home Labs / Small Teams:

1. **Coolify** - Best templates and UI
2. **Dockge** - Lightest weight for simple Compose

Combination I Use:

- **Portainer** for visibility and management
- **Custom CI/CD** for deployments (see CI/CD section)
- **Prometheus + Grafana** for monitoring

USEFUL COMMANDS CHEATSHEET

Node Management

```
# List all nodes
docker node ls

# Take node offline for maintenance
docker node update --availability=drain docker2.yourdomain.io

# Bring node back online
docker node update --availability=active docker2.yourdomain.io

# Force service rebalancing after adding node back
docker service update --force nodeapp
```

Stack Operations

```
# Deploy/update a stack
docker stack deploy -c docker-compose.yml mystack

# List stacks
docker stack ls

# View stack services
docker stack services mystack

# View tasks (containers) in a stack
docker stack ps mystack
```

Service Operations

```
# Scale a service
docker service scale mystack_web=4

# View service logs
docker service logs -f mystack_web

# Force update (repull image & redeploy)
docker service update --force mystack_web

# Inspect service details
docker service inspect mystack_web
```

Cleanup

```
# Remove unused images, containers, networks
docker system prune

# View resource usage
docker stats
```

LOAD BALANCING WITH NGINX

Run Nginx as a Swarm service for:

- SSL termination (or use AWS ALB)
- Static asset caching
- Reverse proxy to your app service
- Rate limiting and WAF

```
nginx:
  image: "yourregistry/nginx:latest"
  deploy:
    replicas: 2
    placement:
      max_replicas_per_node: 1  # One per node for redundancy
  ports:
    - "80:80"
    - "443:443"
  networks:
    awsnet:
```

Nginx proxies to your app using the service name:

```
upstream app {
    server mystack_nodeserver:61339;
}
```

Docker's internal DNS resolves `mystack_nodeserver` to all healthy replicas, and you get round-robin load balancing for free.

COMMON GOTCHAS & TROUBLESHOOTING

Problem: Containers can't communicate between nodes

Solution:

1. Verify the overlay network exists and is attached to both services
2. Check DNS config in `/etc/systemd/resolved.conf`
3. Ensure required ports are open (TCP/UDP 7946, UDP 4789)
4. If using `--opt encrypted`, ensure Protocol 50 (ESP) is allowed and you're using internal VPC IPs

Problem: Service stuck in "Pending" state

Solution:

```
docker service ps myservice --no-trunc
```

Usually it's resource constraints - the scheduler can't find a node with enough CPU/memory.

Problem: Node shows "Down" after reboot

Solution:

```
docker node ls
# Remove the duplicate/stale node entry
docker node rm <stale_node_id>
```

Problem: Portainer Agent disconnected

Solution: Remove and recreate the agent service:

```
docker service rm portainer_agent
# Re-run your portainer agent setup script
```

Problem: Rolling update hangs

Solution: Check health checks aren't too strict. Temporarily loosen them or increase `start_period`:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 60s  # Grace period for startup
```

Problem: Need to debug network traffic

Solution: Use `tcpdump` to inspect traffic on the host:

```
# Filter traffic on port 80 and show the real client IP
tcpdump -A 'port 80' | grep realip

# Watch all traffic to a specific service port
tcpdump -i any port 61339

# Capture to file for later analysis
tcpdump -i any -w capture.pcap port 80

# Watch DNS queries (useful for debugging service discovery)
tcpdump -i any port 53
```

Problem: Container can't resolve service names

Solution: Check Docker's internal DNS is working:

```
# From inside a container
nslookup myservice
dig myservice

# Check /etc/resolv.conf in container - should show 127.0.0.11
cat /etc/resolv.conf
```

FINAL TIPS

1. **Use Portainer** - It's free for small deployments and makes Swarm management so much easier

1. **Always use external networks** - Create them before deploying stacks so you control the configuration

1. **Tag your images properly** - Never use `latest` in production. Use commit hashes or semantic versions

1. **Set resource limits** - A container without limits can kill your entire node

1. **Test your rollback** - Deploy a broken image intentionally to see rollback work

1. **Document everything** - Your future self will thank you

1. **Take snapshots** - Before major changes, snapshot your nodes (if on cloud)

1. **Separate dev and prod configs** - Use different compose files (see below)

DEVELOPMENT VS PRODUCTION CONFIGURATIONS

Keep separate compose files for dev and prod. Here's how I structure it:

Development (`docker-compose_dev.yaml`)

```
version: "3.8"
services:
  app:
    environment:
      - NODE_ENV=development
      - LOG_LEVEL=debug
    restart: always
    deploy:
      replicas: 1  # Single replica for dev
    networks:
      network:
        ipv4_address: 10.5.0.10  # Static IP for dev debugging

networks:
  network:
    driver: bridge  # Bridge network for single-host dev
    ipam:
      config:
        - subnet: 10.5.0.0/16
          gateway: 10.5.0.1
```

Production (`docker-compose_node.yaml`)

