100+ Docker Tips & Tricks

Practical tips for building, running, troubleshooting, and securing containers

From the basics to Compose Watch, Docker Scout, and the latest AI tools in Docker Desktop.

102 tips · 13 categories · Compiled July 2026 · SplitZrate.com

---

Contents

1. Fundamentals & CLI Efficiency 2
2. Dockerfile & Image Building 2
3. Docker Compose 3
4. Networking 4
5. Volumes & Storage 4
6. Security 5
7. Performance & Optimization 5
8. Troubleshooting & Diagnostics 6
9. Docker Desktop & Local Development 6
10. CI/CD & Registry 7
11. Orchestration 7
12. AI & Docker 8
13. Production & Operations 8

1. Fundamentals & CLI Efficiency

1.1 Use docker ps with smart formatting Example: `docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"` — show only the information you actually need instead of the full default table.

1.2 Clean up clutter in one sweep Example: `docker system prune -a --volumes` — removes stopped containers, unused networks, dangling images, and volumes that aren't in use. Use with caution in production.

1.3 Jump into a running container with an interactive shell Example: `docker exec -it container_name /bin/bash` — drop into a running container to troubleshoot without stopping it.

1.4 Watch container logs efficiently Example: `docker logs -f --tail 100 container_name` — follow the last 100 lines live instead of flooding the terminal with the entire log.

1.5 Copy files to and from containers Example: `docker cp container_name:/app/log.txt ./log.txt` — pull a file out of a running container without mounting a volume.

1.6 Use aliases for common commands Example: `alias dps='docker ps --format "table {{.Names}}\t{{.Status}}"'` in your `.bashrc` saves keystrokes every day.

1.7 Inspect anything with docker inspect Example: `docker inspect container_name --format '{{.State.Health.Status}}'` — pull exactly the field you need out of the JSON output instead of reading the whole structure by hand.

1.8 Run one-off commands without keeping the container around Example: `docker run --rm -it ubuntu:26.04 bash` — `--rm` automatically cleans up the container when it exits, perfect for quick tests.

1.9 Keep an eye on resource usage live Example: `docker stats` — a real-time overview of CPU, memory, and network traffic per container.

1.10 Search local images quickly Example: `docker images --filter "reference=myapp*"` — filter the image list directly instead of piping it through `grep`.

1.11 Rename an image without rebuilding it Example: `docker tag old-name:1.0 new-name:1.0` — tags the same image ID with a new name, handy before pushing to a different registry.

---

2. Dockerfile & Image Building

2.1 Build in multiple stages for slimmer images Multi-stage builds separate the build environment from the production environment. Example: build with a full SDK image in the first stage, then copy only the compiled artifact into a minimal runtime image in the final stage.

2.2 Choose slim base images Example: use `python:3.14-slim` instead of `python:3.14` — fewer packages means fewer known vulnerabilities (CVEs) and a smaller image size.

2.3 Use layer caching smartly Put commands that rarely change (e.g. `apt install`) early in the Dockerfile and commands that change often (e.g. `COPY . .`) last — that maximizes how much Docker can reuse from the cache.

2.4 Combine RUN commands for fewer layers Example: `RUN apt update && apt install -y curl git && rm -rf /var/lib/apt/lists/*` — one layer instead of three, and cleaning up in the same step keeps the image size down.

2.5 Use .dockerignore consistently Example: create a `.dockerignore` with `node_modules`, `.git`, `*.log`, and build artifacts — stops unnecessary files from being sent to the build context and leaking into the image.

2.6 Name your build stages clearly Example: `FROM node:22 AS builder` and `COPY --from=builder /app/dist ./dist` — makes the Dockerfile self-documenting and easier to maintain.

2.7 Pin exact versions instead of latest Example: `FROM nginx:1.27.3` instead of `FROM nginx:latest` — guarantees reproducible builds and avoids surprises when "latest" changes underneath you.

