Practical DevSecOps

Docker 101 Guide for Security Professionals

Build, Ship & Scale with Ease

www.practical-devsecops.com

Contents

Introduction

Overview

Docker changed how software gets built, shipped, and run. Instead of relying on heavy virtual machines, Docker uses containers. These are lightweight, isolated environments that package code, dependencies, and system tools together.

You can run the same application on a laptop, a server, or in the cloud without surprises. Developers move faster. Operations teams spend less time fixing “it works on my machine” problems.

Containers are not magic. They share the host’s kernel. One mistake can expose your entire system. Most public images are unvetted. Default settings are risky. Security teams who treat containers like black boxes are asking for trouble.

If you work in DevOps, security, or software development, you need to know how Docker works, where it fails, and how to keep your stack safe. This book gives you the facts, the risks, and the workflows that matter. No marketing. No empty promises. Just what you need to get Docker right.

CHAPTER 1

Containers vs. Virtual Machines

What a container is, what a VM is

A container packages your application, its dependencies, and system tools into a single unit. It runs as a process on the host operating system and shares the host's kernel. You can start and stop containers quickly, and they use less memory than virtual machines.

For example, a Node.js web app can run in a container with all its libraries, and you can move it from your laptop to the cloud without changes. In contrast, a virtual machine runs a full operating system on virtualized hardware. Each VM has its own kernel, drivers, and resources, separated from the host by a hypervisor.

You can run Windows on a VM inside a Linux host, or vice versa. VMs take longer to start and use more resources, but they offer stronger isolation.

How containers isolate processes

Containers use namespaces to hide resources like process IDs, network interfaces, and file systems from other containers. Control groups (cgroups) limit how much CPU, memory, and disk a container can use. This setup keeps containers from interfering with each other. For example, if you run a Python API and a Redis cache in separate containers, they won't see each other's processes or files unless you connect them.

However, all containers still share the same kernel. If an attacker finds a kernel vulnerability, they can break out of a container and reach the host or other containers. VMs don't have this problem. Each VM runs its own kernel, so a compromise in one VM doesn't affect others.

Where containers fall short compared to VMs

Containers start fast and use fewer resources, but their isolation is weaker. If you run untrusted code in a container, you risk exposing the host. VMs are slower and heavier, but they provide a hard wall between workloads.

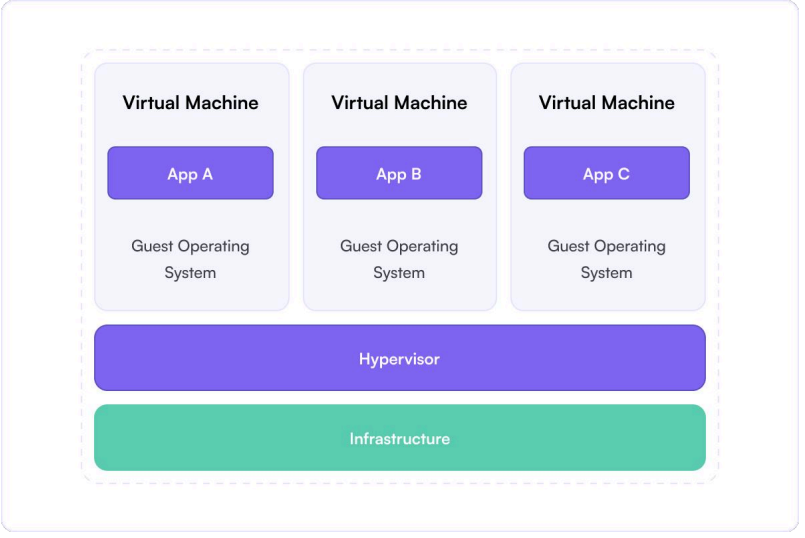
For example, a financial company runs its customer database in a VM to meet compliance rules, while running its web front end in containers for speed.

Security boundaries: what's real, what's marketing

Others often claim containers are as safe as VMs. That's not true. Containers have thinner boundaries. They rely on the host's kernel for isolation. If you misconfigure a container or run it as root, you make it even easier for attackers.

