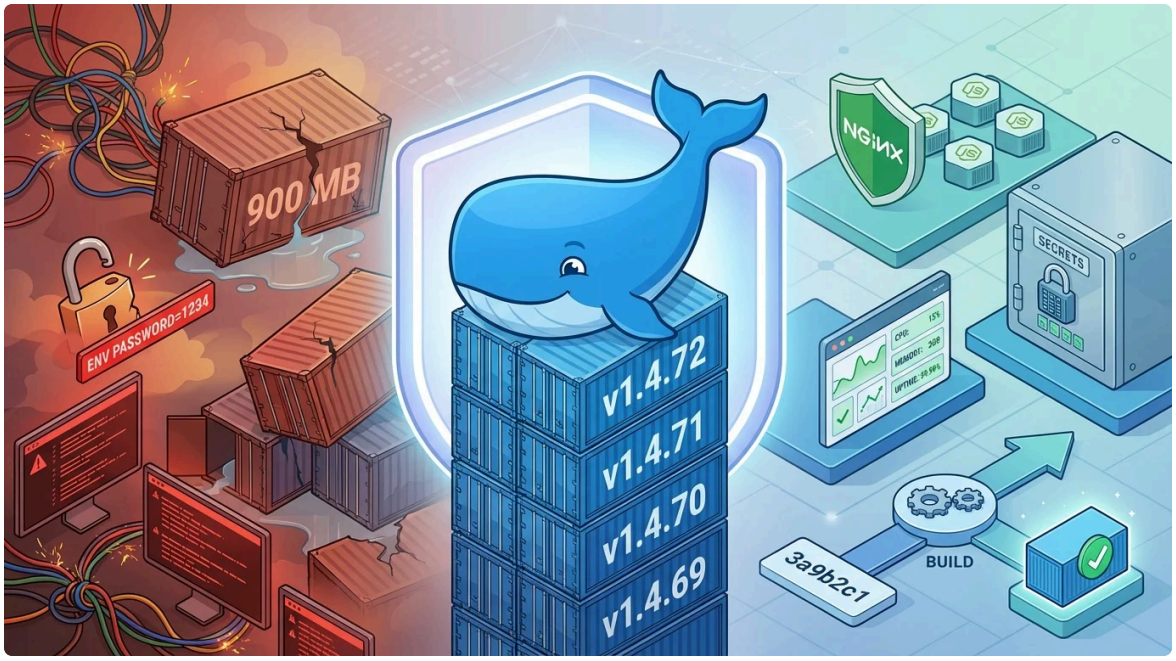


Docker Developer Workflow

From Local Development to Production-Ready Containers

FEBRUARY 22, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

From Local Development to Production-Ready Containers: Dockerfiles, Compose, Secrets, Security, Versioning, Debugging, and the Architecture Decisions That Matter

Join the discussion at r/docker_dev

TL;DR: This is the complete Docker guide for developers - not just deployment, but everything you actually do day-to-day. How to set up Docker Desktop properly. How to write Dockerfiles that produce 30 MB images instead of 900 MB ones. How to structure compose files that work identically on your laptop and in Swarm. How to handle secrets without leaking them into environment variables, logs, or image layers. How to version every build so you can trace a production bug back to the exact git commit. Why your Node.js container shouldn't be doing SSL, compression, or rate limiting (and what should). How to troubleshoot when things break. How to stop running as root. How to fix invisible line ending bugs. How to set up logging that doesn't fill your disk. After years of running Docker in production and development, these are the workflows, patterns, and architecture decisions that actually matter.

I've written extensively about Docker Swarm in production - the architecture, the compose files, the deployment pipelines. But I keep seeing the same pattern: developers read the production guide, get Swarm running on their VPS, and then struggle with everything else. They don't know how to efficiently debug a container that won't start. They rebuild entire images when they change one line of code. They have no idea why their service works locally but fails in Swarm. They put database passwords in environment variables and ship 300 MB containers full of middleware that belongs in NGINX. They've never opened Docker Desktop's built-in tools because nobody told them those tools exist.

This guide covers the full picture - from the first line of your Dockerfile to the versioning system that tells you exactly which build introduced a bug at 2 AM. The development workflow, the architecture decisions, and the security practices that separate teams who ship confidently from teams who debug constantly.

TABLE OF CONTENTS

DOCKER DESKTOP: STOP TREATING IT LIKE AN INSTALLER

Most developers install Docker Desktop, verify `docker --version` works, and never open the application again. That's a mistake. Docker Desktop is a full development environment, and understanding its settings will save you hours of debugging that you'd otherwise blame on Docker itself.

Resource Allocation - The #1 Performance Issue

Open Docker Desktop -> Settings -> Resources.

Docker Desktop runs inside a lightweight VM (even on Linux now, though Linux users can opt out). That VM gets a fixed slice of your system's CPU, RAM, and disk. The defaults are conservative - usually 2 CPUs, 2-4 GB RAM, and a 64 GB virtual disk.

Here's what happens with the defaults: you spin up a Node.js container, a MongoDB container, maybe an NGINX container and an Elasticsearch instance for local development. MongoDB alone wants 1-2 GB. Elasticsearch wants another 2 GB. Your Node app needs headroom. You've already exceeded your allocation before you've written a line of code. Docker doesn't crash - it slows to a crawl. Containers take 30 seconds to start. Builds hang. `npm install` inside a container times out. You blame Docker. Docker is fine. You're starving it.

My recommendation for development machines:

RESOURCE	MINIMUM	RECOMMENDED	NOTES
CPUs	4	6-8	Match roughly half your physical cores
Memory	6 GB	8-12 GB	Depends on your stack
Swap	1 GB	2 GB	Safety net for memory spikes
Disk	64 GB	128 GB	Images and volumes add up fast

If you're running a database, a search engine, and your application containers simultaneously - which most full-stack developers do - 4 GB of RAM is not enough. Period. Bump it to 8 GB minimum and stop wondering why everything is slow.

Docker Desktop Features You Should Actually Use

Docker Scout - Built into Desktop, this scans your images for known vulnerabilities. Run `docker scout cves myimage:latest` or just check the Images tab in Desktop. You'd be surprised how many critical CVEs are sitting in your base images.

This isn't a production concern - it's a development concern. If your local image has vulnerabilities, your production image will too.

Container Logs in the GUI - Click any running container in Docker Desktop and you get live-streaming logs with search and filtering. Most developers still do `docker logs -f container_name` in the terminal, which is fine, but the GUI lets you search across logs instantly without piping through grep. Use both.

Resource Usage Dashboard - Docker Desktop shows real-time CPU and memory per container. When your machine is sluggish, open this before blaming your IDE. You'll often find a container eating 3 GB of RAM because you didn't set memory limits - even in development.

Docker Init - Run `docker init` in your project directory and Docker will generate a Dockerfile, compose file, and .dockerignore tailored to your detected language/framework. It won't be production-perfect, but it's a better starting point than most tutorials.

Extensions - Docker Desktop has an extension marketplace. The most useful ones: Disk Usage (find what's eating your disk), Logs Explorer (aggregate logs across containers), and Resource Usage (detailed per-container metrics). Don't install twenty extensions - install two or three that solve problems you actually have.

The `.dockerignore` File Nobody Writes

Every project should have a `.dockerignore` file. Without one, `docker build` sends your entire project directory as the build context - including `node_modules`, `.git`, test fixtures, local databases, IDE files, and everything else. I've seen build contexts exceed 2 GB because someone forgot this file.

```
# .dockerignore
node_modules
npm-debug.log*
.git
.gitignore
.env
.env.*
*.md
LICENSE
.vscode
.idea
coverage
test
tests
__tests__
.nyc_output
dist
build
*.log
.DS_Store
Thumbs.db
docker-compose*.yml
Dockerfile*
```

This alone can cut your build time from minutes to seconds. The build context is sent to the Docker daemon before the build even starts. Every megabyte counts.

COMPOSE FILES: WHAT WORKS IN DOCKER VS WHAT WORKS IN SWARM

This is the section that will save you a weekend of debugging when you deploy to Swarm for the first time. Docker Compose and Docker Swarm both read compose files, but they interpret them differently. Directives you rely on every day in local development are silently ignored in Swarm. Features Swarm requires don't exist in regular Compose. And some patterns that work perfectly on your laptop will actively break a Swarm deployment.

I'm going to cover all of it - what gets ignored, what breaks, what changes, and why. But first, we need to talk about the mental model that causes most of the problems in the first place.

Stop Thinking of Containers as Computers

This is the single biggest mindset problem I see in developers coming to Docker. They treat a container like a virtual machine. Like a tiny computer. They give it a name, a fixed IP address, a persistent filesystem, and they SSH into it to check things. They configure their application to connect to `172.18.0.5` because that's the IP their database container got. They write files to the container's filesystem and expect them to be there tomorrow.

This mental model will destroy you in Swarm.

A container is not a computer. A container is a **process**. It runs, it does its job, and it can be killed and replaced at any moment. Swarm kills containers all the time - during rolling updates, during node failures, during scaling events, during rebalancing. Your container that's running right now might not be the same container that's running in 30 seconds. Swarm tore it down and spun up a new one because you pushed a new image. Or because the node it was on ran out of memory. Or because you scaled from 3 replicas to 6 and Swarm rebalanced.

When that new container starts, it has:

- A **new** container ID
- A **new** IP address
- A **new** hostname
- A **blank** filesystem (unless you mounted a volume)
- **Zero** knowledge of the container it replaced

If your application stored session data in local files, it's gone. If another service was connecting to it by IP address, that connection is broken. If it was writing logs to its own filesystem without a volume mount, those logs are gone. If it was caching data in memory, that cache is cold.

This is not a bug. This is the entire point. **Containers are disposable by design.** Swarm's power comes from the fact that it can tear down and recreate containers instantly without anyone noticing. But that only works if your containers are *stateless*.

What Stateless Actually Means

Stateless doesn't mean your application can't have state. Obviously your app has a database, user sessions, uploaded files, and configuration. Stateless means **the container itself doesn't hold any of that**. The state lives somewhere else - in a database, in a mounted volume, in an external cache, in environment variables - and the container accesses it at runtime.

Here's the test: if Swarm kills your container right now and starts a new one, does your application work identically? If yes, your container is stateless. If no, you have state leaking into the container and you need to move it out.

WHERE STATE SHOULD NOT LIVE	WHERE IT SHOULD LIVE INSTEAD
Files written to the container filesystem	Docker volumes or external storage (S3, EFS)
In-memory sessions	A database or Redis
Uploaded files on disk inside the container	Mounted volume or object storage
Application config files baked in at runtime	Environment variables or Docker Configs/Secrets
Hardcoded connection strings with IP addresses	Service names resolved by Docker DNS
Local SQLite databases	A proper database service (MongoDB, PostgreSQL)
Application logs written to local files	stdout/stderr (Docker captures them) or a volume-mounted log directory

Every row in that left column is something I've seen developers do. Every one of them breaks in Swarm.

Why Fixed IP Addresses Break Everything

In local development, you can assign static IPs to containers:

```
# This works locally. It will RUIN you in Swarm.
services:
  app:
    networks:
      mynet:
        ipv4_address: 10.5.0.10
  mongo:
    networks:
      mynet:
        ipv4_address: 10.5.0.20

networks:
  mynet:
    driver: bridge
    ipam:
      config:
        - subnet: 10.5.0.0/16
```

Developers do this because it feels familiar. You know where everything is. Your app connects to `10.5.0.20` and it just works. But here's what you've actually done: you've welded your services to specific addresses on a specific network on a specific machine. You now can't:

- **Run replicas** - Where does the second replica go? `10.5.0.10` is taken. Swarm can't assign the same IP to two containers.
- **Run on multiple nodes** - Node 2 might not even have the `10.5.0.0/16` subnet. Overlay networks span nodes, but they assign IPs dynamically. You don't control them.
- **Survive a container restart** - Swarm tears down the old container and creates a new one. The new container might get `10.5.0.47`. Every service that was connecting to `10.5.0.10` is now pointing at nothing.
- **Let Swarm load-balance** - When you have 6 replicas of your app, Swarm's internal load balancer distributes traffic across all 6 using the *service name*. If you hardcoded an IP, you're hitting one specific container. Five replicas sit idle.

But there's a problem that's even worse than all of those, and it bites you even if you only have a single replica. It happens during the most routine operation in Swarm: **a rolling update**.

The Rolling Update IP Problem

When you push a new image and Swarm does a rolling update, here's what happens step by step:

1. Swarm starts a **new** container with the new image
2. Swarm waits for the new container to pass its health check
3. Swarm stops the **old** container
4. The old container shuts down and **releases** its IP address
5. Docker's network stack reclaims the IP

Now imagine your old container had a fixed IP of `10.5.0.10`. The old container is shutting down, but shutdown isn't instant - your application needs time to close connections, flush buffers, finish in-flight requests. During that window, the IP `10.5.0.10` is in limbo. The old container still has it. The new container needs it. And depending on timing, you can end up in a state where:

- The new container can't start because the IP is still held by the dying container
- Two containers briefly claim the same IP and network traffic goes to the wrong one
- The IP gets released and immediately reassigned, but Docker's internal DNS cache hasn't updated yet, so other services are sending traffic to an address that's in the middle of a handoff
- The entire update hangs because the new container is waiting for an IP that the old container won't release until it finishes shutting down, but the old container won't shut down until the new one is healthy - a deadlock

This isn't a theoretical concern. I've seen it happen. The update hangs, the service goes unhealthy, Swarm tries to roll back, and the rollback hits the same IP conflict. Now you're dead in the water and the only fix is to manually remove containers and redeploy.

And this is with just **one** replica. With one container. The simplest possible deployment. You didn't even need replicas for the fixed IP to break you. You just needed to update your image - the most common operation in all of Docker.

With dynamic IPs, none of this happens. Swarm assigns whatever IP is available to the new container. The old container keeps its IP until it's fully shut down. There's no conflict, no race condition, no deadlock. The new container comes up on `10.5.0.47`, Docker DNS updates the service name resolution, and every other service that connects by name seamlessly starts talking to the new container. The old one fades away quietly. This is how Docker was designed to work.

The Deeper Problem: Thinking in Addresses Instead of Names

Fixed IPs are a symptom of a deeper issue - thinking about containers the way you think about servers. On a traditional server, the machine has an IP, the IP doesn't change, and you configure everything to point at that IP. Firewall rules reference IPs. Config files have IPs. Monitoring tools watch IPs. The IP *is* the identity of the machine.

In Docker, the **service name** is the identity. Not the IP, not the container ID, not the hostname. The service name. It's the one thing that never changes across updates, restarts, scaling events, and node failures. Docker DNS resolves the service name to wherever the container currently lives. That's the abstraction layer. That's what makes orchestration work.

Every time you hardcode an IP in a connection string, an environment variable, a config file, or an `/etc/hosts` entry, you're bypassing that abstraction. You're telling Docker "I don't trust your DNS, I'll handle routing myself." And the moment Swarm moves a container - which it will, because that's its job - your manual routing breaks.

The correct approach is to **never reference a container by IP address**. Use service names. Docker's internal DNS server at `127.0.0.11` resolves service names to the correct container (or containers, in the case of replicas) automatically.

```
# CORRECT - this works in both Docker Compose and Swarm
services:
  app:
    environment:
      - MONGO_URI=mongodb://mongo:27017/mydb  # "mongo" is the service name
    networks:
      - app-network

  mongo:
    image: mongo:7
    networks:
      - app-network

networks:
  app-network:
```

Your application connects to `mongo`, not `10.5.0.20`. Docker resolves `mongo` to whatever IP the MongoDB container currently has. If Swarm restarts the container and it gets a new IP, Docker DNS updates automatically. Your application never knows the difference.

