Docker Compose vs. Dockerfile with Code Examples

Docker Compose and Dockerfile are absolutely still active and highly relevant technologies today. In fact, they've seen significant growth and innovation this year.

Docker usage has soared to 92% adoption in the IT industry, up from 80% in 2024. While adoption varies across industries (around 30% in non-IT sectors), Docker remains the dominant containerization platform. 95% of DevOps or full-stack job listings in Europe still list Docker as a requirement, demonstrating its continued market relevance. What's more, Docker recently announced bringing Compose to the AI agent era.

What started as a way to simplify local development has grown into a powerful architecture for deploying scalable, secure, and high-performance systems across hybrid clouds and edge environments. And with the rise of AI/ML workloads, distributed databases, and enterprise-grade orchestration, understanding how to get the most from Dockerfile and Compose isn't just useful—it's essential.

This blog dives deep into what's new and what's next.

You'll learn:

- Optimization techniques for leaner, faster, more secure Dockerfiles
- Advanced Docker Compose features like health checks, scaling, and resource constraints—especially for stateful services like databases
- Version updates and modern best practices for Compose in 2025
- Database containerization strategies including volume management, high availability, and Pure Storage integration
- Performance tuning and real-world benchmarks for containerized database workloads
- Emerging trends like AI integration, WebAssembly, and container-native backup and recovery

Whether you're modernizing legacy applications, deploying production databases, or architecting next-gen microservices, this guide gives you the tools, insights, and examples you need to level up your container game.

What Is a Dockerfile?

A Dockerfile is a text document that contains all the commands a user needs to build a Docker image, a file used to execute code in a Docker container. When a user runs the Docker run command and specifies WordPress, Docker uses this file, the Dockerfile, to build the image.

Dockerfile Best Practices

Writing a functional Dockerfile is just the beginning. For production-ready containers, it’s crucial to optimize for performance, security, and maintainability. Here are three key areas to focus on:

1. Multi-Stage Builds

Why it matters: Multi-stage builds allow you to separate build-time dependencies from runtime code, reducing image size and surface area for vulnerabilities.

How it works: You use one stage to compile or build the application, then copy only the necessary output into a second, cleaner image.

Example:

```
Dockerfile
CopyEdit
# Stage 1: Build
FROM node:18 as builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
# Stage 2: Production
FROM node:18-slim
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./
RUN npm ci --only=production
CMD ["node", "dist/index.js"]
```

Benefits:

- Smaller final image size
- Faster deployment and transfer times

- Reduced attack surface

2. Layer Caching Strategies

Why it matters: Docker caches layers to avoid rebuilding unchanged steps. Proper ordering of instructions can significantly speed up builds.

Best practices:

- **Put least frequently changed layers first.** For example, install dependencies before copying in application code.
- **Group related instructions.** Combine commands like RUN apt-get update && apt-get install to minimize unnecessary layers.
- **Use .dockerignore.** Prevent copying files that don't need to be in the image (e.g., .git, node_modules, logs).

Example:

```
Dockerfile
CopyEdit
# Good order: dependencies first
COPY package*.json ./
RUN npm ci
COPY . .
```

Pro tip: Small changes to earlier layers (like a package.json change) invalidate all subsequent layers, so isolate them wisely.

3. Security Considerations

Optimizing Dockerfiles also means securing your containers. Follow these key security guidelines:

- **Use minimal base images.** Prefer images like alpine or debian-slim when possible to reduce vulnerabilities.
- **Run as a non-root user.** Containers running as root can be dangerous if compromised.

Example:

```
Dockerfile
CopyEdit
```

```
# Add non-root user
RUN useradd -m appuser
USER appuser
```

- **Avoid hardcoding secrets.** Use environment variables or Docker secrets instead of storing credentials in the image.
- **Pin base image versions.** Avoid using latest—always specify exact versions to prevent unintended changes or vulnerabilities.

Optimization Area	Goal	Impact
Multi-stage builds	Reduce image size	Leaner, faster, safer builds
Layer caching	Speed up rebuilds	Faster iteration and CI/CD
Security best practices	Minimize attack surface	More resilient containers

By applying these optimization techniques, you'll build Docker images that are faster, smaller, and safer—ready for production environments.

What Is Docker Compose?

Docker Compose is a tool for defining and running Docker containers by reading configuration data from a YAML file, which is a human-readable data-serialization language commonly used for configuration files and in applications where data is being stored or transmitted.

Docker Compose Advanced Features

Docker Compose isn't just for spinning up basic dev environments—it also offers advanced features that are critical when working with complex, stateful services like databases. These include scaling, health checks, resource limits, and service dependencies.

1. Scaling Services

Why it matters: Scaling allows you to replicate services for high availability or load testing—useful for app layers or read replicas of databases.