VMs provide stronger boundaries because each one runs a separate operating system. In regulated industries, teams typically use Kata Containers or Firecracker microVMs to get container speed with VM-level isolation.

When to use containers, when to use VMs



Use containers for stateless apps, microservices, and when you need to move fast. Containers work well for web services, batch jobs, and anything that needs to scale quickly. For example, a SaaS company runs its CI/CD pipeline in containers to spin up test environments in seconds.

Use VMs for untrusted code, legacy apps, or when you need strong separation. If you need to run different operating systems, use VMs. A common pattern is to run containers inside VMs to combine speed and isolation.

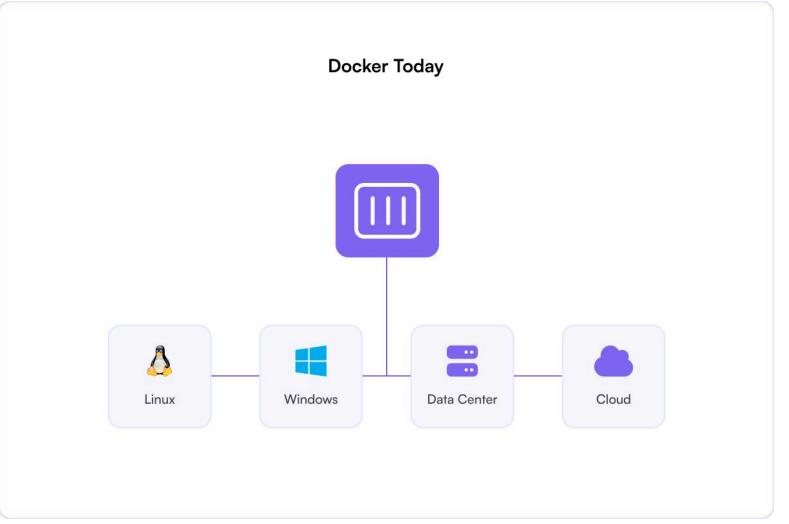
Real-world use cases and trade-offs

Many teams use both containers and VMs. For example, a cloud provider runs customer workloads in VMs for isolation, but each VM runs multiple containers for efficiency. Kubernetes clusters typically run inside VMs to balance security and flexibility.

CHAPTER 2

Docker Engine: The Practical Guide

What Is Docker Engine?



Docker Engine is the core service that builds, runs, and manages containers. It's the background process (the daemon) that does the heavy lifting.

You interact with it using the Docker CLI or API. The Engine runs on Linux, Windows, and macOS, and it's the same everywhere. This consistency is why containers work the same on a laptop, server, or cloud.

Core Components:

Images: These are read-only templates. Every container starts from an image. Images are built from Dockerfiles and can be pulled from registries like Docker Hub or your own private registry.

Containers: These are running instances of images. Each container is isolated but shares the host's kernel. Example: **docker run -d -p 8080:80 nginx:alpine** starts a web server in the background, mapping port 80 in the container to 8080 on the host.

Dockerfile: This is a text file with instructions for building an image. Example:

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm ci --only=production
EXPOSE 3000
CMD ["npm", "start"]
```

This builds a Node.js app image.

Registries: These store and distribute images. Docker Hub is public. You can run your own registry for private images. Example: docker push, **myregistry.local:5000/myimage:latest** uploads an image to a private registry.

Volumes: These store data outside the container's filesystem. Use volumes to persist databases, logs, or configs. Example: **docker run -v mydata:/var/lib/mysql mysql:8** keeps database data even if the container is deleted.

Networking: Docker creates isolated networks for containers. By default, containers get a bridge network. You can create custom networks for multi-container apps. Example: docker network create mynet and then docker run --network=mynet ... to connect containers.

Logs: Every container's output is logged. Use **docker logs <container>** to see what's happening. For production, use logging drivers to send logs to files or external systems.

Pruning: Unused images, containers, networks, and volumes pile up. Clean them with docker system prune or more targeted commands like docker image prune.

Real-World Examples

Building and Running: You have a Python app. Write a Dockerfile, build it with **docker build -t myapp .**, then run it with **docker run -d -p 5000:5000 myapp**

Multi-Stage Builds: For production, keep images lean. Use multi-stage builds to compile code in one stage and copy only the output to the final image.