This is why I said in the production guide: **never hardcode IP addresses in Docker. Ever. Let Docker handle routing via service names.** It's not just a best practice - it's the only approach that survives a Swarm deployment.

The Complete Ignored Directives List

Here is every compose directive that `docker stack deploy` (Swarm) silently ignores. "Silently" is the key word - Swarm doesn't throw an error. It just skips them. Your compose file deploys, your containers start, and something doesn't work the way you expected. You spend three hours debugging before realizing Swarm never even read that directive.

DIRECTIVE	WHAT IT DOES LOCALLY	WHY SWARM IGNORES IT
`build`	Builds an image from a Dockerfile	Swarm only runs pre-built images from registries. No building in production.
`container_name`	Assigns a fixed name to the container	Swarm names containers dynamically: `stackname_service_replica.id`. Fixed names would collide with replicas.
`depends_on`	Controls startup order between services	Swarm has no startup ordering. Services start in parallel. Your app must handle missing dependencies.
`links`	Creates network aliases between containers	Deprecated since Docker 1.9. Use shared networks instead. Swarm doesn't support it.
`restart`	Docker engine restart policy	Swarm uses `deploy.restart_policy` instead - a different mechanism managed by the orchestrator, not the engine.
`networks.ipv4_address`	Assigns a static IP to a container	Swarm manages IPs dynamically across nodes. Static IPs break replicas, load balancing, and multi-node deployments.
`networks.ipv6_address`	Assigns a static IPv6 address	Same reason as above.
`network_mode`	Sets the network mode (host, bridge, etc.)	Swarm manages networking via overlay networks. Custom network modes conflict with orchestration.
`cap_add`, `cap_drop`	Add/remove Linux capabilities	Ignored in stack deploy. Use `deploy.placement.constraints` or configure at the engine level.

`cgroup_parent`	Specifies a parent cgroup	Swarm manages cgroups for task isolation.
`devices`	Maps host devices into the container	Swarm doesn't support device mappings - security and portability reasons.
`tmpfs`	Mounts a temporary filesystem	Use `deploy.placement` and volumes instead in Swarm.
`extra_hosts`	Adds entries to /etc/hosts	Ignored in Swarm. Use Docker DNS or configure your internal DNS server.
`sysctls`	Sets kernel parameters	Ignored in Swarm for security - kernel params affect the whole host.
`userns_mode`	Sets user namespace mode	Swarm manages namespaces at the engine level.
`security_opt`	Security options (AppArmor, seccomp)	Configure these at the Docker engine level on each node, not in the compose file.

That's a long list. And every single one of these will bite you if you depend on it locally and then deploy to Swarm expecting it to work.

Directives That Change Behavior in Swarm

Some directives work in both Docker Compose and Swarm, but behave differently:

`ports` - In Compose, `ports: - "3000:3000"` maps container port 3000 to host port 3000. Straightforward. In Swarm, port publishing works through the *routing mesh*. When you publish a port, it's published on *every node in the Swarm*, even nodes that aren't running that service. Traffic hitting port 3000 on any node gets routed to a container running the service, wherever it lives. This is powerful - but it means port conflicts affect the entire Swarm, not just one host.

You can also publish just the container port without mapping to a host port:

```
# Swarm-style: Docker assigns a random host port
ports:
  - "61339"

# Compose-style: explicit host-to-container mapping
ports:
  - "3000:3000"
```

In Swarm, publishing only the container port lets Docker assign high-numbered host ports dynamically. Your reverse proxy (NGINX) then routes to the service by name - it never needs to know which host port was assigned.

`volumes` - Named volumes work in both modes. But bind mounts (mapping a host directory into the container) are problematic in Swarm because different nodes have different filesystems. If your container gets scheduled on Node 2, the file path from Node 1 doesn't exist. Use named volumes for persistence and Docker Configs/Secrets for configuration files.

```
# This breaks in multi-node Swarm - the path doesn't exist on other nodes
volumes:
  - ./config/app.json:/app/config.json

# This works everywhere - named volume managed by Docker
volumes:
  - app-data:/app/data

# This works everywhere - Docker Secret mounted as a file
secrets:
  - source: app_config
    target: /app/config.json
```

`networks` - In Compose, the default driver is `bridge` (single host). In Swarm, you need `overlay` (multi-host). Swarm can only use overlay networks. If you try to deploy a stack with a bridge network, the services won't be able to communicate across nodes.

Directives That Only Work in Swarm

The `deploy` block is the big one. In regular Docker Compose, the `deploy` key is partially respected (resource limits work), but the orchestration features - replicas, rolling updates, rollback, placement constraints - only function in Swarm mode.

```
deploy:
  # How many instances of this container to run
  mode: replicated
  replicas: 6

  # Spread containers across nodes
  placement:
    max_replicas_per_node: 3
    constraints:
      - node.role == worker          # Only run on worker nodes
      - node.labels.db == true       # Only run on nodes with this label

  # How to update when you push a new image
  update_config:
    parallelism: 2          # Update 2 containers at a time
    delay: 10s              # Wait 10s between batches
    failure_action: rollback  # If new containers fail, roll back
    order: start-first      # Start new before stopping old (zero downtime)

  # How to roll back if an update fails
  rollback_config:
    parallelism: 2
    delay: 10s

  # When and how to restart failed containers
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s

  # CPU and memory limits
  resources:
    limits:
      cpus: '0.50'
      memory: 400M
    reservations:
      cpus: '0.20'
      memory: 150M
```

None of the orchestration pieces - `replicas`, `update_config`, `rollback_config`, `placement` - do anything in plain Docker Compose. If you're developing locally with `docker compose up`, those values are in the file but not active. That's fine. The point is that they're *there* when you deploy to Swarm.

Docker Secrets are also Swarm-only in the traditional sense:

```
# Secrets are first-class in Swarm
secrets:
  db_password:
    external: true    # Created via Portainer or docker secret create

services:
  app:
    secrets:
      - db_password   # Mounted as /run/secrets/db_password inside the containe
```

In Swarm, secrets are encrypted at rest, encrypted in transit, and mounted as in-memory files that never touch disk inside the container. In regular Compose, you can simulate secrets with file-based secrets, but they don't have the same encryption guarantees.

The Complete Side-by-Side Reference

Here's the full picture - everything in one table so you can see at a glance what works where:

DIRECTIVE	DOCKER COMPOSE	DOCKER SWARM	NOTES
`image`	Yes	Yes	**Required** in Swarm (no build in production)
`build`	Yes	No	Keep it for local dev, but always also specify `image`
`container_name`	Yes	No	Breaks replicas - let Swarm name containers
`depends_on`	Yes	No	Build retry logic into your application
`restart`	Yes	No	Use `deploy.restart_policy` instead
`links`	~ Deprecated	No	Use shared networks, never links
`network_mode`	Yes	No	Swarm manages networking via overlay
`networks.ipv4_address`	Yes	No	Use service names, never static IPs
`extra_hosts`	Yes	No	Use DNS server or Docker DNS
`cap_add/cap_drop`	Yes	No	Configure at engine level
`devices`	Yes	No	Security/portability restriction
`sysctls`	Yes	No	Engine-level config
`security_opt`	Yes	No	Engine-level config
`tmpfs`	Yes	No	Use volumes
`ports`	Yes	Yes (routing mesh)	Behavior changes - published on ALL nodes in Swarm

`volumes` (named)	Yes	Yes	Works in both - use this for persistence
`volumes` (bind mount)	Yes	~ Problematic	Path must exist on every node the service runs on
`environment`	Yes	Yes	Works identically in both
`env_file`	Yes	Yes	Works in both
`healthcheck`	Yes	Yes	Works in both - **critical** for Swarm rolling updates
`init`	Yes	Yes	Works in both - always use this
`dns`	Yes	Yes	Works in both - important for internal DNS servers
`networks`	Yes (bridge)	Yes (overlay)	Driver changes: bridge locally, overlay in Swarm
`deploy.replicas`	~ Partially	Yes	Only meaningful in Swarm
`deploy.resources`	Yes	Yes	Works in both - always set limits
`deploy.update_config`	No	Yes	Swarm rolling update strategy
`deploy.rollback_config`	No	Yes	Swarm automatic rollback
`deploy.placement`	No	Yes	Controls which nodes run the service
`deploy.restart_policy`	Yes	Yes	Use this instead of `restart`
`secrets`	~ File-based only	Yes	Full encryption only in Swarm
`configs`	~ Limited	Yes	Swarm-native config management

Print this table. Tape it to your monitor. It will save you more debugging time than any other single piece of information in this article.

Building Habits That Work in Both Worlds

The goal isn't to write two different compose files - one for development and one for Swarm. The goal is to write one compose file that works in both, with minimal environment-specific overrides.

Here's how you break the bad habits:

Stop using `container_name`. You don't need it. Docker Compose generates readable names automatically (`projectname-service-1`). If you need to reference a container, use the service name for network communication and `docker compose logs servicename` for debugging. Hardcoded container names are a crutch from the VM era.

Stop using `depends_on` as a safety net. Instead, make your application handle dependency unavailability. Every service should retry its connections with exponential backoff. This isn't just a Swarm requirement - it's good engineering. Even in production without Swarm, databases restart. Connections drop. Networks hiccup. Your app should handle it gracefully.

```
// Connection retry with exponential backoff
const connectWithRetry = async (uri, maxRetries = 10) => {
  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      await mongoose.connect(uri);
      console.log('MongoDB connected');
      return;
    } catch (err) {
      const delay = Math.min(attempt * 2, 30);  // Cap at 30 seconds
      console.log(`MongoDB attempt ${attempt}/${maxRetries} failed. Retrying in
      await new Promise(resolve => setTimeout(resolve, delay * 1000));
    }
  }
  throw new Error('Failed to connect to MongoDB after maximum retries');
};
```

Stop using `restart: always`. Use `deploy.restart_policy` from day one. It works in both Compose and Swarm:

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

Stop using static IPs. Connect to services by name. Configure your database connections as `mongodb://mongo:27017/mydb`, your Redis connections as `redis://redis:6379`, your Elasticsearch connections as `http://elasticsearch:9200`. Service names. Always.

Stop using bind mounts for configuration. Use environment variables for simple config. Use Docker Secrets for sensitive data. Use Docker Configs for configuration files. All three work in Swarm. Bind mounts to host paths don't.

Stop writing to the container filesystem. Write logs to stdout/stderr - Docker captures them and you can read them with `docker logs` or `docker service logs`. Write persistent data to named volumes. Write temporary data to `/tmp`. Anything written to the container's writable layer vanishes when the container is replaced.

Always write a healthcheck. Swarm uses health checks for everything - rolling updates, rollback decisions, service monitoring. Without a health check, Swarm treats every running container as healthy, even if your application is deadlocked or has lost its database connection. A health check is the difference between Swarm detecting a problem in 30 seconds and you detecting it when a customer emails you.

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 30s
```

A Compose File That Works Everywhere

Here's a compose file structured for both local development and Swarm deployment. Every directive is intentional:

```
services:
  nodeserver:
    # build is for local dev - Swarm ignores it and uses image
    build:
      context: ./nodeserver
      args:
        - BUILD_VERSION=dev

    # image is for Swarm - also used locally if you don't --build
    image: "yourregistry/nodeserver:latest"

    # Signal handling - works in both modes
    init: true

    environment:
      - NODE_ENV=production
      - MONGO_URI=mongodb://mongo:27017/mydb

    # deploy block - orchestration features only activate in Swarm,
    # but resource limits work in both modes
    deploy:
      replicas: 1    # 1 for dev, increase for production
      resources:
        limits:
          memory: 400M
          cpus: '0.50'
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3

      # These only activate in Swarm (harmless in Compose)
      update_config:
        parallelism: 2
        delay: 10s
        failure_action: rollback
        order: start-first
      rollback_config:
        parallelism: 2
        delay: 10s

    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

```
      start_period: 30s

    ports:
      - "3000:3000"

    networks:
      - app-network

  mongo:
    image: mongo:7
    volumes:
      - mongo-data:/data/db     # Named volume - works in both modes
    networks:
      - app-network

volumes:
  mongo-data:

networks:
  app-network:
    driver: bridge    # Change to overlay (or use external) for Swarm
```

For production, you only need to change the network configuration and potentially the replica count. Everything else - the service names, the connection strings, the health checks, the resource limits, the restart policy - carries over unchanged.

The developers who have the smoothest deployments are the ones who stopped treating Docker Compose and Docker Swarm as two different systems. They're the same system. Compose is the single-node version. Swarm is the multi-node version. Write your compose files for the multi-node version from day one, and the single-node version just works.

The Network Transition

Locally you use `bridge` networks. In Swarm you use `overlay`. This is the one significant change between dev and prod:

```
# Development
networks:
  app-network:
    driver: bridge

# Production (use a pre-created external network)
networks:
  app-network:
    driver: overlay
    external: true
    name: awsnet
```

Everything else - service names, port mappings, environment variables - stays identical. Your app connects to `mongodb://mongo:27017/mydb` in both environments. Docker's internal DNS at `127.0.0.11` resolves `mongo` to the right container whether you're on a bridge network locally or an overlay network in Swarm.

THE DOCKERFILE: A COMPLETE WALKTHROUGH

This is the section most guides rush through, and it's the section that matters most. Your Dockerfile is the blueprint for your container. Every instruction in it creates a layer in the final image. Get this wrong and you'll ship 1 GB images full of compilers and test frameworks. Get it right and you'll ship 30-40 MB images that start in milliseconds and have almost no attack surface.

Let's start from zero and build up to a production-ready multi-stage Dockerfile, explaining every single line.

What a Dockerfile Actually Does

A Dockerfile is a recipe. Each instruction is a step. Docker executes them top to bottom, and each step creates a new *layer* - a snapshot of the filesystem at that point. The final image is all those layers stacked on top of each other.

Think of it like painting a wall. The base coat is your operating system. Then you paint on your tools. Then your dependencies. Then your application code. Each coat covers or adds to what's below. The finished wall is your Docker image.

Here's the simplest possible Dockerfile for a Node.js app:

```
FROM node:20
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "server.js"]
```

This works. Your app runs. But this image is about **1 GB**. It contains the full Debian operating system, every system tool, the entire npm cache, your devDependencies (jest, eslint, nodemon - all of it), your `.git` folder if you forgot `.dockerignore`, and enough attack surface to make a security auditor cry.

Let's fix it. One concept at a time.

Every Dockerfile Instruction Explained

Before we get into multi-stage builds, you need to understand what each instruction does. Here's the complete vocabulary:

`FROM` - The starting point. Every Dockerfile begins with `FROM`. It tells Docker which base image to build on top of. Think of it as choosing which blank canvas to paint on. `FROM node:20-bookworm-slim` starts you with a minimal Debian system that already has Node.js 20 installed. `FROM python:bookworm` gives you Python. `FROM nginx:1.27` gives you Nginx. You can use `FROM` multiple times in a single Dockerfile - that's what multi-stage builds are.

`WORKDIR` - Sets the working directory inside the container. Every instruction after this runs relative to this path. If you write `WORKDIR /app`, then `COPY . .` copies files into `/app`. Without `WORKDIR`, everything lands in `/` (the root), which is messy and hard to debug. Always set this.