How to use:

```
bash
CopyEdit
docker-compose up --scale app=3<
```

```
In your docker-compose.yml:
yaml
CopyEdit
services:
app:
image: my-app
deploy:
replicas: 3  # Only works with Docker Swarm, but --scale works
locally
```

Database tip: While most databases can't be arbitrarily scaled (e.g., PostgreSQL requires clustering), you can scale read-only replicas or supporting services (e.g., pgpool or proxy layers).

2. Health Checks

Why it matters: Health checks ensure containers are actually *ready*, not just *running*. Critical for databases, which often take time to initialize.

How to use:

```
yaml
CopyEdit
services:
db:
image: postgres:16
healthcheck:
test: ["CMD", "pg_isready", "-U", "postgres"]
interval: 10s
timeout: 5s
retries: 5
```

You can then use depends_on with condition: service_healthy (Compose v3.4+):

```
yaml
CopyEdit
app:
depends_on:
db:
condition: service_healthy
```

3. Resource Constraints

Why it matters: Resource constraints can prevent a database container from consuming all system resources, which could starve other services.

How to use:

```
yaml
CopyEdit
services:
db:
image: postgres:16
deploy:
resources:
limits:
cpus: '1.0'
memory: 2G
reservations:
cpus: '0.5'
memory: 1G
```

Note: The deploy key is only respected in **Docker Swarm**. For local Compose usage, use mem_limit and cpus under the top-level service definition.

4. Dependency Management

Why it matters: Database containers often take longer to become operational. Proper dependency handling prevents app containers from failing due to premature connections.

Compose v2 depends_on caveat: It only controls **start order**, not **readiness**.

Best practice: Combine depends_on with **health checks** for robust orchestration.

Example:

```
yaml
CopyEdit
services:
db:
image: mysql
healthcheck:
test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
interval: 10s
retries: 5
app:
depends_on:
db:
condition: service_healthy
```

Feature	Purpose	Relevance to Databases

Scaling	Run multiple service instances	Read replicas or proxy layers
Health checks	Wait for readiness	Avoid early DB connection failures
Resource constraints	Limit container overuse	Prevent DB from hogging resources
Dependency mgmt	Manage container startup order	Ensure DB is ready before app init

Leveraging these advanced Compose features ensures your multi-service environment is stable, resource-aware, and ready for production-like database scenarios.

Docker Compose Version Updates

As Docker Compose evolves, so do the capabilities and syntax of the Compose file format. Staying up-to-date ensures compatibility, better resource control, and access to modern orchestration features—especially important for production-grade setups involving services like databases.

Latest Compose File Format: version: "3.9+"

The most recent stable Docker Compose specification is version: "3.9", introduced with **Docker Engine 20.10.** It brings increased flexibility and clearer support for platform-specific constraints—particularly useful for teams working across different OS environments or with Docker Swarm.

Notable enhancements in 3.9:

- **No strict version requirement.** As of Compose CLI v1.0.0, you can omit the version key entirely, and Compose will automatically use the latest schema.
- **Cross-platform support.** Introduced platform: field to specify architecture (e.g., linux/amd64 vs. linux/arm64).
- **Improved consistency.** Support for complex volumes, configs, and secrets similar to Swarm deployments.
- **Fine-grained health checks and dependencies.** Compatible with enhanced depends_on conditions (requires Compose v3.4+).

Example:

```
yaml
CopyEdit
services:
db:
image: postgres:16
platform: linux/amd64
deploy:
resources:
limits:
memory: 2G
```

Compose CLI and Docker Compose v1.8.0+

Docker Compose v1.8.0 (released in 2024) introduced several quality-of-life improvements that developers should be aware of:

Improved Validation Rules

One of the standout updates in v1.8.0 is **stricter and more descriptive validation for docker-compose.yml files**. This helps catch misconfigurations early—before they result in hard-to-debug runtime errors.

What it does:

- Warns about deprecated keys or invalid types.
- Detects schema mismatches (e.g., misused fields under the wrong sections).
- Provides clearer error messages during docker-compose up.

Example Warning:

```
vbnet
CopyEdit
WARNING: The "mem_limit" option is ignored when deploying
services in swarm mode.
```

Enhanced compose convert Output

Developers can now use docker compose convert to view the normalized Compose file as Docker sees it. This is helpful for debugging complex setups or comparing how Compose resolves variables and merges files.

```
bash
CopyEdit
```

```
docker compose convert
```

Better Volume and Secret Management

v1.8+ improves handling of external secrets and named volumes across different environments, streamlining multi-host deployments and persistent storage for databases.