```
FROM golang:1.22 AS builder
WORKDIR /src
COPY . .
RUN go build -o app

FROM scratch
COPY --from=builder /src/app /app
ENTRYPOINT ["/app"]
```

Volume Usage: For a database, always use named volumes. Example: **docker run -d -v pgdata:/var/lib/postgresql/data postgres:16** keeps your data safe across container restarts.

Networking: For a web app and database, create a user-defined bridge network.

Example:

```
docker network create appnet
docker run -d --network appnet --name db postgres:16
docker run -d --network appnet --name web -e
DATABASE_URL=postgres://... mywebapp
```

Now, web can reach db by name.

Troubleshooting and Maintenance

- **Logs:** Use docker logs for quick debugging. For persistent logging, configure a logging driver.
- **Resource Limits:** Set CPU and memory limits to prevent a runaway container from taking down the host. Example: docker run --memory=512m --cpus=1 ...
- **Pruning:** Regularly prune unused resources. Example: docker system prune -a removes all unused images, containers, and networks. Be careful; this is irreversible.

Images: building, storing, and pulling

A Docker image is a read-only template with everything needed to run a container. Build images using a Dockerfile. Store them locally or push them to a registry. Pull images from trusted sources only. Most public images are not reviewed for security. Build your own when possible. Always check the source and scan for vulnerabilities before use.

Containers: running, stopping, removing

A container is a running instance of an image. Start containers with docker run. Stop them with docker stop. Remove them with docker rm. Use docker ps to see what's running. Clean up unused containers to avoid clutter and reduce the attack surface.

Registries: public, private, and the risks of each

A registry stores images. Public registries like Docker Hub are easy to use but risky. Anyone can upload images, and many are not safe. Private registries give you control. Use them for sensitive or internal images. Always scan images before use, no matter where they come from.

A Dockerfile is a script that builds an image. Keep it simple. Use official base images. Avoid adding secrets or credentials. Clean up temporary files to keep images small. Set user to non-root. Pin versions for repeatable builds. Review Dockerfiles for mistakes before building.

Volumes: data persistence, backup, and recovery

Volumes store data outside the container's file system. Use them for databases and anything that needs to survive container restarts. Back up volumes regularly. Know where your data lives. If you lose a volume, you lose the data. Don't store important data inside the container.

Networking: bridge, host, overlay, and what you actually need

Docker networking can be simple or complex. Bridge is the default and works for most cases. Host mode gives containers direct access to the host's network. Overlay is for multi-host setups. Only open the ports you need. Avoid exposing containers to the public unless required.

Logs and pruning: keeping things clean, finding problems

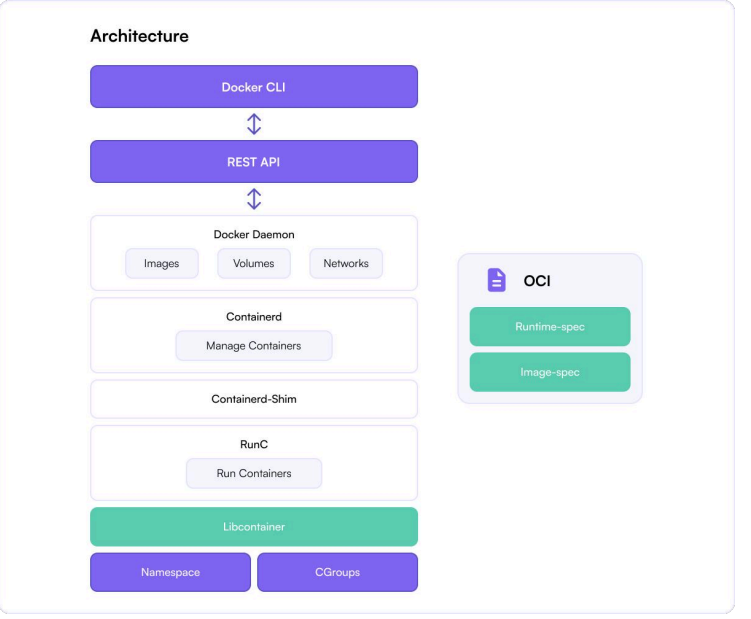
Logs help you spot issues. Use **docker logs** to view output from containers. Set up log rotation to avoid filling up disk space. Prune unused images, containers, and networks with **docker system prune**. Regular cleanup keeps your environment stable and easier to manage.