`COPY` - Copies files from your local machine (the "build context") into the image. `COPY package.json ./` copies your package.json into the WORKDIR. `COPY . .` copies everything. The *order* of your COPY instructions matters enormously for build speed - we'll get to that.

`RUN` - Executes a command inside the image during build. `RUN npm install` installs your dependencies. `RUN apt-get update && apt-get install -y python3` installs system packages. Each `RUN` creates a new layer. Combine related commands with `&&` to minimize layers.

`ENV` - Sets environment variables that persist in the running container. `ENV NODE_ENV=production` means your app sees `process.env.NODE_ENV === 'production'`.

`ARG` - Build-time variables. Unlike `ENV`, these exist *only during the build* and are not present in the running container. Use `ARG BUILD_VERSION=1.0.0` in the Dockerfile, then `docker build --build-arg BUILD_VERSION=1.2.3 .` at build time. Perfect for version numbers, git commit hashes, or toggling build behavior.

`EXPOSE` - Documents which port the container listens on. This is metadata - it doesn't actually publish the port. You still need `-p 3000:3000` in your `docker run` or `ports:` in your compose file. But it's good practice because tools like Docker Desktop use it to show you what's available.

`CMD` - The default command that runs when the container starts. `CMD ["node", "server.js"]` means the container runs your Node app. Can be overridden at runtime with `docker run myimage some-other-command`.

`ENTRYPOINT` - Like `CMD`, but harder to override. When you set `ENTRYPOINT ["node", "server.js"]`, the container *always* runs that command. `CMD` arguments get appended to `ENTRYPOINT`. The distinction matters: use `ENTRYPOINT` when the container should always run your app, use `CMD` when you might want to override it (like running a shell for debugging). In production images, `ENTRYPOINT` is the right choice.

`LABEL` - Metadata. `LABEL version="1.2.3"` adds a label you can query with `docker inspect`. Useful but not critical.

`USER` - Switches to a different user inside the container. By default everything runs as root. `USER appuser` switches to a non-root user. This is a security best practice - if your container gets compromised, the attacker doesn't have root.

Why Multi-Stage Builds Change Everything

Here's the problem with a single-stage Dockerfile. To install npm packages with native dependencies - things like `bcrypt` for password hashing, `sharp` for image processing, `canvas` for server-side rendering - npm needs to *compile C/C++ code*. That compilation requires `python3`, `make`, and `g++` (the GNU C++ compiler). So your Dockerfile installs those tools, runs `npm install`, and the packages compile successfully.

But now those tools are *in your image*. Permanently. Python, a C++ compiler, make, all their system libraries - they're baked into every container that runs from this image. Your 40 MB Node.js application is now sharing space with 300+ MB of build tools it will *never use again*. And if an attacker somehow gets into your container, you've helpfully provided them with a compiler.

Multi-stage builds solve this completely. The idea is simple: **use one stage to build, and a different stage to run.**

Think of it like a construction site. Stage 1 is the construction zone - cranes, scaffolding, welding equipment, cement mixers. You need all of that to build the building. Stage 2 is the finished building - clean, furnished, ready for tenants. You don't leave the cranes inside the lobby. You don't keep the cement mixer in the parking garage. The construction equipment served its purpose and gets left behind.

That's exactly what multi-stage builds do. Let me walk through a complete production Dockerfile, line by line:

```
# syntax=docker/dockerfile:1

# ================================================
# STAGE 1: The construction zone
# This stage exists ONLY to compile dependencies.
# Nothing from this stage ships in the final image
# except what we explicitly copy out.
# ================================================
FROM node:20-bookworm-slim AS builder

WORKDIR /app

# Install the construction equipment.
# python3, make, and g++ are needed because packages like
# bcrypt and sharp contain C/C++ code that must be compiled
# during npm install. These tools are ~300MB. They will NOT
# be in our final image.
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    python3 \
    make \
    g++ \
    && apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Copy ONLY the package files first.
# Why not copy everything? Because of layer caching.
# Docker caches each layer. If nothing in this layer changed
# (same package.json, same lock file), Docker reuses the cache
# and skips the expensive npm install below.
# If we copied ALL source code here, changing one line of
# JavaScript would invalidate this cache and force a full
# npm install every single time.
COPY package.json package-lock.json ./

# Install ONLY production dependencies.
# --omit=dev means: skip jest, eslint, nodemon, prettier,
# and every other devDependency. They're for your laptop,
# not for production.
# npm ci (clean install) is faster and more reliable than
# npm install - it deletes node_modules first and installs
# exactly what's in the lock file.
RUN npm ci --omit=dev

# At this point, this stage has:
# Yes Compiled, production-ready node_modules
```

```
# No python3, make, g++ (300+ MB of build tools)
# No apt cache and package lists
# We want the node_modules. We don't want anything else.

# ==========================================
# STAGE 2: The finished building
# This is what actually ships. Clean base image,
# no build tools, no compilers, no junk.
# ==========================================
FROM node:20-bookworm-slim AS production

# Set the timezone so your logs have correct timestamps
RUN ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime \
    && echo America/New_York > /etc/timezone

WORKDIR /app

# HERE'S THE MAGIC LINE.
# COPY --from=builder means: reach back into Stage 1 (named "builder")
# and copy ONLY /app/node_modules into this fresh, clean image.
# The build tools, the apt cache, the compilers - none of that
# comes along. Just the finished, compiled packages.
COPY --from=builder /app/node_modules /app/node_modules

# Now copy your actual application source code.
# This is separate from node_modules for caching reasons:
# your code changes constantly, but node_modules only changes
# when you add/remove packages.
COPY . .

# Document the port (doesn't publish it, just metadata)
EXPOSE 3000

# Run the app. ENTRYPOINT means this always runs - you can't
# accidentally override it with docker run arguments.
# --trace-warnings shows where deprecation warnings originate
# which is invaluable for debugging.
ENTRYPOINT ["node", "--trace-warnings", "./server.js"]
```

What's in the final image: Node.js runtime, your compiled production dependencies, and your application code. That's it.

What's NOT in the final image: Python, make, g++, the apt package cache, devDependencies, build artifacts, intermediate files. All of that existed only in Stage 1, which gets thrown away after the build.

The Size Difference Is Staggering

Let me put real numbers on this. Here's what a typical Node.js app looks like with and without multi-stage builds:

WHAT'S IN THE IMAGE	SINGLE STAGE	MULTI-STAGE
Base OS + Node.js	~200 MB	~200 MB
python3 + make + g++	~300 MB	**0 MB**
node_modules (prod)	~40 MB	~40 MB
node_modules (dev)	~120 MB	**0 MB**
apt cache & lists	~50 MB	**0 MB**
Your application code	~5 MB	~5 MB
Total	**~715 MB**	**~245 MB**

And that's before you start being smart about what goes into your application code. My production Node containers are 26-38 MB because the code itself is lean - no middleware stacks doing what NGINX should do, no bundled frontend assets, no test fixtures, no documentation files.

The size matters for three reasons: faster deployments (Swarm pulls the image to every node), faster rollbacks (smaller image = faster download = faster recovery), and smaller attack surface (no compilers means an attacker can't compile exploit tools inside your container).

What to COPY in Which Stage - The Decision Framework

This is where people get confused. Here's a simple rule:

Stage 1 (builder) gets: Everything needed to *compile* your dependencies. Package files, build tools, native compilation toolchains. This stage's job is to produce a folder of compiled, production-ready packages.

Stage 2 (production) gets: The compiled output from Stage 1, plus your application code. Nothing else.

Here's how to think about each file in your project:

FILE / DIRECTORY	WHICH STAGE?	WHY
`package.json`	Stage 1 (COPY)	Needed for `npm ci`
`package-lock.json`	Stage 1 (COPY)	Needed for `npm ci`
`node_modules/`	Stage 2 (COPY --from)	Compiled result from Stage 1
`server.js`, `src/`, `routes/`	Stage 2 (COPY . .)	Your application code
`python3`, `make`, `g++`	Stage 1 only (RUN apt-get)	Build tools - never ship these
`.env`	Neither	Should be in `.dockerignore`
`tests/`, `__tests__/`	Neither	Should be in `.dockerignore`
`.git/`	Neither	Should be in `.dockerignore`
`node_modules/` (local)	Neither	Should be in `.dockerignore` - use the container's

The key mental model: **Stage 1 is a factory. Stage 2 is the product. You ship the product, not the factory.**

Layer Caching - Why the Order of Every Line Matters

Docker caches each layer. When you rebuild, Docker checks each instruction top to bottom. If the instruction and its inputs haven't changed, Docker reuses the cached layer and skips it. The moment something *does* change, that layer and *every layer after it* rebuilds.

This means the order of your Dockerfile instructions is a performance decision. Put things that change rarely at the top. Put things that change often at the bottom.

```
# BAD ORDER - every code change triggers a full npm install
COPY . .                          # <- changes every time you edit code
RUN npm ci --omit=dev            # <- forced to re-run (5 minutes)

# GOOD ORDER - npm install only runs when dependencies change
COPY package.json package-lock.json ./   # <- changes rarely
RUN npm ci --omit=dev                     # <- cached 99% of the time (skipped)
COPY . .                                  # <- changes often, but it's instant
```

With the bad order, here's what happens when you change one line of JavaScript: Docker sees `COPY . .` has new content -> cache invalidated -> `npm ci` runs again -> 5 minutes wasted. Every. Single. Time.

With the good order: Docker sees `COPY package.json package-lock.json ./` hasn't changed -> cache hit -> `npm ci` skipped -> Docker sees `COPY . .` has new content -> copies your code -> done in 2 seconds.

This is the difference between a 5-minute build and a 2-second build. Over a day of active development, that's hours saved.

Here's the full caching hierarchy from "changes least" to "changes most":

```
FROM node:20-bookworm-slim          # Almost never changes
RUN apt-get install ...             # Only changes when you need new system pac
COPY package.json package-lock.json # Only changes when you add/remove npm pack
RUN npm ci --omit=dev               # Only runs when the above changes
COPY . .                            # Changes every time you edit code (but it'
```

Every line is ordered so that the expensive, slow operations are cached and the cheap, fast operations happen last. This isn't an optimization - it's how you should always structure a Dockerfile.

`npm ci` vs `npm install` - Use the Right One

`npm install` reads `package.json` and resolves the latest compatible versions. It *updates* the lock file. It's non-deterministic - running it today and tomorrow might give you different versions.

`npm ci` reads `package-lock.json` and installs *exactly* what's listed there. It deletes `node_modules` first to ensure a clean state. It's deterministic - same lock file, same result, every time. It's also faster because it skips the dependency resolution step.

In a Dockerfile, always use `npm ci`. You want reproducible builds. If the build worked yesterday with these exact versions, it should work identically today.

`--omit=dev` - Ship What You Need, Nothing More

Your `package.json` has two dependency sections: `dependencies` (what your app needs to run) and `devDependencies` (what *you* need to develop - test runners, linters, formatters, hot-reload tools).

`npm ci --omit=dev` installs only `dependencies`. Your container doesn't need `jest`, `eslint`, `prettier`, `nodemon`, `typescript` (the compiler - your compiled JS is already in the image), or any other dev tool.

This matters for size and security:

```
# Typical node_modules size comparison
With devDependencies:    ~180 MB
Without (--omit=dev):    ~40 MB
```

That's 140 MB of packages that serve zero purpose in production. Every extra package is a potential vulnerability, a potential license issue, and wasted disk space on every node in your Swarm.

`NODE_ENV` - The One Variable That Changes Everything

This is one of the most misunderstood pieces of the Node.js ecosystem, and it causes real production problems. Developers build and test locally where `NODE_ENV` is usually unset (which means it defaults to `undefined`, and most libraries treat that as development mode). Then the container hits production where `NODE_ENV=production` is set - either by the Dockerfile, the compose file, or the hosting platform - and the app behaves completely differently.

Here's what actually changes when `NODE_ENV=production`:

Express changes its behavior significantly:

- View templates are cached instead of re-read from disk on every request. In development, Express re-reads your `.ejs` or `.pug` files on every request so you see changes immediately. In production, templates are compiled once and cached in memory. If your template has a bug that only shows up when cached (rare but real), you'll never see it locally.
- Error messages are sanitized. In development, Express sends full stack traces back to the browser. In production, it sends a generic "Internal Server Error". If your error handling code relies on parsing the error response, it breaks.
- CSS generated by `express.static()` and view engine responses get ETags and proper cache headers.

npm changes what it installs:

- `npm install` (without `--omit=dev`) automatically skips `devDependencies` when `NODE_ENV=production`. This is the trap. A developer writes a Dockerfile with `ENV NODE_ENV=production` set early (good practice for the final image), but puts `RUN npm ci` *after* it. Now `npm ci` runs with `NODE_ENV=production` active, which means it behaves like `npm ci --omit=dev` - silently skipping devDependencies. If your build step needs TypeScript, Webpack, Vite, Babel, or any other build tool that lives in `devDependencies`, the build fails or produces nothing.

This is the specific Dockerfile trap:

```
# THE TRAP - npm ci inherits NODE_ENV and skips devDependencies
FROM node:20-bookworm-slim
WORKDIR /app
ENV NODE_ENV=production          # <- Set too early
COPY package.json package-lock.json ./
RUN npm ci                        # <- Silently skips devDependencies
COPY . .
RUN npm run build                 # <- FAILS: "tsc: not found" or "vite: not fo
                                  #    because TypeScript/Vite were in devDeper
```

The fix is to control when `NODE_ENV` takes effect:

```
# CORRECT - Install everything, build, THEN set production mode
FROM node:20-bookworm-slim AS builder
WORKDIR /app
COPY package.json package-lock.json ./

# Install ALL dependencies (including devDependencies like TypeScript, Vite, et
RUN npm ci                        # <- No --omit=dev, no NODE_ENV=production ye

COPY . .
RUN npm run build                 # <- Works: tsc, vite, webpack are all availa

# ------------------------------------------
# Production stage - now we go lean
# ------------------------------------------
FROM node:20-bookworm-slim AS production
WORKDIR /app

ENV NODE_ENV=production           # <- Set here, in the final image

# Only production dependencies in the final image
COPY --from=builder /app/dist ./dist
COPY package.json package-lock.json ./
RUN npm ci --omit=dev             # <- Explicit --omit=dev is clearer than rely

ENTRYPOINT ["node", "./dist/server.js"]
```

The multi-stage build solves this naturally: Stage 1 has all your tools, builds your app, and gets thrown away. Stage 2 only has the compiled output and production dependencies.

Frameworks change their internal behavior:

React, Next.js, Vue, and others check `NODE_ENV` at build time and runtime:

FRAMEWORK	DEVELOPMENT MODE	PRODUCTION MODE
React	Includes extra warnings, slower reconciler with detailed error messages, React DevTools support	Minified, no warnings, faster reconciler, DevTools disabled
Next.js	Hot reload, detailed errors in browser, no optimization	Minified bundles, image optimization, ISR/SSG caching active
Vue	Detailed warnings, devtools hook, performance tracing	Warnings stripped, smaller bundle
Webpack/Vite	Source maps, no minification, HMR enabled	Tree shaking, minification, code splitting

The React one is particularly important. `React.createElement` runs different code paths based on `NODE_ENV`. The development version includes prop type checking, key warnings, and a slower but more debuggable rendering pipeline. The production version strips all of that. If you build your React app without `NODE_ENV=production`, your bundle is significantly larger and slower. Most bundlers handle this automatically when you run `npm run build`, but if you have a custom build setup, this is a real gotcha.