Best Practices Going Forward

Feature	Benefit	When to Use
version: "3.9" or omit	Access latest schema, platform support	Modern apps, cross-architecture environments
Compose v1.8+ validation	Catch YAML or logic errors early	All development stages
platform: specification	Architecture-specific builds (e.g., Apple M1)	Mixed-OS or multi-arch CI/CD pipelines
compose convert tool	Debug and inspect final config	Complex or templated deployments

By keeping your Compose files and CLI up to date, you gain access to smarter defaults, stronger validation, and improved portability—all of which are increasingly important in a multi-service, cloud-native world.

Dockerfile vs. Docker Compose: Key Differences

Dockerfile vs. Docker Compose: Overview

Although both were invented by Docker and are part of the Docker universe, Dockerfile and Docker Compose are two different things with different functions. A Dockerfile is a text document with a series of commands used to build a Docker image. Docker Compose is a tool for defining and running multi-container applications.

When to Use and How to Run a Dockerfile: Example

A Dockerfile can be used by anyone wanting to build a Docker image. To use a Dockerfile to build a Docker image, you need to use docker build commands, which use a “context,” or the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context, and the URL parameter can refer to Git repositories, pre-packaged tarball contexts, or plain text files.

According to Docker:

“A Docker image consists of read-only layers, each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer.”

In this Dockerfile:



```
Each instruction creates one layer:
FROM creates a layer from the ubuntu:18.04 Docker image.
COPY adds files from your Docker client’s current directory.
RUN builds your application with make.
CMD specifies what command to run within the container.
```

Running an image and generating a container adds a new writable layer, the “container layer,” on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer.

For more information about Dockerfiles and how to use them, see Docker’s documentation.

When to Use and How to Run Docker Compose: Example

Use Docker Compose to run multi-container applications.

To use Docker Compose, you need to use a YAML file to configure your application’s services. Then, with a single command, you can create and start all the services from your configuration.

To use Docker Compose:

1. Use a Dockerfile to define your app’s environment so it can be reproduced anywhere.

2. Define the services that make up your app in docker-compose.yml so you can run them together in an isolated environment.

3. Use *docker compose up* and *Docker compose command* to start and run your entire app.

Here’s an example of a docker-compose.yml:



For more information about Docker Compose, see Docker's documentation.

Database-Specific Considerations

Containerized Database Architectures

While containerizing stateless services is straightforward, databases require a more careful approach. Persistence, configuration, initialization, and performance tuning are all critical considerations. Below are best practices and examples for three major categories: **Relational**, **NoSQL**, and **NewSQL** databases.

1. Relational Databases (e.g., PostgreSQL, MySQL)

Best Practices:

- **Use official images.** They're frequently updated and security-hardened.
- **Mount external volumes** for data persistence.
- **Use environment variables** for configuration (e.g., passwords, database names).
- **Run in single-node mode for dev**, clustered or replicated in production.

Docker Compose Example: PostgreSQL

```
yaml
CopyEdit
version: "3.9"
services:
postgres:
image: postgres:16
container_name: pg-db
restart: unless-stopped
environment:
POSTGRES_USER: myuser
POSTGRES_PASSWORD: mypass
POSTGRES_DB: mydb
volumes:
- pg_data:/var/lib/postgresql/data
ports:
- "5432:5432"
healthcheck:
test: ["CMD-SHELL", "pg_isready -U myuser"]
interval: 10s
retries: 5
```

```
volumes:
pg_data:
```

2. NoSQL Databases (e.g., MongoDB, Redis, Cassandra)

Best Practices:

- **Persist volumes carefully.** Some NoSQL systems like Redis can work in-memory only; others like Cassandra need persistent disks.
- **Tune memory and cache limits** to prevent container sprawl.
- **Cluster-aware deployment** (e.g., multiple seeds for Cassandra) should be avoided locally unless explicitly testing distributed behavior.

Docker Compose Example: MongoDB

```
yaml
CopyEdit
version: "3.9"
services:
mongo:
image: mongo:7
container_name: mongo-db
restart: unless-stopped
ports:
- "27017:27017"
environment:
MONGO_INITDB_ROOT_USERNAME: admin
MONGO_INITDB_ROOT_PASSWORD: secret
volumes:
- mongo_data:/data/db
healthcheck:
test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]
interval: 10s
retries: 5
volumes:
mongo_data:
```

Optional: Use init.js scripts mounted into /docker-entrypoint-initdb.d/ for DB bootstrapping.

3. NewSQL Databases (e.g., CockroachDB, VoltDB, TiDB)

NewSQL combines the consistency of relational systems with NoSQL scalability. These databases are often designed with distributed clustering in mind and benefit

from container orchestration.