CHAPTER 3

Docker Desktop: Setup and Daily Use

Docker Desktop installation is now faster and less intrusive than ever. The process has been streamlined, with fewer prompts and less friction. Most users can get Docker Desktop running in minutes, regardless of platform.

The most significant change in 2024 and 2025 is silent updates. Docker Desktop now updates itself in the background, without requiring user approval or manual downloads.



This reduces the risk of running outdated software and closes security gaps quickly. However, silent updates mean you might not notice changes until something breaks. Always check the release notes after an update.

Test your environment and workflows before pushing changes to production. If you rely on specific versions or custom plugins, pin your setup and disable auto-updates. Silent updates save time but can introduce surprises. Stay alert.

But the dashboard has limits. Some features are just visual noise. Don't expect in-depth monitoring, security insights, or advanced configuration here. For anything critical, like resource limits, network settings, or security policies, use the command line. The dashboard is a convenience, not a replacement for real control.

Pulling Images Safely

Never pull random images from public registries. Stick to official images or ones you've built and signed yourself. Before using a public image, check its Dockerfile and source. Scan for vulnerabilities with tools like Docker Scout or Trivy.

For example, if you need NGINX, use nginx:alpine from the official library, not a random fork. If you must use a third-party image, review its history and reputation. Don't trust images with unclear origins. One bad image can compromise your entire system.

Using Volumes Without Losing Data

Volumes are the right way to persist data in Docker. Always map volumes for databases, logs, or anything you want to keep. For example, **docker run -v mydata:/var/lib/mysql mysql:8** keeps your database safe even if the container is deleted.

Don't store important data inside the container's filesystem. If you remove a container, the data in a volume stays. If you delete a volume, the data is gone for good. Back up your volumes regularly.

Know the difference between bind mounts and named volumes. Use named volumes for most cases. They're easier to manage and back up.

Multi-Container Apps with Docker Compose

Docker Compose lets you define and run multi-container apps with a single file. Use Compose for anything beyond a simple test. For example, a web app, database, and cache can all be defined in one **docker-compose.yml**. This makes your setup repeatable and easy to share. Keep your Compose files in version control. Review them for exposed ports and sensitive environment variables. Don't hardcode secrets. Use **.env** files or secret management tools. Compose is the right tool for local development, CI pipelines, and small-scale deployments.

One-Click Kubernetes: When to Use It, When to Skip It

Docker Desktop now offers one-click local Kubernetes clusters. This is useful for testing, learning, and prototyping. You can spin up a cluster in minutes and try out Kubernetes features without extra setup. But don't use local clusters for production.