2.8 Use BuildKit cache for faster CI builds Example: `docker buildx build --cache-from type=gha --cache-to type=gha,mode=max .` in GitHub Actions — shares cache between runs and can cut build times in half.

2.9 Build multi-platform images in a single command Example: `docker buildx build --platform linux/amd64,linux/arm64 -t myimage:1.0 --push .` — build and publish for multiple architectures at once, without separate pipelines.

2.10 Offload heavy builds to the cloud Docker Build Cloud runs builds on remote infrastructure instead of your local machine — especially valuable for multi-platform builds that would otherwise take a long time locally.

2.11 Use BuildKit secrets instead of ARG for sensitive data Example: `RUN --mount=type=secret,id=npmtoken npm install` — secrets never end up in the image's layers or history, unlike build arguments.

2.12 Lint your Dockerfile before building Example: `hadolint Dockerfile` — catches common mistakes (wrong layer order, missing version pins) before you even run `docker build`.

2.13 Test that an image actually starts before you push it Example: `docker run --rm myimage:1.0 --version` (or an equivalent quick smoke test) right after building — catches broken entrypoints before they reach CI or production.

---

3. Docker Compose

3.1 Develop with automatic sync and restart Example: `docker compose up --watch` — Compose Watch syncs source changes straight into running containers and rebuilds when needed, eliminating the manual edit-build-run cycle.

3.2 Separate environments with multiple compose files Example: `docker compose -f compose.yaml -f compose.prod.yaml up` — let a base file hold what's shared and override files handle environment-specific differences.

3.3 Use profiles for optional services Example: mark non-critical services (e.g. an email-testing tool, a search engine) with `profiles: ["debug"]` in the compose file so they only start with `docker compose --profile debug up`.

3.4 Handle secrets safely in Compose Example: define secrets under the `secrets:` key and mount them into the service instead of putting sensitive data in environment variables, which can leak via `docker inspect` or logs.

3.5 Define dependency order correctly Example: use `depends_on` with `condition: service_healthy` instead of just a service name — guarantees a database is actually ready before the application starts, not just that the container exists.

3.6 Scale services horizontally, locally Example: `docker compose up --scale worker=5` — run five instances of a service to test load balancing or parallelism locally.

3.7 Validate the compose file before running it Example: `docker compose config` — renders the final configuration (after variable substitution and merging files) so you can catch errors before startup.

3.8 Declare AI models as dependencies Compose now supports AI models as their own resource type alongside services, making model dependencies visible directly in the infrastructure code instead of hidden in application code.

3.9 Reuse .env files across projects Example: `docker compose --env-file .env.staging up` — swap the entire environment configuration with one flag instead of editing the compose file by hand.

---

4. Networking

4.1 Create isolated networks per project Example: `docker network create my-project-net` — keep containers for different projects separated so they can't accidentally reach each other.

4.2 Use service names instead of IP addresses Within the same Docker network, containers can reach each other by service name (e.g. `db` or `redis`) thanks to built-in DNS — never hardcode IP addresses between containers.

4.3 Publish ports deliberately Example: `docker run -p 127.0.0.1:8080:80 nginx` — bind to localhost instead of all interfaces (`0.0.0.0`) if the service shouldn't be reachable from outside.

4.4 Troubleshoot networking issues between containers Example: `docker exec -it container_name ping other_container` or `docker network inspect network_name` to see which containers are actually connected.

4.5 Use host networking mode sparingly Example: `docker run --network host` gives the best performance but removes network isolation — only use it when it's necessary and deliberate.

4.6 Create your own DNS aliases for containers Example: `docker run --network-alias api my-api-image` — give a container extra names it can be reached under on the same network.

4.7 Restrict outbound traffic with network policies In production environments, combine Docker networks with host-level firewall rules to restrict which containers can reach the internet directly.

4.8 Clean up unused networks Example: `docker network prune` — removes networks that no longer have any connected containers, a common source of clutter in older projects.