```
version: "3.8"
services:
  app:
    dns:
      - 10.10.1.122  # Internal DNS
    environment:
      - NODE_ENV=production
      - LOG_LEVEL=info
    deploy:
      mode: replicated
      replicas: ${NODESERVER_REPLICAS}  # Variable replicas
      placement:
        max_replicas_per_node: ${MAX_NODESERVER_REPLICAS_PER_NODE}
      # ... full deploy config
    networks:
      awsnet:  # Overlay network for multi-host

networks:
  awsnet:
    external: true  # Pre-created overlay
    name: awsnet
```

Key differences:

ASPECT	DEVELOPMENT	PRODUCTION
Network	Bridge (single host)	Overlay (multi-host)
Replicas	1	Variable via env
Container names	Static	Dynamic
Debug logging	Enabled	Disabled
Resource limits	Relaxed	Strict
DNS	Default	Internal DNS server

CI/CD VERSIONING & DEPLOYMENT WORKFLOW

Getting versioning right is crucial for debugging production issues. Here's how to set up proper version tracking.

The Version File Approach

Create a version file that gets updated by your build pipeline:

```
# /path/to/project/globals/_versioning/buildVersion.txt
1.2.45
```

Your CI/CD script reads this and passes it to Docker:

```
# pushToProduction.sh
#!/bin/bash

# Read version from file
BUILD_VERSION=$(cat ./globals/_versioning/buildVersion.txt)

# Get git commit hash
LONG_COMMIT=$(git rev-parse HEAD)

# Build with version info
docker compose build \
  --build-arg GIT_COMMIT=$LONG_COMMIT \
  --build-arg BUILD_VERSION=$BUILD_VERSION

# Push to registry
docker compose push

# Deploy to swarm
docker stack deploy -c docker-compose.yml mystack
```

Complete Deployment Script Example

```
#!/bin/bash
set -e

# Configuration
REGISTRY="yourregistry"
SERVICE="nodeserver"
COMPOSE_FILE="docker-compose.yml"

# Read version
BUILD_VERSION=$(cat ./globals/_versioning/buildVersion.txt)
LONG_COMMIT=$(git rev-parse HEAD)
SHORT_COMMIT=$(git rev-parse --short HEAD)

# Export for docker-compose
export BUILD_VERSION
export LONG_COMMIT
export DOCKER_BUILD_VERSION="${BUILD_VERSION}"

echo "Building version ${BUILD_VERSION} (commit: ${SHORT_COMMIT})"

# Build images
docker compose -f $COMPOSE_FILE build

# Tag with version AND latest
docker tag ${REGISTRY}/${SERVICE}:latest ${REGISTRY}/${SERVICE}:${BUILD_VERSION

# Push both tags
docker compose -f $COMPOSE_FILE push
docker push ${REGISTRY}/${SERVICE}:${BUILD_VERSION}

# Deploy to swarm
echo "Deploying to swarm..."
docker stack deploy -c $COMPOSE_FILE mystack

echo "Deployed version ${BUILD_VERSION} successfully"
```

CONCLUSION

Docker Swarm isn't as flashy as Kubernetes, but it's incredibly capable for production workloads. We've been running it for years with minimal issues - the key is getting the networking, DNS, and Dockerfiles right from the start.

What we covered:

- The complete Swarm hierarchy (Swarm → Nodes → Stacks → Services → Tasks)
- VPS requirements and cost planning across providers
- Production-ready installation and initialization
- DNS configuration that actually works
- Encrypted overlay networks
- Multi-stage Dockerfiles with ModSecurity WAF
- Advanced compose patterns (cache_from, placement constraints, global mode)
- Docker Configs vs Secrets
- Full rollback configuration
- Complete Prometheus + Grafana + cAdvisor monitoring stack
- Docker Management Platforms (Portainer, Dokploy, Coolify, CapRover, Dockge)
- CI/CD versioning and deployment workflows
- Secret management done right
- Dev vs Prod configurations

If you're considering Swarm vs K8s, ask yourself:

- Do you have a dedicated platform team? → K8s might be worth it
- Small team needing "good enough" orchestration? → Swarm will save you countless hours
- Need to ship fast with battle-tested patterns? → Swarm + these configs = production ready

The configs in this guide have been refined over years of production use. Take what you need, adapt it to your stack, and save yourself the debugging time I went through.

GITHUB REPO

All compose files, Dockerfiles, and configs from this guide:

github.com/TheDecipherist/docker-swarm-guide

What's your Swarm setup? Running it in production? Home lab? What providers are you using? Drop your configs and war stories below – I'll incorporate the best tips into V2.