They don't match the complexity or security of real-world environments. If you don't need Kubernetes, keep it turned off to save resources. Local clusters are great for development, but always test your workloads on a real cluster before going live.

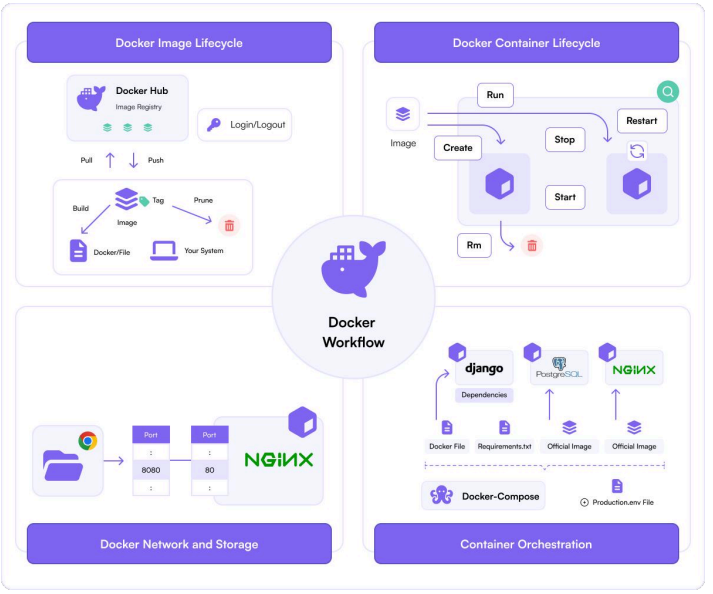
Improved Kubernetes View: What's Changed, Why It Matters

The Kubernetes view in Docker Desktop now shows more details about pods, services, and resources. You can see which pods are running, their status, and resource usage. This helps with troubleshooting and learning.

You can spot issues early, like failed pods or misconfigured services. But don't rely on the dashboard for full security or production monitoring. Use dedicated tools like **kubectl**, Prometheus, or Grafana for anything serious. The improved view is a step forward, but it's still just a starting point.

CHAPTER 4

Workflows That Don't Waste Time



Building Images: Repeatable, Reliable, and Secure

Write a clear Dockerfile. Pin every version for your base image and dependencies. Build images the same way every time to avoid surprises. Scan each image for vulnerabilities before you push it anywhere.

Never add secrets or credentials to your images. Keep images small by removing unnecessary files and layers. Use multi-stage builds to separate build tools from your final image. This keeps your production images lean and secure.

Running Containers: Map Ports, Set Variables, Avoid Mistakes

Map only the ports you need. Don't expose everything by default. Set environment variables in a file, not on the command line, to prevent leaks. Double-check for typos and missing variables before you start a container. Use **--rm** for temporary containers so they don't pile up. Avoid running containers as root. Create a non-root user in your Dockerfile and use it by default. This reduces risk if someone breaks out of the container.

Persisting Data: Volumes, Bind Mounts, and Production Pitfalls

Use named volumes for data that must survive container restarts. Bind mounts work for development but often cause permission problems in production. Always test your data persistence setup before you go live. Know exactly where your data lives and how to back it up. If you lose a volume, you lose the data. Automate your backups and test your restore process.

Exposing Ports: Safe Practices

Expose only the ports you need. Never map sensitive services to public interfaces. Use a firewall to control access. After starting containers, check for open ports with docker ps or a port scanner. Never expose databases or admin panels to the internet unless you have no other option. If you must, use strong authentication and network restrictions.

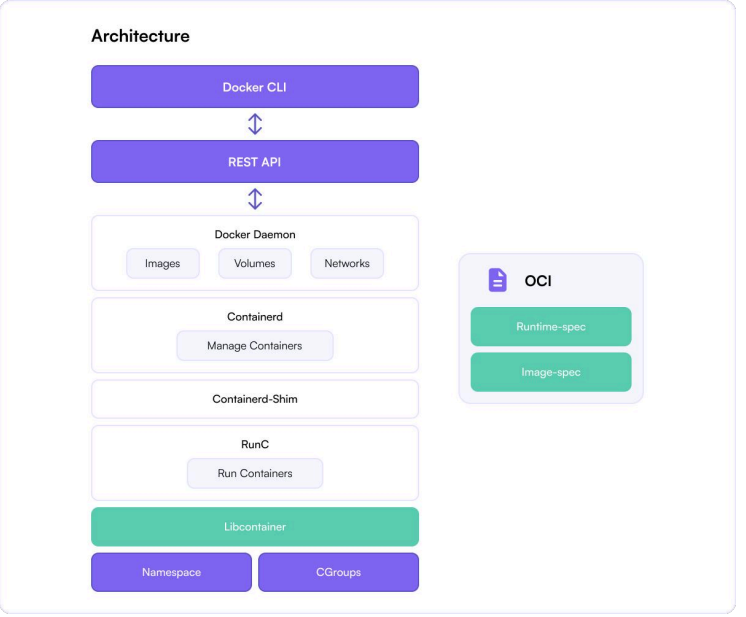
You can spot issues early, like failed pods or misconfigured services. But don't rely on the dashboard for full security or production monitoring. Use dedicated tools like **kubectl**, Prometheus, or Grafana for anything serious. The improved view is a step forward, but it's still just a starting point.

Basic Networking: Connect What's Needed, Isolate the Rest