---

5. Volumes & Storage

5.1 Prefer named volumes over bind mounts in production Example: `docker volume create my-data` and mount with `-v my-data:/var/lib/data` — Docker manages the lifecycle better than direct paths into the host's filesystem.

5.2 Use bind mounts for active development Example: `-v $(pwd)/src:/app/src` — mirror local source code straight into the container so changes show up immediately without a rebuild.

5.3 Back up a volume the easy way Example: `docker run --rm -v my-data:/data -v $(pwd):/backup ubuntu tar czf /backup/backup.tar.gz /data` — archive a volume's contents to a local file without extra tooling.

5.4 Clear out unused volumes regularly Example: `docker volume prune` — removes volumes no longer referenced by any container, a common source of forgotten disk usage.

5.5 Mount volumes read-only where you can Example: `-v config:/etc/app/config:ro` — stops a container from accidentally overwriting configuration files it should only be reading.

5.6 Check where a volume is actually stored Example: `docker volume inspect my-data` — shows the mount point on the host, handy for manual troubleshooting or migration.

5.7 Migrate data between volumes safely Example: `docker run --rm -v old-vol:/from -v new-vol:/to ubuntu bash -c "cp -a /from/. /to/"` — copy contents between named volumes without going through the host's filesystem.

---

6. Security

6.1 Scan images before you push them Example: `docker scout cves myimage:1.0` — catch known vulnerabilities early in the development workflow instead of discovering them in production.

6.2 Use docker scout sbom instead of the old sbom command `docker sbom` is deprecated in favor of `docker scout sbom` — update any scripts or pipelines that still use the old command.

6.3 Run containers as non-root by default Example: add `USER node` (or create your own restricted account) in the Dockerfile — limits the damage if a container gets compromised.

6.4 Sign and verify images Use tools like Cosign to sign images in your pipeline and verify signatures before deployment, so only approved images can run in production.

6.5 Restrict a container's capabilities Example: `docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE myimage` — strip all Linux capabilities and add back only what's actually needed.

6.6 Set resource limits to prevent a "noisy neighbor" Example: `docker run --memory=512m --cpus=1.0 myimage` — stops one container from hogging the entire host's capacity because of a bug or memory leak.

6.7 Keep a production checklist Before an image goes to production: pinned base image versions, a non-root user, no leftover dev dependencies, scanned with no critical findings, and a working healthcheck.

6.8 Use secrets management, never environment variables, for sensitive data Sensitive values in plain environment variables show up in `docker inspect` and can leak into logs — use Compose secrets or an external secrets manager instead.

6.9 Keep Docker Engine and Desktop updated Security fixes for container-escape vulnerabilities (in runc, for example) ship on an ongoing basis — an outdated Docker Engine is one of the most common causes of container-related security incidents.

6.10 Make the filesystem read-only where you can Example: `docker run --read-only --tmpfs /tmp myimage` — makes the entire container filesystem read-only except for explicitly allowed temporary directories, sharply limiting what an attacker can do after a breach.

---

7. Performance & Optimization

7.1 Measure image size before and after changes Example: `docker images myimage --format "{{.Size}}"` — compare the size before and after a Dockerfile change to confirm an optimization actually helped.

7.2 Use healthchecks that reflect real readiness Example: `HEALTHCHECK CMD curl -f http://localhost:3000/health || exit 1` — a container that's "running" isn't the same as a container that can actually handle traffic.

7.3 Avoid unnecessary layers in the Dockerfile Every `RUN`, `COPY`, and `ADD` creates a new layer. Combine related commands where it makes sense to keep the image both smaller and faster to pull.

7.4 Analyze what's taking up space in an image Example: `docker history myimage:1.0` — see the size per layer and spot where unnecessary data is sneaking in.

7.5 Use tmpfs for fast, temporary storage Example: `docker run --tmpfs /tmp myimage` — RAM-backed storage for temporary files that don't need to outlive the container.