Logging libraries change their output:

- Winston, Pino, and Bunyan default to verbose/debug levels in development and info/warn in production.
- If you're debugging a production issue and wonder why you can't see debug logs, `NODE_ENV=production` is often why.

The real-world scenario that burns people:

A developer builds a TypeScript + Express API. Locally, `NODE_ENV` is unset. Everything works. They write a Dockerfile:

```
FROM node:20-bookworm-slim
WORKDIR /app
ENV NODE_ENV=production
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npx tsc                          # FAILS: tsc not installed
```

It fails because `npm ci` saw `NODE_ENV=production` and skipped TypeScript. The developer "fixes" it by removing `ENV NODE_ENV=production`. Now the image builds, but the production container runs without `NODE_ENV=production` - Express sends stack traces to users, React includes its development warnings, logging is verbose, and template caching is off.

The correct answer is always: **install everything in the build stage, set `NODE_ENV=production` in the final stage.** If you're using multi-stage builds (and you should be), this happens automatically because your build stage and production stage are different images.

One more trap - `--omit=dev` vs `NODE_ENV=production` during install:

These two mechanisms overlap but aren't identical:

MECHANISM	WHAT IT DOES	WHEN TO USE
`npm ci --omit=dev`	Explicitly skips devDependencies	In your Dockerfile's production stage. Clear, intentional, self-documenting
`NODE_ENV=production` + `npm ci`	npm detects the env and implicitly skips devDependencies	Accidental in Dockerfiles. Intentional on hosting platforms like Heroku, Railway, Render
`npm ci` (no flags, no NODE_ENV)	Installs everything	In your Dockerfile's build stage when you need devDependencies for compilation

Always prefer `--omit=dev` explicitly over relying on `NODE_ENV` to control npm's install behavior. It makes the Dockerfile self-documenting - anyone reading it can see exactly what's happening without needing to know that `NODE_ENV` affects npm's behavior.

Hosting platforms set `NODE_ENV=production` automatically. Heroku, Railway, Render, AWS Elastic Beanstalk, and most PaaS platforms set this by default. Docker Swarm and Kubernetes do not - you must set it yourself in your compose file or deployment manifest. If you forget, your production containers run in development mode with all the performance and security implications described above.

Build Targets - One Dockerfile, Multiple Images

Sometimes you want a development image with hot-reload and debugging tools, and a production image that's lean. Don't write two Dockerfiles. Use build targets:

```
# syntax=docker/dockerfile:1

# ================================================
# STAGE 1: Builder (shared by both targets)
# ================================================
FROM node:20-bookworm-slim AS builder
WORKDIR /app
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.json package-lock.json ./
RUN npm ci --omit=dev

# ================================================
# STAGE 2: Production target
# ================================================
FROM node:20-bookworm-slim AS production
WORKDIR /app
RUN ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime \
    && echo America/New_York > /etc/timezone
COPY --from=builder /app/node_modules /app/node_modules
COPY . .
EXPOSE 3000
ENTRYPOINT ["node", "--trace-warnings", "./server.js"]

# ================================================
# STAGE 3: Development target (extends production)
# ================================================
FROM production AS development
# Now install devDependencies ON TOP of the production image
RUN npm install
# Override the entrypoint with nodemon for hot-reload
CMD ["npx", "nodemon", "--inspect=0.0.0.0:9229", "server.js"]
```

Build whichever target you need:

```
# For production - lean, no dev tools
docker build --target production -t myapp:latest .

# For development - includes nodemon, debugger, devDeps
docker build --target development -t myapp:dev .
```

One Dockerfile. Two images. The production image is 245 MB with only what it needs. The development image extends it with devDependencies and debugging tools. Both are built from the same source, so there's no drift between environments.

Choosing Your Base Image - Stop Using `node:latest`

The base image you choose in your `FROM` instruction determines 80% of your final image size. Here's what's actually available and why it matters:

BASE IMAGE	SIZE	WHAT'S IN IT	RECOMMENDATION
`node:20`	~1 GB	Full Debian, every tool, man pages, docs	**Never use for production images**
`node:20-bookworm-slim`	~200 MB	Minimal Debian, Node.js, essentials only	**Use this - best balance**
`node:20-alpine`	~140 MB	Alpine Linux (musl libc), ultra-minimal	Fine if nothing uses glibc
`node:20-alpine` + native modules	Varies	Alpine + compiled native code	**Risky - musl breaks things**

The full `node:20` image includes a complete Debian installation with hundreds of packages you'll never need - compilers, editors, documentation, shells, networking tools. It's designed for convenience, not for production.

`node:20-bookworm-slim` strips that down to the bare minimum needed to run Node.js. It still uses glibc (the standard C library), so all native npm packages work correctly. This is what I use for every production image.

`node:20-alpine` is even smaller because Alpine Linux uses musl instead of glibc. The problem is that musl handles some system calls differently, and native npm packages compiled against glibc sometimes fail silently or crash on musl. I've hit this with `bcrypt`, `sharp`, and MongoDB's native driver. The 60 MB you save isn't worth the debugging time.

Always pin a specific version. Not `node:latest` (which could be anything), not `node:20` (which gets minor updates), but `node:20-bookworm-slim` or even `node:20.11-bookworm-slim` if you want full reproducibility. When you're debugging a production issue at 2 AM, you want to know *exactly* which Node.js version is running. `latest` is not an answer.

Image Tags: A Complete Tagging Strategy

Image tags are how you identify, version, and track your Docker images. Most developers never think about tagging - they build `myapp:latest`, push `myapp:latest`, deploy `myapp:latest`, and then have absolutely no idea which version of their code is running in production. This section explains every tagging strategy, when to use each one, and how to combine them into a system that makes deployments traceable and rollbacks instant.

How Docker Tags Actually Work

A tag is just a human-readable label that points to a specific image digest (a SHA256 hash). When you do `docker build -t myapp:latest .`, you're telling Docker: "build this image, and label the result as `myapp:latest`." The tag is a pointer, not the image itself.

Multiple tags can point to the same image. When you do:

```
docker build -t myapp:latest -t myapp:1.4.72 -t myapp:a3f8c2d .
```

You've created one image with three labels pointing to it. They're not three copies - they're three names for the same SHA256 digest. If you push all three, the registry stores one image and three tag references.

Tags are also **mutable**. If you build a new image tomorrow and tag it `myapp:latest`, the tag moves. It now points to tomorrow's build. Yesterday's build still exists (it has a digest), but the label `latest` no longer points to it. This is exactly why `latest` is dangerous in production - the tag can shift under you without you doing anything.

The `:latest` Tag - What It Is and When to Use It

`latest` is Docker's default tag. If you don't specify a tag, Docker uses `latest`:

```
docker build -t myapp .          # Actually myapp:latest
docker pull nginx                 # Actually nginx:latest
docker push myapp                 # Actually myapp:latest
```

What `latest` does NOT mean: It does not mean "the most recently built image." It does not mean "the newest version." It does not automatically update when you push a new image. It's just a tag - the word "latest" is purely a naming convention. Docker assigns no special behavior to it.

What goes wrong with `latest` in production:

- You push `myapp:latest` on Monday. It works.
- Your teammate pushes `myapp:latest` on Wednesday with a broken migration.
- A Swarm node restarts Thursday morning, pulls `myapp:latest`, and gets Wednesday's broken build.
- Your production is now running different code than it was on Tuesday. Nobody deployed anything. The tag just points somewhere different now.
- You try to roll back. Roll back to what? `myapp:latest` is the broken version. There's no previous tag to reference.

When `:latest` IS fine:

- Local development - `docker compose up` pulling `myapp:latest` is perfectly fine when you're the only one building and it's your own machine
- As one of multiple tags - tagging an image as both `myapp:latest` AND `myapp:1.4.72` gives you a convenience pointer plus a stable reference
- For base images in development - using `FROM node:20-bookworm-slim` in development (where `bookworm-slim` is effectively a "latest-ish" tag for that Debian release) is acceptable, though you should pin more specifically for production Dockerfiles

When `:latest` is NOT fine:

- Production deployments - never deploy `:latest` to production
- Any compose file used in Swarm - always use a specific version or SHA
- CI/CD pipelines - always tag with something unique and traceable

- Any environment where someone other than you might push to the same tag

`Semantic Version Tags (:1.4.72)`

This is the most readable tagging strategy. Your image version corresponds to a release version that means something to your team:

```
docker build -t myapp:1.4.72 .
docker push myapp:1.4.72
```

How to automate it: Store your version in a file that your build pipeline reads:

```
# buildVersion.txt
1.4.72
```

```
# Build script
BUILD_VERSION=$(cat ./globals/_versioning/buildVersion.txt)
docker build -t myapp:${BUILD_VERSION} .
docker push myapp:${BUILD_VERSION}
```

Advantages:

- Human-readable - `myapp:1.4.72` immediately tells you "this is version 1.4.72"
- Rollback is obvious - `docker service update --image myapp:1.4.71` rolls back to the previous release
- Compatible with semantic versioning - major.minor.patch maps to breaking.feature.fix
- Easy to communicate - "we're running 1.4.72 in production" is something everyone on the team understands

Disadvantages:

- Requires discipline - someone needs to bump the version number on each release
- Version bumps are a manual step (unless fully automated in CI)
- Doesn't tell you *which commit* the image was built from (pair with SHA tags for that)

Use when: You have a release process, even a simple one. You bump a version number when you deploy. This is the right strategy for most teams.

`Git SHA Tags (:a3f8c2d)`

Tag the image with the git commit hash it was built from:

```
SHORT_SHA=$(git rev-parse --short HEAD)
docker build -t myapp:${SHORT_SHA} .
docker push myapp:${SHORT_SHA}
```

Advantages:

- Fully automatic - no manual version bumping required
- Every commit produces a unique tag - no collisions
- Direct traceability - `docker inspect` shows `myapp:a3f8c2d`, you run `git show a3f8c2d` and see exactly what code is in that image
- Perfect for debugging - "what's running in production?" -> `a3f8c2d` -> `git log a3f8c2d` -> you can see the commit message, the diff, the author, the timestamp
- Works with any branching strategy - you don't need release branches or version files

Disadvantages:

- Not human-readable - `myapp:a3f8c2d` doesn't tell you anything unless you look it up in git
- Rollback requires knowing the previous SHA - `docker service update --image myapp:???` requires looking up which SHA was running before
- Can't tell at a glance whether one tag is "newer" than another - `a3f8c2d` and `b7e1f4a` have no inherent ordering

Use when: You deploy frequently (multiple times per day), you don't have a formal release process, or you want zero-friction tagging that requires no manual steps.

`Full SHA Digest Tags (:sha-a3f8c2d9e4b1...)`

The full 40-character git SHA, or even the Docker image digest itself:

```
FULL_SHA=$(git rev-parse HEAD)
docker build -t myapp:sha-${FULL_SHA} .

# Or reference the image by its actual Docker digest (immutable)
docker pull myapp@sha256:abc123def456...
```

The key difference: Docker digests (the `sha256:` format) are *immutable*. Nobody can push a new image and overwrite the digest. If you deploy by digest, you are guaranteed to run the exact same bytes forever. Tags can move. Digests cannot.

Use when: You need absolute, cryptographic certainty that the image you tested is the image that runs. Regulated industries, financial systems, security-critical deployments. For most teams, short SHA tags are sufficient.

Branch Tags (`:main`, `:develop`, `:feature-login`)

Tag images with the git branch they were built from:

```
BRANCH=$(git rev-parse --abbrev-ref HEAD)
docker build -t myapp:${BRANCH} .
```

Advantages:

- Immediately tells you which branch the image came from
- Useful for preview environments - deploy `myapp:feature-login` to a preview URL for testing
- CI/CD can automatically build and tag branches

Disadvantages:

- Mutable - every push to `main` overwrites `myapp:main`. Same problem as `:latest` for production
- Branch names can contain characters that aren't valid in Docker tags (slashes need to be replaced)
- Not suitable for production - for the same reasons `:latest` isn't

Use when: Preview environments, staging environments where you want to test a specific branch. Never for production.

Date/Timestamp Tags (`:2026-02-22`, `:20260222-143052`)

Tag images with the build date:

```
docker build -t myapp:$(date +%Y%m%d-%H%M%S) .
# Result: myapp:20260222-143052
```

Advantages:

- Immediately tells you *when* the image was built
- Natural ordering - you can sort by tag name to find the newest
- No manual versioning needed

Disadvantages:

- Doesn't tell you *what code* is in the image - you know when, but not which commit
- Multiple builds on the same day need second-level precision
- Not standard - most tools and teams don't expect date-based tags

Use when: Rarely as the primary strategy. Sometimes useful as a secondary tag alongside version or SHA tags for quick visual reference.

The Combined Strategy (What You Should Actually Do)

The best approach uses multiple tags on the same image. You're not choosing one - you're applying several labels to the same artifact:

```
BUILD_VERSION=$(cat ./globals/_versioning/buildVersion.txt)
SHORT_SHA=$(git rev-parse --short HEAD)

# Build once, tag multiple times
docker build \
  -t myapp:latest \
  -t myapp:${BUILD_VERSION} \
  -t myapp:${SHORT_SHA} \
  .

# Push all tags (same image, multiple labels)
docker push myapp:latest
docker push myapp:${BUILD_VERSION}
docker push myapp:${SHORT_SHA}
```

Now your image has three tags:

TAG	PURPOSE	EXAMPLE
`myapp:latest`	Convenience pointer for local dev	`myapp:latest`
`myapp:1.4.72`	Human-readable version for deployments and rollback	`myapp:1.4.72`
`myapp:a3f8c2d`	Git traceability for debugging	`myapp:a3f8c2d`

Deploy production using the version tag: `docker service update --image myapp:1.4.72`

Debug an issue by checking the SHA: `docker inspect` -> `myapp:a3f8c2d` -> `git show a3f8c2d`

Roll back using the previous version: `docker service update --image myapp:1.4.71`

Local development uses latest: `docker compose up` pulls `myapp:latest` - always the most recent build, which is exactly what you want locally.

This gives you readability, traceability, and convenience - all pointing to the same image bytes.

Tagging in Your Compose File

Your compose file should reference the version tag, not `latest`, for any environment that isn't pure local development:

```
services:
  nodeserver:
    # Build is for local dev - variable substitution
    build:
      context: ./nodeserver
      args:
        - BUILD_VERSION=${BUILD_VERSION}
        - GIT_COMMIT=${LONG_COMMIT}

    # Image uses version from .env or CI pipeline
    image: "${REGISTRY}/nodeserver:${BUILD_VERSION}"
```

With your `.env` file:

```
REGISTRY=yourregistry
BUILD_VERSION=1.4.72
LONG_COMMIT=a3f8c2d9e4b1f2c3d4e5f6a7b8c9d0e1f2a3b4c5
```

This means `docker compose up` locally uses the version you're working on, and `docker stack deploy` in production uses the exact version tag from the compose file. Change `BUILD_VERSION` in your deploy script, and the whole pipeline knows which version to push and deploy.