Use Docker networks to connect containers that need to talk to each other. Don't put everything on the default bridge network. Isolate sensitive services on their own network. Limit which containers can reach the outside world. Review your network setup regularly and remove unused networks.

Using Docker buildx and Arcane Docker: What's New and Useful

Docker buildx lets you build images for multiple platforms from one place. Use it if you need to support different architectures. Arcane Docker adds tools for advanced builds and automation. Learn the basics first. Test everything before you roll out changes. Don't trust new tools blindly.



CHAPTER 5

Security Risks: Where Things Break

Image Risks: Supply Chain Attacks, Poisoned Images, Trust Issues

Most public images are not reviewed or vetted. Attackers upload poisoned images with hidden malware, cryptominers, or backdoors. Supply chain attacks target dependencies inside images, not just the main application. Never trust an image just because it's popular or has many downloads.

Build your own images whenever possible. If you must use a public image, audit every layer and dependency. Scan images for known threats before you use or deploy them. Use tools like Trivy or Docker Scout to catch vulnerabilities early. Don't let convenience override security.

Configuration Mistakes: Default Passwords, Open Ports, Privilege Escalation

Leaving default passwords in place is an open invitation for attackers. Open ports expose your services to the world, making them easy targets for automated scans and attacks. Privilege escalation happens when containers run as root or with excessive permissions. Always change default credentials. Close any ports you don't need.

Run containers with the least privilege possible. Use the USER directive in your Dockerfile to avoid running as root. Review your container and host configurations regularly. Small mistakes here lead to big breaches.

Runtime Risks: Container Escapes, Kernel Exploits, Noisy Neighbors

Containers share the host's kernel. If an attacker breaks out of a container, they can attack the host or other containers. Kernel exploits are rare but can be devastating. Noisy neighbors; containers that hog CPU, memory, or disk, can cause outages or degrade performance.

Set resource limits for every container. Keep your host operating system and Docker Engine patched. Monitor for unusual activity, such as unexpected processes or network connections. Don't assume isolation is perfect.

Registry Hygiene: Why It Matters, How to Do It

A dirty registry is a security risk. Old, unpatched images stick around and get reused accidentally. Delete unused or outdated images. Use private registries for sensitive workloads.

Require image signing and scanning before deployment. Control who can push and pull images. Review registry access logs. Don't let your registry become a dumping ground for abandoned or risky images.

Logging and Monitoring: What to Watch, What to Ignore

Set up logging for all containers. Watch for failed logins, unexpected network traffic, and changes to running containers. Ignore routine noise, but don't miss real threats. Use alerts for high-risk events. Review logs regularly and investigate anything suspicious. If you don't monitor, you won't catch problems until it's too late.