7.6 Prime the CI cache with registry-based caching Example: `docker buildx build --cache-from type=registry,ref=myregistry/cache --cache-to type=registry,ref=myregistry/cache,mode=max .` — shares build cache across CI runs without depending on a specific build agent.

7.7 Limit how many log lines Docker keeps in memory Pair a trimmed log driver with external log aggregation (Loki or ELK, for example) for high-volume workloads — otherwise logging itself can become a bottleneck.

---

8. Troubleshooting & Diagnostics

8.1 Understand why a container crashed Example: `docker inspect container_name --format '{{.State.ExitCode}}'` together with `docker logs container_name` usually gives you the full picture of what went wrong.

8.2 Debug a container that has already exited Example: `docker run -it --entrypoint sh myimage` — start a new instance with an interactive shell instead of the default command, to explore the filesystem manually.

8.3 Use Gordon AI for fast troubleshooting Docker Desktop's built-in AI agent Gordon can analyze why a container is exiting unexpectedly or why a build is underperforming — worth asking before you dig in manually.

8.4 Monitor events in real time Example: `docker events` — watch a live stream of everything happening in your Docker daemon (starts, stops, health status) while troubleshooting an intermittent issue.

8.5 Check daemon logs for system-level problems Example: `journalctl -u docker.service -f` on Linux — when the problem seems to be in the Docker daemon itself rather than a single container.

8.6 Test network connectivity from inside a container Example: `docker run --rm --network container:my-app nicolaka/netshoot curl -v http://localhost:3000` — borrow another container's network stack to troubleshoot connectivity without installing tools in the production image.

8.7 See what changed inside a container since it started Example: `docker diff container_name` — shows exactly which files were changed, added, or removed inside a *running or stopped container* compared to its own starting state (the image it was created from). It compares a container to itself, not two separate images — useful for spotting unexpected writes to the filesystem or confirming what an install script actually touched.

---

9. Docker Desktop & Local Development

9.1 Sandbox AI coding agents with docker sbx `docker sbx` (Docker Sandboxes) runs an AI coding agent — Claude Code, Codex, Gemini CLI, and similar tools — inside an isolated container instead of directly on your machine, so the agent can read, edit, and run code without full access to your host filesystem or network. Worth reaching for whenever you're letting an agent operate somewhat autonomously and want a hard boundary around what it can touch.

9.2 Take advantage of synchronized file shares for WSL2 Synchronized file shares give significantly faster filesystem performance for Windows developers running Linux containers via WSL2, compared to a traditional bind mount across the Windows/Linux boundary.

9.3 Let VirtioFS handle file sharing on Mac The older osxfs sharing system has been fully phased out in favor of VirtioFS, which delivers better performance and avoids unnecessary file-ownership overhead.

9.4 Use Resource Saver to save battery Docker Desktop's Resource Saver mode automatically pauses the Docker engine when it's idle — good to have enabled on laptops.

9.5 Test Kubernetes locally, right inside Desktop Docker Desktop ships with a built-in Kubernetes cluster you can switch on with one click — good for testing manifests locally before they go to a real cluster.

9.6 Keep an eye on the log viewer for a quick overview The now generally-available log viewer in Docker Desktop can be exported directly — useful for sharing troubleshooting data with colleagues without manually copying terminal output.

9.7 Keep Desktop extensions switched off unless you're actively using them Extensions have been off by default since recent versions — only enable the ones you actually need to keep Desktop lightweight and reduce the attack surface.

---

10. CI/CD & Registry

10.1 Use multi-stage caching in CI pipelines Example: build the builder stage separately and cache it between runs — only changed layers need rebuilding, which can cut pipeline times in half.

10.2 Tag images with the commit hash, not just "latest" Example: `docker build -t myregistry/app:$(git rev-parse --short HEAD) .` — makes it possible to trace exactly which code is running in a given environment.