Embedding Version Information Inside the Image

Tags are external labels - they can be changed, moved, or deleted from the registry. For absolute traceability, also embed the version information inside the image itself using `ARG` and `LABEL`:

```
ARG BUILD_VERSION=dev
ARG GIT_COMMIT=unknown

LABEL version="${BUILD_VERSION}"
LABEL git.commit="${GIT_COMMIT}"
LABEL build.date="$(date -u +'%Y-%m-%dT%H:%M:%SZ')"
```

Now `docker inspect` shows the version, commit, and build date regardless of what the image is tagged as. Even if someone retags the image or pulls it by digest, the metadata is baked in.

You can also write this to a file your application exposes:

```
RUN echo "{\"version\":\"${BUILD_VERSION}\",\"commit\":\"${GIT_COMMIT}\"}" > /a
```

Then add a `/version` endpoint to your app:

```
app.get('/version', (req, res) => {
  const version = require('./version.json');
  res.json(version);
});
```

Hit `/version` in production and you know exactly what's running - no SSH, no `docker inspect`, no guessing. This is the difference between "I think we're running 1.4.72" and "I can prove we're running 1.4.72, built from commit a3f8c2d."

The Complete Version Tracking Pipeline

The tagging strategies and embedded labels above are the building blocks. This section connects them into a complete system where every build is automatically versioned, every deployment is traceable, and every production bug can be traced back to the exact git commit that introduced it.

Step 1: The version file.

Store your version in a simple text file in your project. This file is the single source of truth for your application's version:

```
globals/_versioning/buildVersion.txt
1.4.72
```

Semantic versioning (semver) works well here: `major.minor.patch`. Major is for breaking changes. Minor is for features. Patch is for bug fixes. But the critical thing isn't the format - it's that every build increments the number. No two builds should ever have the same version. Some teams use `1.4.72`, some use `1.4.72.1189` with a fourth build number, some use `2025.02.22.3` as a date-based version. Pick a convention and never reuse a number.

Step 2: The build script that ties everything together.

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

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

# Read version from the version file
BUILD_VERSION=$(cat ./globals/_versioning/buildVersion.txt)
LONG_COMMIT=$(git rev-parse HEAD)
SHORT_COMMIT=$(git rev-parse --short HEAD)
BUILD_DATE=$(date -u +'%Y-%m-%dT%H:%M:%SZ')
BRANCH=$(git rev-parse --abbrev-ref HEAD)

# Export for docker compose interpolation
export BUILD_VERSION
export LONG_COMMIT

# Build once with all build args passed in
docker compose build \
  --build-arg BUILD_VERSION=$BUILD_VERSION \
  --build-arg GIT_COMMIT=$LONG_COMMIT \
  --build-arg BUILD_DATE=$BUILD_DATE

# Tag with both version and SHA (same image, two labels)
docker tag ${REGISTRY}/${SERVICE}:${BUILD_VERSION} ${REGISTRY}/${SERVICE}:${SHO
docker tag ${REGISTRY}/${SERVICE}:${BUILD_VERSION} ${REGISTRY}/${SERVICE}:lates

# Push all three tags
docker push ${REGISTRY}/${SERVICE}:${BUILD_VERSION}
docker push ${REGISTRY}/${SERVICE}:${SHORT_COMMIT}
docker push ${REGISTRY}/${SERVICE}:latest

echo "Built and pushed ${SERVICE}:${BUILD_VERSION} (commit ${SHORT_COMMIT})"
```

Step 3: The Dockerfile that bakes version info into the image.

The build args from the script become labels and a version file inside the image:

```
# In your production stage:
ARG BUILD_VERSION=dev
ARG GIT_COMMIT=unknown
ARG BUILD_DATE=unknown

LABEL version="${BUILD_VERSION}"
LABEL git.commit="${GIT_COMMIT}"
LABEL build.date="${BUILD_DATE}"

# Write a version file your app can read at runtime
RUN printf '{"version":"%s","commit":"%s","date":"%s"}' \
    "${BUILD_VERSION}" "${GIT_COMMIT}" "${BUILD_DATE}" > /app/version.json
```

Now your image carries its own identity. Even if someone retags it, even if the registry loses metadata, `docker inspect` and the `/version` endpoint will always tell you exactly what's inside.

Step 4: The application exposes version info to monitoring tools.

This is where most teams stop - they tag images and call it done. But the real power comes when your running application actively reports its version to your monitoring stack:

```
const fs = require('fs');

// Load version info once at startup
let versionInfo = { version: 'unknown', commit: 'unknown', date: 'unknown' };
try {
  versionInfo = JSON.parse(fs.readFileSync('./version.json', 'utf8'));
} catch (err) {
  // Running locally without a build - that's fine
}

// Expose via endpoint (for humans and health checks)
app.get('/version', (req, res) => {
  res.json(versionInfo);
});

// Include in every log line (for log aggregation tools)
const pino = require('pino');
const logger = pino({
  level: process.env.LOG_LEVEL || 'info',
  base: {
    service: 'nodeserver',
    version: versionInfo.version,
    commit: versionInfo.commit,
  },
});

// Now every log entry includes:
// {"level":30,"time":1705315200,"service":"nodeserver","version":"1.4.72","com
```

When every log line includes the version and commit hash, your log aggregator (ELK, Loki, CloudWatch, Datadog) can filter and group by version. You can answer questions like:

- "Show me all errors from version 1.4.72" - to confirm a bug was introduced in a specific release
- "Compare error rates between 1.4.71 and 1.4.72" - to see if a deployment made things worse
- "When did version 1.4.72 first appear in logs?" - to know exactly when the deploy completed across all replicas

Step 5: Expose version as monitoring metrics and tags.

For tools like Datadog, Prometheus, or Grafana, expose the version as a metric label or tag:

```
// Prometheus-style metrics (using prom-client)
const client = require('prom-client');

// Create a gauge that always reports 1, with version labels
const buildInfo = new client.Gauge({
  name: 'app_build_info',
  help: 'Application build information',
  labelNames: ['version', 'commit', 'node_version'],
});
buildInfo.set(
  {
    version: versionInfo.version,
    commit: versionInfo.commit,
    node_version: process.version,
  },
  1
);

// Include version in custom metrics
const httpRequestDuration = new client.Histogram({
  name: 'http_request_duration_seconds',
  help: 'HTTP request duration',
  labelNames: ['method', 'route', 'status', 'version'],
});

// In your request handler:
const end = httpRequestDuration.startTimer();
// ... handle request ...
end({ method: req.method, route: req.route.path, status: res.statusCode, versio
```

In Grafana, you can now build dashboards that show response times per version. When 1.4.72 deploys and the p99 latency jumps from 50ms to 200ms, you see it immediately on the graph - and you know exactly which commit to blame.

For Datadog specifically, set the version as a service tag:

```
// If using dd-trace (Datadog APM)
const tracer = require('dd-trace').init({
  service: 'nodeserver',
  version: versionInfo.version,  // Shows up in Datadog APM as the service vers
  env: process.env.NODE_ENV,
});
```

Datadog's deployment tracking feature uses this version tag to draw vertical lines on your dashboards at deployment boundaries. You can see exactly when each version went live and how it affected error rates, latency, and throughput.

Step 6: Increment the version after each build.

After a successful build and push, bump the patch number:

```
# At the end of your build script, after pushing:
CURRENT=$(cat ./globals/_versioning/buildVersion.txt)
MAJOR=$(echo $CURRENT | cut -d. -f1)
MINOR=$(echo $CURRENT | cut -d. -f2)
PATCH=$(echo $CURRENT | cut -d. -f3)
NEW_PATCH=$((PATCH + 1))
echo "${MAJOR}.${MINOR}.${NEW_PATCH}" > ./globals/_versioning/buildVersion.txt
git add ./globals/_versioning/buildVersion.txt
git commit -m "Bump version to ${MAJOR}.${MINOR}.${NEW_PATCH}"
```

Some teams auto-increment patch for every build and manually bump minor/major for features and breaking changes. Some use their CI/CD system to handle this entirely. The mechanism doesn't matter - what matters is that no two builds share a version number.

Why This Matters at 2 AM

Here's the scenario. It's 2 AM. Your monitoring alert fires. Error rate on the `/api/orders` endpoint spiked to 40%.

Without version tracking:

You SSH into the Swarm manager. Run `docker service inspect mystack_nodeserver`. The image is `myapp:latest`. What version is this? When was it deployed? What changed? You have no idea. You check git log. There were 6

commits today by 3 different developers. Which one is running? You don't know. You can't even roll back confidently because you don't know what "the previous working version" was.

With version tracking:

Your Datadog dashboard shows the error rate spike started at 11:47 PM, exactly when version `1.4.72` was deployed. You click on the deployment marker. The version label tells you commit `a3f8c2d`. You run `git show a3f8c2d` and see the diff - someone changed the order validation logic. You run `docker service update --image myapp:1.4.71 mystack_nodeserver`. Swarm rolls back in 30 seconds. Error rate drops to zero. You go back to sleep. Tomorrow you look at the diff and fix the bug.

That's the difference. Not "we think it might be related to yesterday's deploy." Instead: "version 1.4.72, commit a3f8c2d, deployed at 11:47 PM, introduced a bug in `orderValidator.js` line 47, rolled back to 1.4.71 at 2:03 AM, resolved."

Putting It All Together - The Complete Production Dockerfile

Here's the Dockerfile I use as a starting template for every new Node.js service. Every line has been refined through years of production use:

```
# syntax=docker/dockerfile:1

# ================================================
# STAGE 1: Build dependencies
# This stage compiles native npm packages.
# Nothing from here ships in the final image.
# ================================================
FROM node:20-bookworm-slim AS builder

WORKDIR /app

# Build tools for native npm packages (bcrypt, sharp, etc.)
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    python3 \
    make \
    g++ \
    && apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Package files first -> layer caching optimization
COPY package.json package-lock.json ./

# Production deps only, exact versions from lock file
RUN npm ci --omit=dev

# ================================================
# STAGE 2: Production image
# Clean slate - only Node.js, your code,
# and the compiled packages from Stage 1.
# ================================================
FROM node:20-bookworm-slim AS production

# Timezone for correct log timestamps
RUN ln -snf /usr/share/zoneinfo/America/New_York /etc/localtime \
    && echo America/New_York > /etc/timezone

WORKDIR /app

# Production mode - Express caches views, npm behaves differently,
# logging defaults to info level. See the NODE_ENV section above.
ENV NODE_ENV=production

# Compiled node_modules from the builder stage
COPY --from=builder /app/node_modules /app/node_modules
```

```
# Application source code (last because it changes most often)
COPY . .

# Document the port
EXPOSE 3000

# Run the app - ENTRYPOINT can't be accidentally overridden
ENTRYPOINT ["node", "--trace-warnings", "./server.js"]
```

If you take one thing from this entire section: **the Dockerfile is not a script you write once and forget. It's architecture.** The order of instructions, the choice of base image, the separation of build tools from runtime - these decisions determine whether your image is 40 MB or 1 GB, whether your builds take 2 seconds or 5 minutes, and whether an attacker who gets into your container finds a bare runtime or a fully equipped workshop.

DEPLOY IMAGES, NOT CODE: WHY "GIT PULL ON PRODUCTION" IS AN ANTI-PATTERN

This is one of those topics where two completely different approaches exist, and one of them is clearly wrong - but the wrong one is surprisingly common because it *feels* simpler.

I'm talking about how your code gets from your machine to production. There are two schools of thought:

The wrong way: You SSH into your production server. You run `git pull`. You run `npm install`. Maybe you run a build step. You restart the process. Congratulations - you've just compiled code on a production machine. Some teams automate this with a CI/CD pipeline that does the same thing: clone the repo on the server, install dependencies, build, restart. It's the same problem with a fancier wrapper.

The right way: You build a Docker image on your development machine (or in a CI pipeline). You test that exact image locally. You push that exact image to a registry. Your production server pulls that exact image and runs it. No git. No npm install. No build step. No compiler. The image that ran on your laptop is byte-for-byte identical to the image running in production.

These are not two equally valid approaches. One of them is how Docker was designed to work. The other is using Docker as a glorified process manager while throwing away its primary advantage.

What "Git Pull on Production" Actually Does to You

When you build code on a production server, here's everything that can go wrong - and eventually will:

Your production server needs build tools. To run `npm install` with native packages, your server needs `python3`, `make`, `g++`, and whatever else your dependency tree requires for compilation. These are compilers. On a production server. They serve zero runtime purpose but expand your attack surface dramatically. This is exactly what multi-stage builds were invented to eliminate - and you've just put the tools back.

`npm install` is non-deterministic across environments. Even with a lock file, `npm install` can behave differently on different machines. Different OS versions, different system libraries, different versions of `node-gyp` - all of these can produce different compiled binaries. The `bcrypt` binary compiled on your Ubuntu 22.04 dev machine is not guaranteed to be identical to the one compiled on your Ubuntu 24.04 production server. I've seen services crash in production because a native module compiled differently on the production OS than it did in development. With a pre-built image, you compile once and run the result everywhere.

Your deploy depends on npm being available. Your production deployment now has an external dependency on the npm registry. If npm is down - and it does go down - you can't deploy. If npm rate-limits you, you can't deploy. If a package author yanks a version (remember `left-pad`?), your `npm install` fails and you can't deploy. With a pre-built image sitting in your own registry, your deploy depends on nothing external. The image is self-contained.

Your deploy depends on git being available. Same problem. If GitHub has an outage, you can't deploy. With images in your own registry (Docker Hub, ECR, a self-hosted registry), you have no dependency on your source control provider during deployment.

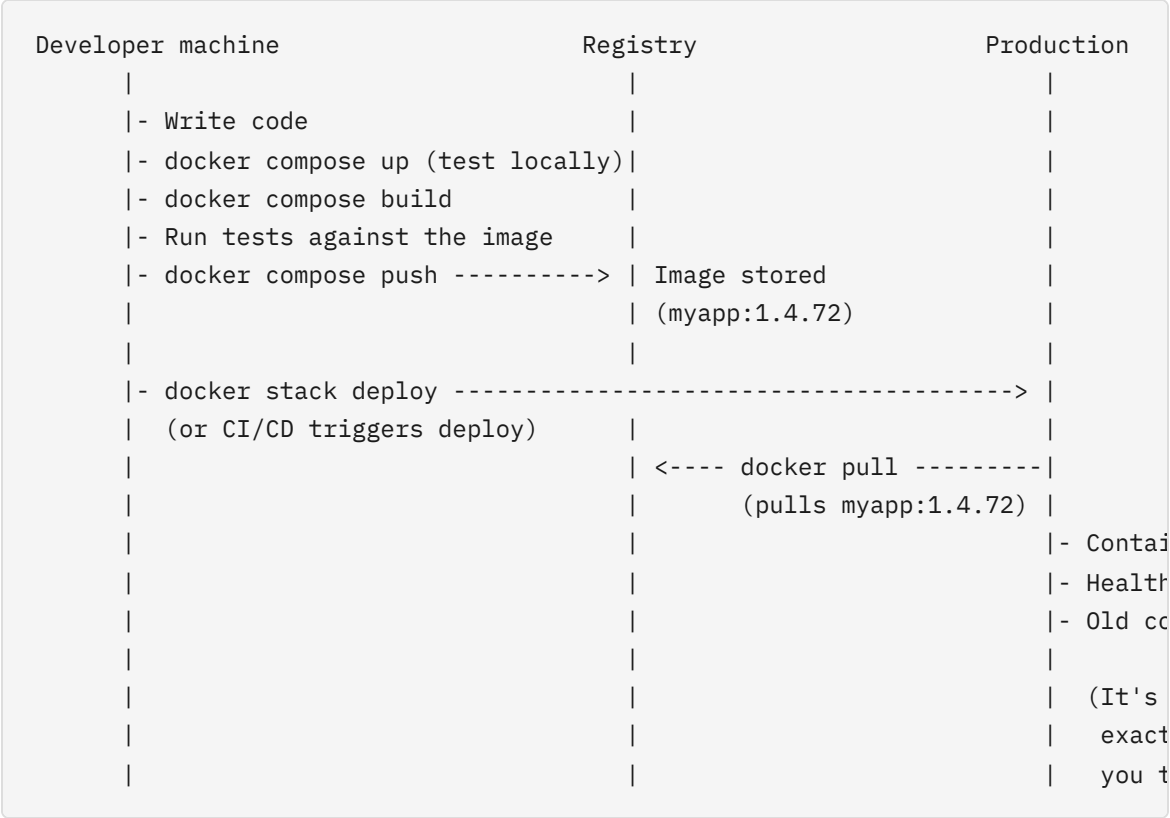
You can't roll back instantly. When production breaks after a `git pull` deploy, your rollback is: SSH in, run `git checkout` to a previous commit, run `npm install` again (because `node_modules` might need different dependencies for the old version), restart. That's minutes of downtime while you fumble with commands under pressure. With Docker images, your rollback is `docker service update --image myapp:1.4.71 mystack_nodeserver` - Swarm pulls the previous image (which is probably already cached on the node) and rolls back in seconds. You can even set Swarm to do this automatically when a health check fails.

Every deploy is a fresh build, and builds can fail. Even if your code hasn't changed, a `git pull && npm install` can fail because a transitive dependency published a broken version, because a system library got updated on the server, because disk space ran low during compilation, because the build step exceeded available memory. Every deploy is a gamble that the build environment is still in the state you expect. With a pre-built image, the "build" step on the production server is `docker pull` - downloading a file. It either downloads or it doesn't. There's no compilation to fail.

You have no guarantee that what you tested is what's running. This is the fundamental problem. You tested your code on your machine. Then you deploy by building it again on a different machine. You're testing one artifact and running a different one. Every time you rebuild, you create a new artifact. Multi-stage Docker builds produce a single artifact - the image - and that image is what you test locally, what you push to staging, and what you deploy to production. Same SHA256 hash. Same bytes. Same behavior.

The Image-Based Workflow

Here's how it should work, step by step:



The build happens once - on your machine or in CI. The result is a versioned, immutable image. That image gets tested. If it passes, it gets pushed. Production pulls it and runs it. No compilation. No dependency resolution. No build tools. No git. Just your tested, verified artifact running in a clean container.

The Deployment Script

This is the script from our production guide. It builds, tags, pushes, and deploys in one pipeline:

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

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

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

# Export for docker-compose variable substitution
export BUILD_VERSION
export LONG_COMMIT

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

# Build the image (this is the ONLY time compilation happens)
docker compose -f $COMPOSE_FILE build

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

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

# Deploy to Swarm - production just pulls and runs
echo "Deploying to swarm..."
docker stack deploy -c $COMPOSE_FILE mystack

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

The production server never sees your source code. It never runs `npm install`. It never compiles anything. It receives a finished, tested image and runs it. That's the entire deployment.

"But I Use CI/CD - Isn't That the Same?"

Not necessarily. A lot of CI/CD pipelines do something like this:

```
# GitHub Actions / GitLab CI - the WRONG pattern
deploy:
  steps:
    - ssh production-server
    - cd /app && git pull origin main
    - npm ci --production
    - pm2 restart all
```

This is `git pull on production` with extra steps. You've added CI/CD tooling but you're still building on the production machine. You still have all the problems listed above.

The correct CI/CD pipeline builds the image in the CI environment, pushes it to a registry, and then tells the production server to pull and run it:

```
# The RIGHT pattern
build:
  steps:
    - docker build -t myapp:${VERSION} .
    - docker push myapp:${VERSION}

deploy:
  steps:
    - ssh production-server
    - docker service update --image myapp:${VERSION} mystack_nodeserver
```

The CI runner has the build tools. The CI runner has npm access. The CI runner compiles the native modules. The production server has none of that. It just runs images.

Why This Matters Even More in Swarm

In a single-server deployment, building on production is bad practice. In Swarm, it's not even possible - and that's by design.

Swarm only runs pre-built images. There is no `docker build` in Swarm. When you do `docker stack deploy`, Swarm reads the `image:` field from your compose file, pulls that image from the registry, and starts containers from it. The `build:` field is ignored completely. This isn't a limitation - it's an architectural decision.

Why? Because Swarm runs containers across multiple nodes. If you have 6 replicas across 3 nodes, each node needs the image. If you built the image on one node, the other two don't have it. Pulling from a registry ensures every node gets the same image. It also means scaling is instant - when you scale from 6 to 12 replicas, Swarm pulls the image (usually already cached) to new nodes and starts containers immediately. There's no "wait for the build to finish" step.

The entire Docker ecosystem - registries, image layers, content-addressable storage, multi-stage builds - was designed around this one idea: **build once, run anywhere**. When you build on production, you're running Docker without using the feature that makes Docker worth using.

The Comparison at a Glance

ASPECT	GIT PULL ON PRODUCTION	PRE-BUILT IMAGES
Build tools on prod server	Yes (compilers, python, make)	No
Dependency on npm registry at deploy time	Yes - deploy fails if npm is down	No - image is self-contained
Dependency on git at deploy time	Yes - deploy fails if GitHub is down	No - image is in your registry
What you tested = what's running?	No - rebuilt on different machine	Yes - same image, same SHA256
Rollback speed	Minutes (checkout, install, restart)	Seconds (pull cached previous image)
Deploy can fail due to build issues	Yes - transitive deps, disk, memory	No - it's just a download
Works with Swarm	No - Swarm only runs images	Yes - designed for this
Attack surface on prod server	Large (compilers, build tools)	Minimal (runtime only)
Reproducibility	No - non-deterministic across machines	Yes - byte-for-byte identical everywhere

There is no scenario where building on production is the better choice. The image-based workflow is faster to deploy, faster to roll back, more secure, more reliable, and the only approach that works with orchestrators like Swarm. If you're currently doing `git pull` deploys, migrating to image-based deploys is the single highest-value change you can make to your deployment pipeline.

"But GitHub Actions Builds My Image For Me"

There's a third deployment pattern that's become extremely common, and it's subtler than the "git pull on production" problem. It goes like this:

You develop locally. You run `node server.js` directly on your machine, or maybe you do `docker compose up` with a basic development setup. You push to GitHub. A GitHub Actions workflow (or GitLab CI, or any CI service) builds your Docker image, pushes it to a registry, and deploys it to production. You never built the production image locally. You never ran it. You never tested it outside of CI.

This is better than building on production - the image is pre-built, immutable, and versioned. But it introduces a blind spot that catches developers constantly: **you're deploying an image you've never actually run.**

The GitHub Actions runner builds on Ubuntu. It might have different system libraries than your Mac. The multi-stage build might produce slightly different results. The `npm ci` in the CI environment might resolve a dependency differently because it has a different architecture or Node version. The image works - it builds, it passes the unit tests CI runs against it - but you've never pulled it, started it, and sent a request to it.

This is how bugs slip through that seem impossible. "It passes all the tests, it builds fine, but it crashes in production." You look at the logs and there's a native module that compiled for linux/amd64 in CI but your local testing was on linux/arm64, or a dependency that expected a system library that exists in the CI runner but not in the slim production base image. You'd have caught this in 10 seconds if you'd pulled the production image and run it locally.

The fix: After your CI builds the image, pull it and run it locally before promoting it to production. Or better yet, build the image locally first, test it, then push it. CI can run additional integration tests against the same image, but the developer should have already verified it works. The point of image-based deployment is that the same image runs everywhere - but that only helps you if "everywhere" includes your development machine.

You Need a Staging Environment (And You Probably Don't Have One)

Here's the real problem, and it's bigger than which CI service builds your image.

Most developers work like this: they run their Node app (or Python app, or whatever) locally with `docker compose up`. Their compose file has their app and a database. Maybe Redis. They hit `http://localhost:3000` in their browser, everything works,

they ship it.

In production, the request path looks nothing like this:

```
Local development:
  Browser -> http://localhost:3000 -> Node app

Production:
  Browser -> DNS -> Load Balancer -> WAF -> NGINX (SSL/reverse proxy)
    -> Docker overlay network -> Node app (one of 6 replicas)
```

Count the layers that exist in production that don't exist in your local setup: DNS resolution, load balancer, Web Application Firewall (WAF), NGINX as a reverse proxy handling SSL termination, path rewriting, header injection, rate limiting, and the Docker overlay network routing to one of multiple replicas behind Swarm's internal load balancer.

Every single one of those layers can break your application in ways you'll never see locally. And these aren't exotic edge cases - they're the most common production failures I see:

NGINX reverse proxy breaks your WebSocket connections. Your app uses WebSockets. Locally, the browser connects directly to your Node server on port 3000, the WebSocket upgrade succeeds, everything works. In production, the request goes through NGINX first. NGINX doesn't proxy WebSocket upgrades by default - you need explicit configuration:

```
location /ws {
    proxy_pass http://nodeserver:3000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
}
```

Without this, your WebSocket connections silently fail in production. You never caught it because you never had NGINX in front of your app during development.

The WAF blocks legitimate requests. Your API accepts JSON payloads. Some of those payloads contain strings that look suspicious to a WAF - SQL keywords, HTML tags, base64-encoded data, long strings. The WAF blocks the request with a 403 before it ever reaches your application. Your app didn't even get the chance to process it. Locally, there's no WAF, so these requests work fine. In production, certain API calls randomly fail and your application logs show nothing because the request never arrived.

NGINX rewrites headers your app depends on. Your application reads `req.headers.host` or `req.ip` to determine the request origin. Locally, these values are `localhost:3000` and `127.0.0.1`. Behind NGINX, the `Host` header might be the external domain, and the client IP is NGINX's IP - not the user's IP - unless NGINX is configured to pass `X-Forwarded-For` and `X-Real-IP`, and your app is configured to trust the proxy.

```
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header Host $host;
```

```
// Express needs to know it's behind a proxy
app.set('trust proxy', 1);

// Now req.ip gives the real client IP, not NGINX's IP
```

But here's the gotcha nobody warns you about: `X-Forwarded-For` is not a single IP address. It's a comma-separated chain of every proxy the request passed through. In production, your request might traverse a CDN, a cloud load balancer, a WAF, and your NGINX - each one appending its IP to the chain. By the time the request reaches your app, the header looks like this:

```
X-Forwarded-For: 203.0.113.50, 198.51.100.178, 10.0.0.2, 172.18.0.5
                 ^             ^               ^          ^
                 Real client   CDN edge        AWS ALB    NGINX
```

If your code does `req.headers['x-forwarded-for']` and treats the result as a single IP, you'll get the entire string `"203.0.113.50, 198.51.100.178, 10.0.0.2, 172.18.0.5"` - not an IP address. Your IP-based rate limiter, your geolocation lookup, your audit log - all broken because they're trying to parse a comma-separated list as a single address.

Express's `trust proxy` setting handles this parsing *if configured correctly*, but the number you pass matters:

```
// trust proxy = 1: Trust one proxy hop (your NGINX)
// req.ip returns the IP one hop before NGINX
// In the chain above: 10.0.0.2 (AWS ALB) - still wrong!
app.set('trust proxy', 1);

// trust proxy = 2: Trust two proxy hops (NGINX + ALB)
// req.ip returns: 198.51.100.178 (CDN) - still might be wrong!
app.set('trust proxy', 2);

// trust proxy = 'loopback,linklocal,uniquelocal': Trust private IPs
// This skips all RFC 1918 addresses and returns the first public IP
// Usually the correct approach for most setups
app.set('trust proxy', 'loopback,linklocal,uniquelocal');

// trust proxy = true: Trust ALL proxies - DANGEROUS
// An attacker can spoof X-Forwarded-For and inject any IP
// Never use this in production
app.set('trust proxy', true);  // DON'T DO THIS
```

The safe default for most deployments (Swarm behind NGINX, or any cloud setup with a load balancer):

```
// Trust private/internal network IPs as proxies
// This works because your proxies (NGINX, ALB, etc.) have internal IPs
// The first non-private IP in the chain is your real client
app.set('trust proxy', 'loopback,linklocal,uniquelocal');

// Now req.ip is the real client IP
// And req.ips is the full array of proxy chain IPs
console.log(req.ip);   // "203.0.113.50" (the real client)
console.log(req.ips);  // ["203.0.113.50", "198.51.100.178", "10.0.0.2", "172.1
```

If you're doing anything with client IPs - rate limiting, geo-blocking, logging, analytics - you need to understand your proxy chain and configure `trust proxy` to match it. The wrong setting means you're either getting NGINX's IP instead of the client's, or you're vulnerable to IP spoofing where an attacker sets a fake `X-Forwarded-For` header and you trust it.

If you've never tested your app behind a reverse proxy, every piece of logic that depends on client IP, protocol, or host will behave differently in production.

SSL termination changes the protocol. Your app checks `req.protocol` or `req.secure` to enforce HTTPS. Locally, everything is HTTP. In production, NGINX terminates SSL and forwards plain HTTP to your app. Your app sees HTTP, thinks the connection is insecure, and either redirects infinitely or blocks the request. The fix is the `X-Forwarded-Proto` header, but you'd never know you needed it if you never tested behind NGINX with SSL.

Request body size limits differ. NGINX has a default `client_max_body_size` of 1 MB. Your app accepts file uploads of 10 MB. Locally, uploads work fine - no NGINX, no limit. In production, NGINX returns a `413 Request Entity Too Large` before your app ever sees the upload. Your app logs nothing. The user gets a cryptic error.

Timeouts kill long-running requests. Your app has an endpoint that processes a report and takes 45 seconds. Locally, it works because there's nothing between the browser and your app to enforce a timeout. In production, NGINX has a default `proxy_read_timeout` of 60 seconds (seems fine for 45 seconds), but the load balancer might have a 30-second idle timeout. The connection drops at 30 seconds, NGINX gets a 502, the user gets an error, and your app keeps processing a report nobody will ever see.

What a Staging Environment Actually Looks Like

A staging environment is a replica of your production infrastructure where you can test your application under real conditions before deploying to production. It doesn't need to be the same scale - you don't need 6 replicas and 3 nodes - but it needs to have the same *architecture*.

At minimum, your staging environment should have:

```
Staging:
  Browser -> NGINX (with SSL, same config as production)
    -> Your app (same Docker image as production)
    -> Your database (same version as production)
```

If production has a WAF, staging should have the same WAF rules. If production has a load balancer, staging should have the same type. If production runs on Docker Swarm with overlay networks, staging should too - even if it's a single-node Swarm.

The crucial part: **staging must run the exact same Docker image that will go to production.** Not a different build. Not a development build. The same image, pulled from the same registry, with the same tag. If you test image `myapp:1.4.72` in staging and it works, you deploy `myapp:1.4.72` to production. Not `myapp:latest`. Not `myapp:1.4.73` because you made "one small fix." The exact same image.

For smaller teams or solo developers who can't justify a full staging server, you can still replicate the production architecture locally with Docker Compose. Add NGINX to your local compose file:

```
services:
  nginx:
    image: nginx:stable-alpine
    ports:
      - "443:443"
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/certs:/etc/nginx/certs:ro
    networks:
      - app-network
    depends_on:
      - nodeserver

  nodeserver:
    build:
      context: ./nodeserver
    image: "yourregistry/nodeserver:latest"
    # NOT exposed to host - only accessible through NGINX
    # No ports mapping here
    networks:
      - app-network

  mongo:
    image: mongo:7
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network

volumes:
  mongo-data:

networks:
  app-network:
```

Now your local development hits NGINX first, just like production. You'll discover the WebSocket, header, timeout, and body size issues *before* they become production incidents. You'll configure `X-Forwarded-For` headers because you'll notice your app is logging NGINX's IP instead of your IP. You'll add the WebSocket upgrade config because you'll see your real-time features fail locally.

This is the gap that most developers have. They build a clean image, they push it to a registry, they deploy it - all the right steps - and then it breaks in production because the image was never the problem. The problem is everything *between the user and the image* that doesn't exist in their development environment. NGINX, WAF rules, SSL termination, load balancer timeouts, proxy headers - all of it invisible until it breaks at 2 AM with real users.

Build your staging environment to match production. Test your image behind the same proxy, the same WAF, the same network configuration. The deployment pipeline should be: develop -> test locally behind NGINX -> push image -> deploy to staging -> verify in staging -> promote to production. Every layer of that pipeline catches a different category of bug. Skip one and you're gambling.

TROUBLESHOOTING ON YOUR DEVELOPMENT MACHINE: THE COMPLETE GUIDE

When something goes wrong - a build fails, a container won't start, services can't talk to each other, performance tanks - most developers start guessing. They restart Docker. They `docker compose down && up`. They rebuild everything from scratch. They Google the error message and try whatever Stack Overflow says without understanding why. Thirty minutes later they've solved nothing, and they've destroyed their build cache in the process.

Stop guessing. Troubleshooting Docker is systematic. Every problem falls into one of five categories: build failures, container startup failures, network issues, performance problems, or image/layer issues. Each category has a specific debugging sequence, specific commands, and specific fixes. Learn the sequence once and you can diagnose anything.

Category 1: Build Failures

Your `docker build` or `docker compose build` fails. The image doesn't get created. This is the most common problem developers hit, and the error messages range from obvious to cryptic.

The Debugging Sequence

Step 1 - Read the error output. All of it.

Docker tells you exactly which step failed. The output looks like this:

```
=> [builder 4/6] RUN npm ci --omit=dev           12.3s
=> ERROR [builder 5/6] COPY . .                    0.0s
------
 > [builder 5/6] COPY . .:
------
ERROR: failed to solve: failed to compute cache key: failed to calculate checks
of ref: "/app/config/production.json": not found
```

The error is in the last few lines. Docker tried to `COPY` a file that doesn't exist in the build context. The fix is either adding the file or checking your `.dockerignore`.

Step 2 - Identify which build stage failed:

```
# See which layers are cached and which rebuilt
docker build --progress=plain -t myapp:debug .
```

The `--progress=plain` flag shows the full output of every step instead of the collapsed view. Every line is visible. If a `RUN` command fails, you see its complete stdout and stderr.

Step 3 - Debug inside a failed build stage:

When a build fails at step 5 of 8, the first 4 steps completed successfully. You can run a shell in the last successful layer:

```
# Build up to a specific stage
docker build --target builder -t myapp:debug-builder .

# Shell into it to investigate
docker run --rm -it myapp:debug-builder /bin/sh

# Now you're inside the container at the state where the build failed
# Check if files exist, if paths are correct, if tools are installed
ls -la /app/
which npm
node --version
cat /etc/os-release
```

This is the single most powerful build debugging technique. Instead of guessing why step 5 failed, you drop into the container at step 4 and look around.

Common Build Errors and Fixes

"COPY failed: file not found"

```
# Check what Docker actually sees as the build context
# This command shows you exactly what gets sent to the Docker daemon
docker build --progress=plain . 2>&1 | head -5
# Look for: "transferring context: 45.2MB" - is that size reasonable?

# Check your .dockerignore - you might be excluding the file
cat .dockerignore

# List the build context manually (simulates what Docker sees)
# Everything not in .dockerignore gets sent
find . -not -path './.git/*' -not -path './node_modules/*' | head -50
```

The file path in your `COPY` instruction is relative to the build context (the `.` in `docker build .`), not relative to the Dockerfile. If your Dockerfile is at `./docker/Dockerfile` but your build context is `.`, then `COPY ./config.json /app/` looks for `./config.json` in the project root, not in `./docker/`.

"npm ERR! network" or "Could not resolve host"

```
# Test DNS resolution from inside a build
docker run --rm alpine nslookup registry.npmjs.org

# Check Docker's DNS configuration
docker run --rm alpine cat /etc/resolv.conf

# Check if Docker Desktop can reach the internet
docker run --rm alpine wget -qO- https://registry.npmjs.org/ | head -1

# If behind a corporate proxy, check Docker Desktop settings:
# Settings -> Resources -> Proxies
# Or set them via environment:
docker build --build-arg HTTP_PROXY=http://proxy:8080 \
             --build-arg HTTPS_PROXY=http://proxy:8080 .
```

"no space left on device"

```
# See exactly what's consuming disk space
docker system df

# Detailed breakdown
docker system df -v

# Safe cleanup - removes stopped containers, dangling images, build cache
docker system prune

# Aggressive cleanup - also removes all unused images
docker system prune -a

# Nuclear - also removes unused volumes (DELETES DATABASE DATA)
docker system prune -a --volumes

# If you just want to clear the build cache
docker builder prune

# Clear build cache older than 24 hours
docker builder prune --filter until=24h

# See Docker Desktop's virtual disk size on Mac
ls -lh ~/Library/Containers/com.docker.docker/Data/vms/0/data/Docker.raw
```

"exec format error" or "standard_init_linux.go: exec user process caused: exec format error"

You're building an image on one architecture (Apple Silicon ARM64) and trying to run it on another (x86_64), or vice versa:

```
# Check your current platform
docker version --format '{{.Server.Arch}}'

# Build for a specific platform
docker build --platform linux/amd64 -t myapp:amd64 .

# Build for multiple platforms
docker buildx build --platform linux/amd64,linux/arm64 -t myapp:multi .

# Check what platform an image was built for
docker inspect myapp:latest | grep Architecture
```

Build step runs but produces wrong results (silent failures)

```
# Run the specific command interactively inside the build stage
docker build --target builder -t myapp:debug .
docker run --rm -it myapp:debug /bin/sh

# Inside the container, run the failing command manually:
npm ci --omit=dev 2>&1
# or
node -e "require('./server.js')"
# or whatever command is behaving unexpectedly

# Check environment variables that might affect the build
env | sort

# Check which versions of tools are installed
node --version
npm --version
python3 --version 2>/dev/null
```

Category 2: Container Startup Failures

Your image builds successfully, but `docker compose up` shows a container that immediately exits or keeps restarting.

The Debugging Sequence

Step 1 - Check container status and exit code:

```
# Show ALL containers, including stopped ones
docker ps -a

# Output:
# CONTAINER ID  IMAGE         COMMAND              STATUS                     PO
# abc123        myapp:latest  "node server.js"     Exited (1) 5 seconds ago
# def456        mongo:7       "mongod"             Up 3 minutes               27
```

The STATUS column tells you everything. `Exited (1)` means the process returned exit code 1.

Step 2 - Read the container logs:

```
# Full logs of the stopped container
docker logs abc123

# Last 50 lines (if the logs are long)
docker logs --tail 50 abc123

# With timestamps
docker logs --tail 50 -t abc123

# If the container is restart-looping, follow the logs in real time
docker logs -f abc123
```

Step 3 - If logs don't explain it, inspect the container:

```
# Full container configuration and state
docker inspect abc123

# Key things to look for:
docker inspect abc123 | grep -A 5 '"State"'
# Shows: Status, ExitCode, OOMKilled, Error, StartedAt, FinishedAt

docker inspect abc123 | grep -A 3 '"OOMKilled"'
# If true, the container ran out of memory

docker inspect abc123 | grep -A 10 '"Config"'
# Shows: Cmd, Entrypoint, Env, Image - verify these are correct
```

Step 4 - If you need to get inside a container that keeps crashing, override the entrypoint:

```
# Start the container with a shell instead of the app
docker run --rm -it --entrypoint /bin/sh myapp:latest

# Now you're inside the container that was crashing
# Check if files exist
ls -la /app/
cat /app/package.json

# Try running the app manually to see the real error
node server.js

# Check if the right environment variables are set
env | grep MONGO
env | grep NODE_ENV

# Test if you can reach external services
wget -qO- http://google.com
nslookup mongo
```

This is crucial. When a container crashes on startup, you can't `docker exec` into it because it's not running. The `--entrypoint /bin/sh` override lets you get inside the image and investigate.

`Exit Code Reference`

EXIT CODE	MEANING	COMMON CAUSE	FIX
0	Success (clean exit)	The process finished normally - but you expected it to keep running	Your `CMD` runs a script that completes instead of a long-running process. Use `CMD ["node", "server.js"]` not `CMD ["npm", "run", "build"]`
1	Application error	Unhandled exception, missing env var, bad config	Read the logs - the stack trace tells you exactly what failed
126	Permission denied	Entrypoint script isn't executable, or has Windows line endings	`RUN chmod +x /app/entrypoint.sh` - but if that doesn't fix it, run `dos2unix` first. See the Line Endings section
127	Command not found	The binary in CMD/ENTRYPOINT doesn't exist in the image	Check your multi-stage build - did you copy the binary? Check the PATH
137	OOMKilled (SIGKILL)	Container exceeded its memory limit	Increase `deploy.resources.limits.memory` or fix the memory leak
139	Segfault (SIGSEGV)	Native module crash, bad binary for the architecture	Check platform (ARM vs x86), rebuild native modules
143	SIGTERM (graceful stop)	Container received shutdown signal	Normal during `docker compose down` - only a problem if unexpected

Category 3: Network Issues Between Containers

This is the most frustrating category because the symptoms are vague - "it can't connect" - and there are a dozen possible causes. Follow this sequence exactly.

The Debugging Sequence

Step 1 - Verify both containers are running:

```
docker ps
# Both services must show "Up" status
# If one is restarting or exited, fix that first (Category 2)
```

Step 2 - Verify both containers are on the same network:

```
# List all networks
docker network ls

# Inspect a specific network to see which containers are attached
docker network inspect myapp_app-network

# Look for the "Containers" section in the output:
# "Containers": {
#   "abc123": { "Name": "myapp-nodeserver-1", "IPv4Address": "172.18.0.3/16" },
#   "def456": { "Name": "myapp-mongo-1", "IPv4Address": "172.18.0.2/16" }
# }
```

If both containers aren't listed under the same network, they can't communicate. Check your compose file - both services must declare the same network.

Step 3 - Test DNS resolution from inside the source container:

```
# Shell into the container that's trying to connect
docker exec -it myapp-nodeserver-1 /bin/sh

# Check DNS resolver configuration
cat /etc/resolv.conf
# Should show: nameserver 127.0.0.11 (Docker's internal DNS)

# Resolve the target service name
nslookup mongo
# Should return an IP address like 172.18.0.2

# If nslookup isn't available (slim images), try:
getent hosts mongo
# Or:
ping -c 1 mongo
# Or install it:
apt-get update && apt-get install -y dnsutils  # Debian
apk add bind-tools  # Alpine
```

If DNS resolution fails, the services are either on different networks or you're using the wrong service name. The DNS name is the **service name from your compose file**, not the container name, not the image name.

Step 4 - Test connectivity to the target port:

```
# From inside the source container:

# Test TCP connection to the target service
wget -qO- http://mongo:27017 --timeout=5
# Or:
nc -zv mongo 27017
# Or for HTTP services:
wget -qO- http://nodeserver:3000/health --timeout=5
curl -f http://nodeserver:3000/health
```

If DNS resolves but the connection fails:

```
# Check if the target is actually listening on the expected port
docker exec -it myapp-mongo-1 /bin/sh
netstat -tlnp   # Or: ss -tlnp
# Look for the port in the output. If it's not there, the service
# isn't listening - it might still be starting up, or the port is wrong
```

Step 5 - Check for port conflicts on the host:

```
# See what ports Docker has mapped to the host
docker ps --format "table {{.Names}}\t{{.Ports}}"

# Check if something else on your machine is using the same port
lsof -i :3000    # Mac/Linux
netstat -ano | findstr :3000  # Windows

# If port 3000 is already in use by another process, Docker can't bind to it
```

Step 6 - Deep packet inspection (when nothing else works):

```
# Install tcpdump in the container (if not present)
docker exec -it myapp-nodeserver-1 /bin/sh
apt-get update && apt-get install -y tcpdump  # Debian
apk add tcpdump  # Alpine

# Watch DNS queries (port 53)
tcpdump -i any port 53

# Watch traffic to a specific port
tcpdump -i any port 27017

# Watch all traffic between two containers
tcpdump -i any host 172.18.0.2

# Save capture to file for analysis in Wireshark
tcpdump -i any -w /tmp/capture.pcap port 3000
docker cp myapp-nodeserver-1:/tmp/capture.pcap ./capture.pcap
```

The Most Common Network Mistakes

SYMPTOM	CAUSE	FIX
"Connection refused"	Target service isn't running or isn't listening on that port	Check `docker ps`, check the port number, wait for startup
"Host not found" / DNS failure	Services on different networks, or wrong service name	Check compose file networks, use the service name (not container name)
Connection hangs / timeout	Firewall, wrong IP, or service is listening on 127.0.0.1 inside the container	Make your app listen on `0.0.0.0`, not `localhost` or `127.0.0.1`
"Connection reset"	Target container crashed while connecting	Check target container logs (Category 2)
Works from host but not from container	Accessing `localhost` from inside a container means the container itself, not the host	Use the service name instead of `localhost`
Port conflict	Two services mapping to the same host port	Change the host port in `ports:` mapping

The `0.0.0.0` gotcha deserves special attention. Many frameworks default to listening on `127.0.0.1` (localhost). Inside a container, `127.0.0.1` means "this container only" - other containers can't reach it. Your application must listen on `0.0.0.0` (all interfaces):

```
// WRONG - only accessible from inside this container
app.listen(3000, '127.0.0.1');

// CORRECT - accessible from other containers on the same network
app.listen(3000, '0.0.0.0');

// ALSO CORRECT - Node.js defaults to 0.0.0.0 when host is omitted
app.listen(3000);
```

Category 4: Performance Problems

Your containers work, but they're slow. Builds take forever. The app is sluggish. Docker Desktop is eating your CPU. Here's how to find out what's actually wrong.

Diagnosing Slow Containers

Step 1 - Check real-time resource usage:

```
# Live stats for all running containers
docker stats

# Output:
# CONTAINER ID  NAME       CPU %   MEM USAGE / LIMIT   MEM %   NET I/O       BL
# abc123        nodeapp    45.2%   345MiB / 400MiB     86.2%   1.2MB / 500kB  5
# def456        mongo      12.3%   285MiB / 1GiB       27.8%   5.6MB / 3MB    2