Real-World Breaches: What Actually Happened, What You Can Learn

Breaches happen when teams trust public images, skip updates, or leave ports open. Attackers use automated tools to find and exploit weak setups. Learn from past incidents. Review breach reports and apply fixes to your own environment. Don't assume you're safe just because you haven't been hit yet. Stay proactive.

CHAPTER 6

Best Practices and What's New in 2024 - 2025

Building Secure Images: Minimal Base, No Secrets, Signed Images

Start every image with the smallest base that gets the job done. Fewer packages mean fewer vulnerabilities. Never put secrets, passwords, or keys in your images. Use build-time arguments or secret management tools to inject sensitive data only when needed.

Sign your images to prove they haven't been tampered with. Scan every image for vulnerabilities before you use or deploy it. If you find issues, fix them before moving forward.

Running Containers as Non-Root: Why It Matters, How to Do It

Running containers as root gives attackers a direct path to your host. Always set a non-root user in your Dockerfile using the USER instruction. Test your containers to make sure they work without root.

If something breaks, fix it. Don't take shortcuts or ignore warnings. Running as non-root is a basic security step that blocks many common attacks.

Network Controls: Limiting Exposure, Using Firewalls, Not Trusting Defaults

Open only the ports you need. Use firewalls to block unwanted traffic. Don't trust Docker's default network settings. Isolate sensitive containers on their own networks. Review your network setup after every change. If you expose a service, make sure you know who can reach it and why. Never assume defaults are safe.

Regular Updates: Patching, Silent Component Updates in Docker Desktop

Keep Docker, your images, and your host operating system up to date. Docker Desktop now updates itself in the background. This reduces manual work but can introduce changes without warning. Always check your environment after updates. Test your stack before rolling out new versions. Don't let silent updates catch you off guard.

Kubernetes in Docker Desktop: Local Clusters, Security Implications

Docker Desktop makes it easy to spin up a local Kubernetes cluster. This is great for testing and learning. Don't use local clusters for production. They are not as secure or robust as real clusters. Review your cluster's settings and keep it isolated from sensitive data. Treat local clusters as sandboxes, not as production environments.

Docker MCP: What It Is, Why It's Trending, What to Watch For

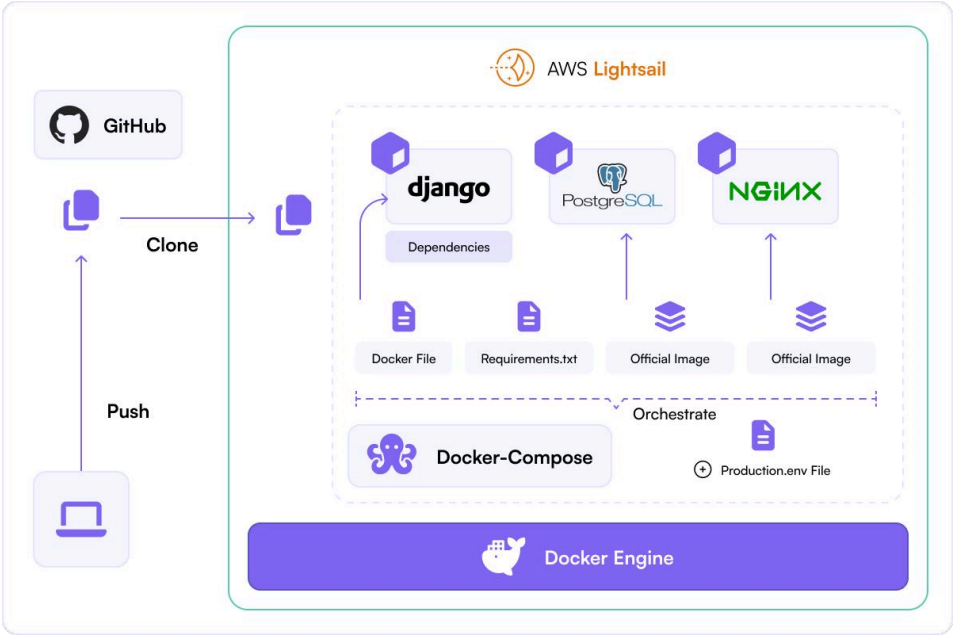
Docker MCP is a new way to manage and connect containers at scale. It's getting attention for its flexibility and speed. Watch for new features and security updates. Don't rush to adopt MCP until you understand the risks and how it fits your workflow.

Docker Model Runner: AI Workloads in Containers. What's Possible!

Docker Model Runner lets you run AI models in containers. This makes deployment easier but brings new risks. Large models can eat up resources and expose sensitive data. Always monitor resource usage and restrict access to models.

Arcane Docker and Buildx: New Tools, New Risks, New Workflows

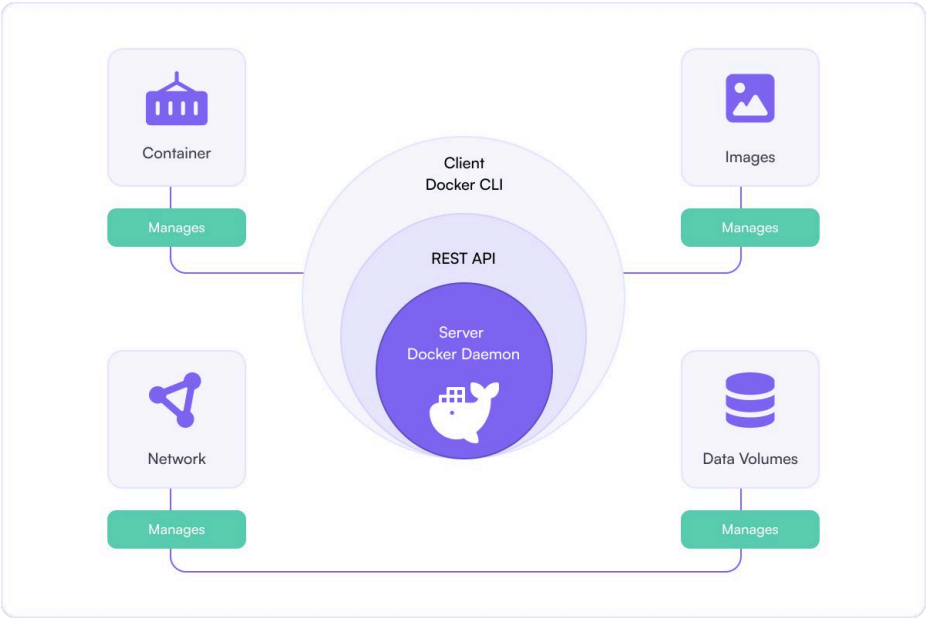