10.3 Scan images automatically in the pipeline Add `docker scout cves` as a step in CI before pushing to the registry — stop vulnerable images before they ever reach production.

10.4 Use a private registry for internal images Example: `docker push myregistry.internal:5000/app:1.0` — keep internal images out of public registries for both security and cost reasons.

10.5 Automatically clean up old images in the registry Set up lifecycle rules in your registry (most cloud registries support this) to automatically remove images older than a certain number of days.

10.6 Build once, deploy everywhere Build an image once in CI and reference the same digest through the entire chain (test, staging, production) instead of rebuilding per environment — guarantees what was tested is exactly what gets deployed.

---

11. Orchestration

11.1 Try Swarm for simple clustering Example: `docker swarm init` followed by `docker service create --replicas 3 myimage` — built-in orchestration without having to stand up a full Kubernetes cluster for smaller workloads.

11.2 Update services with zero downtime Example: `docker service update --image myimage:2.0 --update-parallelism 1 --update-delay 10s myservice` — rolling update, one instance at a time.

11.3 Use Docker Desktop's bundled Kubernetes for local testing Before manifests go to a production cluster, test them locally against Desktop's integrated Kubernetes cluster for faster iteration.

11.4 Define resource limits in orchestration manifests Always set `limits` and `requests` (Kubernetes) or the equivalent in Swarm services — stops a single service from starving the rest of the cluster.

11.5 Keep health checks consistent between Compose and orchestration If the same healthcheck logic is used locally (Compose) and in production (Kubernetes/Swarm), it reduces the risk of surprises at deployment time.

---

12. AI & Docker

12.1 Run AI models locally with Docker Model Runner Example: `docker model pull ai/smollm2:360M` followed by `docker model run ai/smollm2:360M` — prototype against a language model locally without depending on external APIs or sending data to a third party.

12.2 Use registry mirrors for models Docker Model Runner now supports registry mirrors, making it faster and more reliable to pull models in environments with limited or unstable internet connectivity.

12.3 Containerize MCP servers safely Docker MCP Toolkit runs Model Context Protocol servers in containers, giving AI tools access to resources without installing them directly on the host machine — easier to sandbox and clean up.

12.4 Ask Gordon before troubleshooting manually Gordon, Docker Desktop's built-in AI agent, can offer concrete optimization suggestions for a Dockerfile or explain why a build is performing worse than expected — a good first step before diving in yourself.

12.5 Version AI models just like images Treat model versions in your Compose file like any other dependency — pin the version, document changes, and test before upgrading in production.

---

13. Production & Operations

13.1 Use restart policies deliberately Example: `docker run --restart unless-stopped myimage` — let Docker automatically restart the container on crash or host reboot, except when you've stopped it manually yourself.

13.2 Log to the right driver for your environment Example: `docker run --log-driver json-file --log-opt max-size=10m --log-opt max-file=3 myimage` — stops container logs from growing unbounded and filling the disk.

13.3 Separate configuration from the image with environment variables Build the same image for every environment and inject environment-specific configuration via environment variables or compose files — avoid building different images for test, staging, and production.

13.4 Set up automatic restarts on health failures Combine a `HEALTHCHECK` in the Dockerfile with an orchestrator (Swarm or Kubernetes) that automatically replaces instances that repeatedly fail the health check.

13.5 Document images with OCI standard labels Example: `LABEL org.opencontainers.image.source="https://github.com/org/repo"` — makes it traceable where an image came from and which version it corresponds to, directly in the image's metadata.

13.6 Have a clear rollback plan Always keep the previous working image tag available in the registry so a rollback is a single `docker service update --image app:previous-version` away, not a crisis-mode rebuild.

13.7 Monitor containers continuously, not just during incidents Feed `docker stats`-style metrics into a real monitoring system (Prometheus, Grafana) instead of just checking manually after something has already gone wrong.

© SplitZrate 2026 · 100+ Docker Tips & Tricks