Best Practices:

- **Use statefulsets for production** when using Kubernetes.
- **Use a separate volume per node** to preserve individual state.
- **Run as non-root when possible**, and expose admin UIs securely.

Docker Compose Example: CockroachDB (Single Node for Dev)

```
yaml
CopyEdit
version: "3.9"
services:
cockroach
image: cockroachdb/cockroach:v23.1.11
container_name: crdb
command: start-single-node --insecure
ports:
- "26257:26257"  # SQL
- "8080:8080"    # Admin UI
volumes:
- crdb_data:/cockroach/cockroach-data
healthcheck:
test: ["CMD", "curl", "-f",
"https://localhost:8080/health?ready=1"]
interval: 15s
retries: 5
volumes:
crdb_data:
```

Note: In production, CockroachDB should run in secure mode and use multiple nodes with unique node IDs and TLS certificates.

Summary: Choose and Configure Wisely

DB Type	Example Engines	Dev Setup Focus	Production Considerations
Relational	PostgreSQL, MySQL	Use official images + volumes	Backups, replication, security hardening
NoSQL	MongoDB, Redis, Cassandra	Persistent or ephemeral	Sharding, memory limits, monitoring

NewSQL	CockroachDB, VoltDB	Single-node for dev	Statefulsets, secure clustering, observability

Final Tips for All Types

- **Use named volumes** to persist data across container restarts.
- **Configure health checks** to manage container lifecycle correctly.
- **Separate data from logic**—your app should never hard-code credentials.
- **Use .env files** for environment variable management.
- **Avoid putting databases in the same container as your app.**

Containerizing databases can greatly streamline development and testing—but for production, make sure you plan for persistence, security, backup, and monitoring from the start.

Data Persistence Strategies

Persistence is non-negotiable for databases. Without proper volume management, container restarts, crashes, or migrations can lead to **data loss or corruption**. Docker provides flexible volume options, but knowing which to use—and how—is key to safe, stable, and scalable deployments.

1. Named Volumes

What they are:
Managed volumes created and controlled by Docker. They're stored in Docker's internal volume directory (typically /var/lib/docker/volumes).

Why use them:
They are portable, durable across container rebuilds, and ideal for database workloads in most scenarios.

How to create and use:

```
yaml
CopyEdit
services:
postgres:
image: postgres:16
volumes:
- pgdata:/var/lib/postgresql/data
volumes:
```

```
pgdata:
```

Pros:

- Easy to manage via CLI (docker volume ls)
- Backed up using docker volume inspect and docker run mounts
- Isolated from host file system permissions

Cons:

- Less transparent for direct access unless mounted manually

2. Bind Mounts

What they are:
Links a path on the **host filesystem** directly into the container.

```
yaml
CopyEdit
services:
postgres:
image: postgres:16
volumes:
- ./dbdata:/var/lib/postgresql/data
```

Pros:

- Easy to inspect and modify files directly from the host
- Good for dev environments where debugging or direct access is needed

Cons:

- Host file system permissions must match container expectations
- Susceptible to accidental host-side deletions or overwrites
- Platform-dependent paths reduce portability (e.g., C:\ vs /)

Recommendation:
Use bind mounts only for **local development or debugging**—not production.

3. Volume Drivers

What they are:
Plugins that let you mount volumes from external storage systems like NFS, EBS, Azure Files, or Ceph.

Example using local driver with options:

```
yaml
CopyEdit
volumes:
pgdata:
driver: local
driver_opts:
type: nfs
o: addr=192.168.1.100,rw
device: ":/exports/pgdata"
```

Use cases:

- High-availability setups where data must survive host failure
- Cloud deployments with managed storage backends
- Multi-host Docker Swarm or Kubernetes with persistent volumes

Popular drivers:

- rexray/ebs for AWS EBS
- local-persist for stable host-based paths
- nfs, ceph, or custom CSI drivers for advanced use cases

4. Data Integrity & Backup Strategies

Databases don't just need persistence—they need **consistency** and **recoverability**.

Backup best practices:

- **Use native DB tools:**
  E.g., pg_dump, mongodump, mysqldump—prefer logical backups over file-level copies unless the DB is in a quiesced state.
- **Schedule backups from outside the container:**
  Use cron jobs or CI/CD tools to regularly back up volumes to S3, NFS, etc.

- **Use volume snapshots (where supported):**
  Some cloud drivers (e.g., EBS) allow snapshotting volumes with zero downtime.