Arcane Docker and buildx add advanced build features and automation. They can accelerate development but also introduce new attack surfaces. Learn how these tools work before using them in production. Test everything and review security settings with every update.



CHAPTER 7

How do Docker Client and Servers Work?

Docker uses a simple client; server model: the Docker client (CLI) sends requests, and the Docker server (the daemon, often called dockerd) does all the actual work of building images and running containers.



Core idea:

The client is just a front end that you interact with using commands like docker run, docker ps, docker build. It does not run containers itself.

The server (daemon) is a long-running background process that receives those requests, manages images, starts/stops containers, sets up networks/volumes, and returns results to the client.

How they talk?

Communication happens via the Docker Engine API, which is a REST API.

On a typical Linux machine, the client talks to the daemon over a Unix socket (for example **/var/run/docker.sock);** for remote control it can talk over TCP to a Docker daemon running on another host.

Where they run!

Both client and daemon can run on the same machine (common on Linux servers and developer laptops).

The client can also run on one machine (e.g., your laptop) while talking to a daemon on another machine (e.g., a remote Linux server or VM), which is how remote Docker management works.

What happens when you run a command?

Example: docker run nginx:

1. Client parses your command and converts it into an HTTP request to the Docker API.
2. Daemon receives it, checks if the nginx image exists locally; if not, it pulls it from a registry (like Docker Hub).
3. Daemon creates a container from that image, sets up filesystem, network, ports, and starts the container process.
4. Daemon sends status and logs back to the client, which displays them in your terminal.

Multiple clients and daemons

- One client can talk to multiple daemons (using different DOCKER_HOST contexts).
- A daemon can accept requests from many clients, which is important in CI/CD systems and shared build servers.

Key Takeaways

- Containers are not VMs. Don't treat them like they are.
- Most Docker images in the wild are unsafe. Build your own or audit everything.
- Default settings are dangerous. Change them.
- Never run containers as root unless you want trouble.
- Volumes can leak data. Know what's stored and where.
- Registry hygiene is non-negotiable. Use private registries when possible.
- Logging is your early warning system. Set it up right.
- Docker Desktop is easier than ever, but silent updates mean you need to check your environment regularly.
- Kubernetes integration is now simple, but complexity brings new attack surfaces.
- Docker MCP and Model Runner are new, powerful, and risky if you don't understand them.
- Security is not a feature. It's a process. Review, test, and update constantly.
- tIf you're not sure about a setting, assume it's unsafe until proven otherwise.

Practical DevSecOps

Invest in Certified Container Security Expert Course

Enroll today and secure containers >

Demand is high, and spots are limited! Secure your place today!

www.practical-devsecops.com