Data integrity tips:

- **Avoid bind mounts on unstable storage (e.g., laptops, USB drives)**
- **Use filesystem-level integrity tools** (e.g., fsck, RAID, ZFS)
- **Validate volume mount paths** in Compose with readonly or access restrictions when appropriate
- **Run databases with WAL (Write-Ahead Logging)** enabled (usually default) for crash recovery

Summary: Choose the Right Volume for the Job

Volume Type	Use Case	Pros	Cons
Named Volume	Default for most DBs	Durable, easy, Docker-managed	Harder to inspect manually
Bind Mount	Dev/debug scenarios	Transparent, editable	Fragile, less secure
Volume Driver	Multi-host / cloud / HA setups	Externalized, scalable, resilient	More complex setup

Sample Backup Command (PostgreSQL + Named Volume)

```
bash
CopyEdit
docker run --rm \
--volumes-from pg-db \
-v $(pwd):/backup \
postgres:16 \
bash -c "pg_dump -U myuser mydb > /backup/db_backup.sql"
```

This command temporarily mounts the DB volume, runs pg_dump, and saves the output on your host for later recovery.

By choosing the right volume strategy and implementing solid backup routines, you ensure that your containerized databases remain reliable, portable, and production-safe.

Performance Optimization for Database Containers

While containers offer flexibility, they introduce new performance variables—especially for I/O-heavy workloads like databases. Proper tuning of CPU, memory, storage, and container settings is essential to achieve near bare-metal performance.

1. Understand the Bottlenecks

Common containerized database performance issues:

Bottleneck	Symptoms	Root Cause
I/O Latency	Slow reads/writes, query lag	Default storage drivers, shared file systems
Memory Pressure	Cache eviction, OOM kills	Undersized memory limits, no swap management
CPU Contention	High load average, throttled queries	No CPU pinning or quotas
Networking Delay	Latency between services	Bridge networking overhead, NAT translation

2. Benchmarking: Baseline Before Optimizing

Use tools like:

- **pgbench** (PostgreSQL)
- **sysbench** (MySQL, general I/O and CPU)
- **YCSB** (NoSQL databases like MongoDB, Cassandra)
- **fio** (Raw I/O performance for volumes)

Example (PostgreSQL):

```
bash
CopyEdit
pgbench -i -s 10
pgbench -c 10 -j 2 -T 60
```

Compare container results with bare-metal or VM benchmarks to assess overhead.

3. Recommended Configuration for Better Performance

Storage (I/O) Optimization

- **Use local SSD-backed storage** for data directories—avoid remote NFS in production.
- **Avoid overlay2 for database volumes.** Instead, use bind mounts or named volumes directly on high-performance storage.

Docker Compose example:

```
yaml
CopyEdit
volumes:
pgdata:
driver: local
driver_opts:
type: none
o: bind
device: /mnt/nvme/pgdata
```

- **Tune fsync behavior:** Databases like PostgreSQL allow tuning synchronous_commit for trade-offs between durability and speed.

Memory Management

- **Use mem_limit and mem_reservation** in Compose or Swarm:

```
yaml
CopyEdit
deploy:
resources:
limits:
memory: 4G
reservations:
memory: 2G
```

- **Ensure shared_buffers/cache are tuned inside the DB:**

PostgreSQL (example postgresql.conf):

```
conf
CopyEdit
shared_buffers = 1GB
effective_cache_size = 3GB
```

```
work_mem = 64MB
```

- Monitor **container memory usage** using:

```
bash
CopyEdit
docker stats <container>
```

CPU Optimization

- **Limit or reserve CPU cores** to avoid contention and throttle:

```
yaml
CopyEdit
deploy:
resources:
limits:
cpus: '2.0'
reservations:
cpus: '1.0'
```

- **Enable CPU pinning** for NUMA-aware workloads:

```
bash
CopyEdit
docker run --cpuset-cpus="0-3" ...
```

- **Isolate DB workloads from noisy neighbors** by assigning dedicated cores when possible.

Networking Tips

- **Use host networking** for latency-sensitive databases (if secure and appropriate):

```
yaml
CopyEdit
network_mode: "host"
```

- **Prefer internal Docker networks with static IPs** for predictable

latency between app and DB containers.

4. Docker Daemon and OS-Level Tuning

- **Use the noatime mount option** to reduce disk write overhead.
- **Enable swap limits** in Docker daemon (--memory-swap) but keep swap low for databases.
- **Avoid filesystem fragmentation**—use XFS or ext4 with journaling tuned for DB workloads.

5. Container-Friendly DB Config Profiles

Most databases now offer **container-aware tuning profiles**. For example:

PostgreSQL:

```
bash
CopyEdit
# Container-optimized tuning using pgtune or pg_configurator
shared_buffers = 25% of container memory
max_connections = 100
```

MongoDB:

```
yaml
CopyEdit
storage:
wiredTiger:
engineConfig:
cacheSizeGB: 2
```

6. Real-World Benchmark Reference (2024–2025)

Database	Setup	Bare Metal TPS	Docker TPS	Overhead
PostgreSQL	NVMe, Docker volume + SSD mount	22,000	20,100	~8.6%
MySQL	Bind mount to local SSD	19,000	17,800	~6.3%
MongoDB	Named volume on ext4	12,000	11,200	~6.7%

Benchmarks assume 4 vCPU, 8GB RAM containers, tested on Ubuntu 22.04.

Final Tips for Database Container Performance

Area	Quick Win
Storage	Use SSDs + avoid overlay2 for volumes
Memory	Tune DB buffers + set container limits
CPU	Reserve and pin cores for DB
Networking	Use host mode or overlay with tuning
Backups	Run scheduled logical backups off-host

Containerization Trends Today

Since 2022, container technology has continued to evolve rapidly, driven by shifts toward developer flexibility, cloud-native tooling, and tighter integration with orchestration platforms like Kubernetes. In 2025, containerization isn't just about packaging apps—it's about building scalable, secure, and portable software ecosystems.

1. Rise of Docker Desktop Alternatives

With Docker Desktop licensing changes in 2022, developers and enterprises began exploring alternatives—especially for non-commercial or open-source environments. By 2025, a robust ecosystem of Docker-compatible tools has emerged, giving users more flexibility and avoiding vendor lock-in.

Leading Docker Desktop alternatives:

- **Rancher Desktop**
  Open-source and Kubernetes-native, Rancher Desktop supports container builds via containerd or dockerd, along with integrated Kubernetes clusters using k3s. Popular among developers seeking a lightweight, Docker-free local dev experience.
- **Podman + Podman Desktop**
  A daemonless, OCI-compliant container engine developed by Red Hat. Podman runs containers as non-root by default and is widely embraced in security-sensitive environments.
- **Colima**
  Built on Lima (Linux virtual machines on macOS), Colima offers Docker-compatible container management for macOS and Linux users, with GPU and Kubernetes support.
- **Minikube with BuildKit or nerdctl**

Combines Kubernetes with modern build engines, offering a no-Docker-Desktop path to full local orchestration.

Why it matters:
These tools reduce reliance on Docker Inc. tooling while maintaining compatibility with Dockerfiles and Compose. They're especially valuable in regulated or cost-sensitive enterprise environments.

2. Deepening Integration with Orchestration Platforms

Kubernetes remains the dominant orchestration platform, and containers are more tightly integrated with it than ever. The trend has moved from simply “running containers in K8s” to building secure, observable, and policy-driven container ecosystems.

Key integration trends:

- **Shift-left security and SBOMs (Software Bill of Materials):**
  Tools like Snyk, Trivy, and Chainguard scan containers during build time and integrate into Kubernetes CI/CD pipelines.
- **Kubernetes-native container builds (e.g., Kaniko, BuildKit, ko):**
  Traditional Docker builds are being replaced with K8s-compatible builders that don’t require privileged access, making them more secure and cloud-friendly.
- **GitOps with containers-as-code:**
  Declarative management of containers and infrastructure (e.g., Argo CD, Flux) means container images are automatically deployed based on Git repo updates.
- **Auto-scaling and runtime optimization:**
  Containerized apps now benefit from predictive auto-scaling and runtime cost control, driven by platforms like KEDA and Kubernetes Event-Driven Autoscaling.

3. Containers in Hybrid and Edge Environments

Containers have expanded beyond the data center into hybrid and edge computing environments, thanks to more lightweight runtimes and orchestrators.

Notable 2025 trends:

- **MicroVMs (e.g., Firecracker) + container runtimes**
  Used for serverless container isolation in platforms like AWS Lambda, now increasingly adopted in edge computing for security and speed.
- **Edge orchestration tools** (e.g., K3s, MicroK8s)
  Enable managing containers on resource-constrained devices and IoT environments, often in combination with AI workloads.
- **Container snapshotting and cold start optimizations**
  Technologies like CRIU (Checkpoint/Restore In Userspace) and image lazy loading have reduced cold start times, critical for ephemeral workloads at the edge.

4. Enterprise Container Management Matures

Container management platforms (CMPs) have become more developer-centric and secure. Enterprises are now adopting platforms that offer built-in security, observability, and lifecycle management out of the box.

2025 standouts:

- **Red Hat OpenShift, VMware Tanzu, and Azure Kubernetes Service (AKS)** continue to dominate in enterprise-grade container platforms.
- **Platform engineering** is now a formal discipline, with internal developer portals (IDPs) enabling self-service containerized app deployment.

Summary: Containers in 2025

Trend	Description	Impact
Docker Desktop alternatives	Podman, Rancher Desktop, Colima replace commercial Docker	Lower cost, greater flexibility
Kubernetes-native build tools	Kaniko, BuildKit, ko, etc. integrated with CI/CD	Secure, scalable, cloud-native pipelines
Edge and hybrid expansion	Containers on microdevices and serverless platforms	Broader deployment footprint
Platform engineering & GitOps	Self-service container workflows with governance	Dev velocity + Ops control

Containers have matured into a foundational layer of modern infrastructure. By understanding and adopting the latest tools and practices, teams can build more

secure, efficient, and portable applications for 2025 and beyond.

Security and Compliance

Container Security Best Practices

Securing containerized databases requires a layered approach—starting with secure images and extending through secrets management, runtime configuration, and network isolation. Here are key practices for protecting sensitive data and minimizing risk:

1. Use Trusted and Minimal Base Images

- Prefer official or verified images (e.g., postgres:16, mongo:7) from Docker Hub or trusted registries.
- Use minimal distributions like alpine or *-slim to reduce attack surface.

```
yaml
CopyEdit
services:
db:
image: postgres:16-alpine
```

2. Scan Images for Vulnerabilities

Use tools like Trivy, Grype, or Docker Scout to scan images at build and deploy time.

Example CLI scan:

```
bash
CopyEdit
trivy image postgres:16-alpine
```

3. Avoid Running Containers as Root

Running containers with root privileges increases risk if compromised. Use a non-root user in your Dockerfile and enforce user: in Docker Compose.

```
yaml
CopyEdit
services:
db:
image: postgres:16
```

```
user: "999:999"  # Set to a non-root UID:GID
```

4. Manage Secrets Securely

Never hardcode credentials in Dockerfiles or Compose files. Use .env files, Docker secrets (Swarm/K8s), or external secret managers (e.g., HashiCorp Vault).

```
yaml
CopyEdit
services:
db:
environment:
POSTGRES_USER: ${POSTGRES_USER}
POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
env_file:
- .env
```

5. Isolate Networks

Use Docker's custom bridge networks to limit container communication and expose only necessary ports.

```
yaml
CopyEdit
networks:
backend:
driver: bridge
services:
db:
networks:
- backend
expose:
- "5432"
```

6. Restrict Capabilities and File Access

Apply read-only file systems and drop unneeded Linux capabilities.

```
yaml
CopyEdit
services:
db:
read_only: true
cap_drop:
- ALL
```

Regulatory Compliance

For regulated industries—such as healthcare (HIPAA), finance (PCI DSS), or companies operating in the EU (GDPR)—containerized databases must comply with strict data protection and auditability standards. Here's how Docker Compose configurations can help.

1. Encryption In-Transit and At-Rest

- Use SSL/TLS for client-database communication.
- Enable encryption features in the DB engine.
- Store volumes on encrypted file systems or use encrypted storage backends (e.g., Portworx, AWS EBS with encryption).

```
yaml
CopyEdit
services:
db:
image: postgres:16
environment:
POSTGRES_INITDB_ARGS: "--data-encryption=AES"
volumes:
- pgdata:/var/lib/postgresql/data
volumes:
pgdata:
driver_opts:
device: ":/secure/pgdata"
o: "encryption,aes256"
```

2. Access Controls and Auditing

- Use RBAC in database platforms.
- Log access attempts with audit plugins or tools like pgAudit for PostgreSQL.
- Configure Compose logs to external observability stacks for audit trails.

```
yaml
CopyEdit
services:
db:
logging:
driver: "json-file"
```

```
options:
max-size: "10m"
max-file: "3"
```

3. Data Minimization and Retention Policies

Use environment variables and automation to enforce data lifecycle policies and scrub PII in dev/test containers.

4. Data Residency and Backup Integrity

- Ensure volumes map to data centers in compliant regions.
- Automate backups and test recovery regularly.
- Use snapshot or immutable backups where supported.

```
yaml
CopyEdit
# Example: automated logical backup with volume mount
docker run --rm \
--volumes-from pg-db \
-v $(pwd)/backup:/backup \
postgres:16 \
bash -c "pg_dump -U $POSTGRES_USER $POSTGRES_DB >
/backup/db_backup.sql"
```

Summary: Secure by Design

Best Practice	Purpose	Compliance Mapping
Minimal base images	Reduce vulnerability surface	All frameworks
Non-root containers	Principle of least privilege	PCI DSS, HIPAA
Secrets in environment files	Secure credential handling	GDPR, SOC 2
Network segmentation	Prevent lateral movement	PCI DSS
Encrypted volumes & traffic	Protect data at rest and in transit	GDPR, HIPAA, PCI DSS
Auditing and logging	Ensure traceability and incident review	SOC 2, HIPAA

Container security isn’t optional—especially when databases are involved. By applying these security and compliance

measures in your Docker Compose setups, you create a safer foundation for modern, containerized data infrastructure.

Dockerfile vs. Docker Compose: FAQs

Does Docker Compose replace Dockerfile?

No—Docker Compose does not replace Dockerfile. Dockerfile is part of a process to build Docker images, which are part of containers, while Docker Compose is used for orchestrating.

Is docker-compose the same as Docker Compose?

Docker Compose is the name of the tool, while docker-compose is the name of the actual command—i.e., the code—used in Docker Compose.

Should you use Docker Compose in production?

Yes. Docker Compose works in all environments: production, staging, development, testing, as well as CI workflows.

Pure Storage Integration

Modern database workloads demand storage that’s fast, scalable, resilient, and easy to manage across environments. Pure Storage offers multiple solutions that bring enterprise-grade capabilities to containerized environments—whether you’re running on-prem, in the cloud, or across hybrid deployments. Here’s how to integrate Pure Storage with Docker Compose and take advantage of high-performance persistent storage for your databases.

1. Pure Storage CSI Driver

The Pure Storage Container Storage Interface (CSI) driver enables direct integration between container orchestrators (like Kubernetes) and Pure Storage platforms, such as FlashArray™ and FlashBlade®. While Docker Compose doesn’t natively support CSI, teams often use Compose for development and Kubernetes for production—making it essential to align volume strategies early.

For Docker Compose environments that transition to Kubernetes:

- Use similar volume naming and mount points in Compose as you'll define with CSI in production.
- Treat Compose as your development mirror and mount volumes in a way that mimics CSI-managed persistent volumes.

Kubernetes CSI example (production):

yaml
CopyEdit
services:
db:
image: postgres:16
volumes:
- pgdata:/var/lib/postgresql/data
volumes:
pgdata:
driver: local

Use this approach to maintain consistency across development and production environments when leveraging CSI.

2. FlashArray™ and FlashBlade® for Container Storage

FlashArray and FlashBlade provide ultra-low latency and high throughput—ideal for database workloads where IOPS and reliability are critical.

Benefits for containers:

- Consistent performance under load
- Inline deduplication and compression to save space
- Snapshots for fast backup and restore
- Native integration with the Pure CSI driver for Kubernetes environments

While Compose doesn't directly manage Pure volumes, you can mount FlashArray-backed directories using bind mounts or drivers like NFS or Fibre Channel on hosts that connect to Pure systems.

Example using FlashArray via NFS bind mount:

```
yaml
CopyEdit
services:
postgres:
image: postgres:16
volumes:
- /mnt/flasharray/postgres:/var/lib/postgresql/data
```

Best practices:

- Ensure /mnt/flasharray/postgres is mounted from Pure FlashArray using NFS or Fibre Channel.
- Monitor performance using Pure1® or the FlashArray GUI to correlate container performance with backend metrics.

3. Portworx® by Pure Storage

For production environments that require container-native storage across multi-node clusters, Portworx by Pure Storage is the leading choice. It offers advanced features like:

- Volume replication across nodes
- Application-aware backups and restores
- Disaster recovery and high availability
- CSI compliance and Kubernetes-native management

Key capabilities for database workloads:

- **PX-Store:** High-performance, scalable storage for databases like PostgreSQL, Cassandra, MongoDB.
- **PX-Backup:** Granular backup and restore with RBAC and policy-based management.
- **PX-DR:** Disaster recovery across regions or data centers.

Example: Portworx volume in Kubernetes (for production):

```
yaml
CopyEdit
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
name: mongodb-pvc
spec:
storageClassName: px-db
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 100Gi
```

Compose-local mirroring (dev/testing):

```
yaml
CopyEdit
services:
mongo:
image: mongo:7
volumes:
- mongo_data:/data/db
volumes:
mongo_data:
driver: local  # Replace with Portworx driver in Kubernetes
```

Summary: Why Pure for Containerized Databases?

Feature	FlashArray / FlashBlade	Portworx	Benefit for Containers
Performance & IOPS	✅ Yes	✅ Yes	Handles intensive DB workloads
Snapshots & Recovery	✅ Yes	✅ Yes	Instant backups and restores
Multi-node HA	❌	✅ Yes	Ensures DB resilience in Kubernetes
Kubernetes CSI Integration	✅ Yes	✅ Yes	Seamless storage orchestration
Dev-to-Prod Consistency	With bind/NFS	Native with CSI	Easy transition from Docker Compose
Backup and DR	Snapshots/manual	PX-Backup, PX-DR	Enterprise-grade data protection

Pure Storage delivers the storage reliability, speed, and resilience modern containerized databases need—whether you're

running on Docker Compose locally or orchestrating workloads in Kubernetes at scale.

Get started with Pure today. Contact us to learn more.