# Stats for a specific container
docker stats myapp-nodeserver-1

# One-shot (non-streaming) - useful for scripts
docker stats --no-stream
```

What to look for:

- **CPU % is consistently above 100%** - Your process is CPU-bound. Check your application code for tight loops, unoptimized algorithms, or blocking operations

- **MEM % is above 80%** - You're approaching the memory limit. Increase the limit or find the memory leak
- **MEM USAGE keeps climbing** - Memory leak. Profile your application
- **BLOCK I/O is very high** - Heavy disk reads/writes. On macOS, this is often bind mounts (see the macOS performance section in Pitfalls)
- **NET I/O is unexpectedly high** - Chatty service-to-service communication, or a container making too many external requests

Step 2 - Inspect processes inside the container:

```
docker exec -it myapp-nodeserver-1 /bin/sh

# Show running processes with CPU and memory
top
# or
ps aux

# Check what a Node.js process is actually doing
# (requires node --inspect in your CMD)
# Add to your Dockerfile CMD:
# CMD ["node", "--inspect=0.0.0.0:9229", "server.js"]
# Then connect Chrome DevTools to localhost:9229
```

Step 3 - Check Docker Desktop resource allocation:

```
# How much resource does the Docker VM have?
docker info | grep -E "CPUs|Total Memory"
# CPUs: 4
# Total Memory: 7.775GiB

# If this is lower than expected, increase in:
# Docker Desktop -> Settings -> Resources
```

Step 4 - Check if macOS file sharing is the bottleneck:

```
# Benchmark file I/O inside a container with a bind mount
docker run --rm -v $(pwd):/mnt alpine sh -c \
  "time dd if=/dev/zero of=/mnt/testfile bs=1024 count=10000"

# Compare to native file I/O (no bind mount)
docker run --rm alpine sh -c \
  "time dd if=/dev/zero of=/tmp/testfile bs=1024 count=10000"

# If the bind mount version is 10x+ slower, you have the macOS
# file sharing overhead. Solutions:
# 1. Use VirtioFS (Docker Desktop -> General -> file sharing)
# 2. Use anonymous volumes for node_modules
# 3. Narrow your bind mounts (mount ./src not .)
```

Diagnosing Slow Builds

Step 1 - Check what's being sent as build context:

```
# Watch the build output for the context size
docker build . 2>&1 | head -3
# "Sending build context to Docker daemon  1.2GB"
# If this is hundreds of MB, your .dockerignore is wrong

# Check your .dockerignore exists and is correct
cat .dockerignore

# Minimum .dockerignore for Node.js:
# node_modules
# .git
# npm-debug.log
# .env
# dist
# coverage
# .nyc_output
```

If you're sending 1+ GB of build context, the build is slow before it even starts. Fix your `.dockerignore`.

Step 2 - Check build cache effectiveness:

```
# Build with full output to see cache hits
docker build --progress=plain -t myapp:latest . 2>&1 | grep -E "CACHED|DONE"

# CACHED steps are using the cache (fast)
# DONE steps are executing (slow)

# See which step takes the longest
docker build --progress=plain -t myapp:latest . 2>&1 | grep -E "\[.*\].*[0-9]+\
```

If you see `RUN npm ci` executing on every build even when you haven't changed dependencies, your layer ordering is wrong. `COPY package*.json` must come before `RUN npm ci`, and `COPY . .` must come after.

Step 3 - Profile build time per step:

```
# Use BuildKit's built-in timing
DOCKER_BUILDKIT=1 docker build --progress=plain -t myapp:latest . 2>&1 | \
  grep -E "^\#[0-9]+ (DONE|CACHED|ERROR)" | \
  sort -t' ' -k3 -rn

# This shows you which steps are slowest
# Common culprits:
# - RUN npm ci (10-60 seconds depending on deps)
# - RUN apt-get install (20-60 seconds)
# - COPY . . (slow on macOS with many files)
```

Step 4 - Inspect build cache state:

```
# See the build cache and its size
docker builder du

# See individual cache entries
docker builder du --verbose

# If cache is corrupted or stale, clear it
docker builder prune

# Clear only cache older than 7 days
docker builder prune --filter until=168h
```

Common Build Performance Fixes

PROBLEM	SYMPTOM	FIX
Build context too large	"Sending build context" shows hundreds of MB	Fix `.dockerignore` - exclude `node_modules`, `.git`, `dist`, data files
Dependencies reinstall every build	`npm ci` runs on every build even for code-only changes	Move `COPY package*.json` BEFORE `RUN npm ci`, put `COPY . .` AFTER
Base image pulls every time	Build starts with downloading the base image	Use `docker pull` only when you want to update. Docker caches base images locally
Large `RUN apt-get` steps	System package installation takes 30+ seconds	Combine all `apt-get` into one `RUN` layer and clean up in the same layer
Slow npm on macOS	`npm ci` takes 3-5x longer in Docker on Mac vs native	Move `npm ci` to a non-bind-mounted layer (it already should be - don't bind-mount `node_modules`)
Build cache not working at all	Every layer rebuilds every time	Check if you're using `--no-cache` as a habit. Check if a prior `COPY . .` invalidates subsequent layers

Diagnosing Slow Multi-Stage Builds

Multi-stage builds have a specific performance characteristic: each stage builds independently, and later stages wait for `COPY --from=builder` to complete. If your builder stage is slow, everything downstream waits.

```
# Time each stage separately by building targets
time docker build --target builder -t myapp:builder .
time docker build --target production -t myapp:prod .

# If the builder stage is slow, isolate which step:
docker build --target builder --progress=plain -t myapp:builder . 2>&1 | \
  grep -E "DONE [0-9]"
```

Common fixes for slow multi-stage builds:

Slow native module compilation - packages like `bcrypt`, `sharp`, and `canvas` need to compile C/C++ code. This is inherently slow (10-30+ seconds). Make sure this step is cached by putting `COPY package*.json` and `RUN npm ci` before `COPY . .`:

```
# CORRECT ORDER - npm ci is cached unless package.json changes
COPY package.json package-lock.json ./
RUN npm ci
COPY . .

# WRONG ORDER - npm ci runs every time you change any file
COPY . .
RUN npm ci
```

Multiple `apt-get install` layers - each `RUN` is a separate layer. Combine all system package installations:

```
# SLOW - three layers, three cache checks, three installs
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y make g++

# FAST - one layer, one cache check, cleanup in same layer
RUN apt-get update && \
    apt-get install -y --no-install-recommends python3 make g++ && \
    rm -rf /var/lib/apt/lists/*
```

Builder stage has too many files - if your builder `COPY . .` copies thousands of files, it's slow on macOS even if the files haven't changed. Narrow what you copy:

```
# Instead of copying everything and then installing
COPY . .
RUN npm run build

# Copy only what the build needs
COPY package.json package-lock.json ./
RUN npm ci
COPY src/ ./src/
COPY tsconfig.json ./
RUN npm run build
```

Category 5: Image and Layer Debugging

Your image is too big, you're not sure what's inside it, or you need to understand why the image changed between builds.

Inspecting Image Contents

```
# See image size and layer count
docker image ls myapp
docker image inspect myapp:latest | grep -E "Size|Created"

# See every layer and its size
docker history myapp:latest
# Each row shows: the instruction, the layer size, and when it was created
# Large layers are your optimization targets

# Detailed history with full commands (not truncated)
docker history --no-trunc myapp:latest

# See what files changed in each layer
# Install dive - the best tool for image inspection
# Mac: brew install dive
# Linux: download from https://github.com/wagoodman/dive
dive myapp:latest
# This shows you a layer-by-layer filesystem diff
# You can see exactly which files were added/changed/deleted in each layer
```

Finding What's Making Your Image Large

```
# Quick size check
docker image ls myapp:latest --format "{{.Size}}"

# Layer-by-layer breakdown
docker history myapp:latest --format "{{.Size}}\t{{.CreatedBy}}" | \
  sort -rh | head -10
# Shows the 10 largest layers

# If you don't have dive, you can export and inspect manually:
docker save myapp:latest | tar -xf - -C /tmp/image-inspect/
ls -la /tmp/image-inspect/
# Each directory is a layer. Check their sizes.
```

Common image bloat causes and fixes:

CAUSE	HOW TO FIND IT	FIX	
Build tools in final image	`docker history` shows `apt-get install python3 make g++` in a single-stage build	Use multi-stage builds - build tools stay in the builder stage	
`node_modules` with devDependencies	`docker exec myapp ls node_modules \`	wc -l shows hundreds of packages	Use `npm ci --omit=dev` in the production stage
Apt cache left behind	`docker history` shows a large `apt-get install` layer	Add `rm -rf /var/lib/apt/lists/*` in the same `RUN` layer	
`.git` directory inside image	`.dockerignore` missing `.git` entry	Add `.git` to `.dockerignore`	
Full base image	Using `node:20` (~1 GB) instead of `node:20-bookworm-slim` (~200 MB)	Switch to `-bookworm-slim` base images	
Unnecessary files copied	`COPY . .` includes test files, docs, data, logs	Use `.dockerignore` or selective `COPY` commands	

Comparing Images Between Builds

```
# Compare sizes of two tagged versions
docker image ls | grep myapp

# Compare layer history
diff <(docker history myapp:1.4.71 --no-trunc) \
     <(docker history myapp:1.4.72 --no-trunc)

# Find which layers are shared between two images
docker inspect myapp:1.4.71 --format '{{json .RootFS.Layers}}' | python3 -m jso
docker inspect myapp:1.4.72 --format '{{json .RootFS.Layers}}' | python3 -m jso
# Shared layer hashes = shared content (not re-downloaded or re-stored)
```

Quick Reference: Every Troubleshooting Command in One Place

Keep this section bookmarked. These are the commands you'll reach for first.

Container Lifecycle

```
docker ps                          # Running containers
docker ps -a                       # ALL containers (including stopped)
docker logs <container>            # Container output
docker logs --tail 100 -f <container>  # Last 100 lines, follow mode
docker logs -t <container>         # With timestamps
docker inspect <container>         # Full config and state
docker exec -it <container> /bin/sh    # Shell into running container
docker run --rm -it --entrypoint /bin/sh <image>  # Shell into crashed containe
docker stop <container>            # Graceful stop (SIGTERM)
docker kill <container>            # Force stop (SIGKILL)
docker rm <container>              # Remove stopped container
docker rm -f <container>           # Force remove (even if running)
```

Compose Operations

```
docker compose up -d               # Start all services (detached)
docker compose up -d --build       # Start with rebuild
docker compose up -d --build nodeserver  # Rebuild single service
docker compose down                # Stop and remove containers
docker compose down -v             # Stop, remove containers AND volumes
docker compose logs -f nodeserver  # Follow logs for one service
docker compose ps                  # Status of compose services
docker compose restart nodeserver  # Restart single service
docker compose exec nodeserver /bin/sh  # Shell into compose service
```

Image and Build

```
docker image ls                     # List local images
docker image ls --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"  # Clean
docker history <image>              # Layer-by-layer breakdown
docker history --no-trunc <image>  # Full commands (not truncated)
docker build --progress=plain .    # Build with full output
docker build --target builder .    # Build specific stage only
docker build --no-cache .          # Build ignoring all cache
docker builder du                   # Build cache disk usage
docker builder prune                # Clear build cache
docker inspect <image>              # Image metadata and config
docker save <image> > image.tar    # Export image to file
```

Network Debugging

```
docker network ls                   # List all networks
docker network inspect <network>   # Containers attached, subnet, gateway
docker exec <container> nslookup <service>    # DNS resolution test
docker exec <container> wget -qO- http://<service>:<port>/health  # HTTP test
docker exec <container> nc -zv <service> <port>   # TCP connectivity test
docker exec <container> cat /etc/resolv.conf      # DNS config
docker exec <container> netstat -tlnp             # Listening ports
```

Resource Monitoring

```
docker stats                        # Live CPU/memory/network per container
docker stats --no-stream           # One-shot snapshot
docker system df                    # Disk usage (images, containers, volumes, c
docker system df -v                 # Detailed disk usage
docker inspect <container> | grep -A 5 OOMKilled  # Check for memory kills
docker info | grep -E "CPUs|Memory"               # Docker VM resources
```

Cleanup

```
docker system prune                # Safe: stopped containers, dangling images,
docker system prune -a             # Aggressive: all unused images too
docker system prune -a --volumes   # Nuclear: also removes unused volumes
docker image prune                 # Only dangling images
docker container prune             # Only stopped containers
docker volume prune                # Only unused volumes (DELETES DATA)
docker builder prune               # Only build cache
```

Swarm Operations

```
docker swarm init --advertise-addr <ip>:2377  # Initialize Swarm
docker stack deploy -c docker-compose.yml mystack  # Deploy stack
docker stack ps mystack            # Container status across nodes
docker service ls                  # All services
docker service logs -f mystack_nodeserver  # Service logs (all replicas)
docker service inspect mystack_nodeserver  # Service config
docker service scale mystack_nodeserver=4  # Scale replicas
docker service update --force mystack_nodeserver  # Force re-pull and restart
docker service update --image myapp:1.4.72 mystack_nodeserver  # Update image
docker node ls                     # List Swarm nodes
docker node update --availability=drain <node>  # Drain node for maintenance
docker node update --availability=active <node>  # Bring node back
```

THE REAL WORKFLOWS: HOW TEAMS ACTUALLY SHIP WITH SWARM VS KUBERNETES

Most comparisons between Docker Swarm and Kubernetes focus on feature matrices - autoscaling, service meshes, CRDs. That's not what matters day-to-day. What matters is the workflow: what does a developer do from the moment they write code to the moment it's running in production? How many files do they touch? How many tools do they need to learn? How many things can break between "git push" and "live in production"?

This section compares the actual workflows - the build, tag, push, deploy cycle - that teams use in each system, and the pain points they complain about in each. This isn't a feature comparison. It's a workflow comparison.

The Swarm Deployment Workflow

The typical Swarm workflow is notable for how few moving parts it has. You need three things: a CI system (GitHub Actions, GitLab CI, Jenkins), a container registry (Docker Hub, GHCR, a private registry), and SSH access to a Swarm manager node. That's it.

The full cycle looks like this:

```
Developer pushes to main
        |
        v
CI pipeline triggers
        |
        |-- 1. Build image:  docker build -t registry/app:1.4.72 .
        |-- 2. Push image:   docker push registry/app:1.4.72
        |-- 3. SSH to manager node
        \-- 4. Deploy:       docker stack deploy -c docker-compose.yml myapp
        |
        v
Swarm pulls new image, performs rolling update
```

A real GitHub Actions workflow for Swarm deployment:

```
name: Deploy
on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login to Registry
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and Push
        uses: docker/build-push-action@v6
        with:
          push: true
          tags: |
            ghcr.io/yourorg/app:${{ github.run_number }}
            ghcr.io/yourorg/app:latest

      - name: Deploy to Swarm
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ secrets.SWARM_HOST }}
          username: deploy
          key: ${{ secrets.SSH_KEY }}
          script: |
            docker login ghcr.io -u ${{ github.actor }} -p ${{ secrets.GITHUB_T
            export IMAGE_TAG=${{ github.run_number }}
            docker stack deploy -c /opt/stacks/myapp/docker-compose.yml \
              --with-registry-auth myapp
```

The compose file on the server:

```
services:
  app:
    image: ghcr.io/yourorg/app:${IMAGE_TAG:-latest}
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
        order: start-first
      rollback_config:
        parallelism: 1
      resources:
        limits:
          memory: 512M
          cpus: "0.50"
    healthcheck:
      test: wget -qO- http://localhost:3000/health || exit 1
      interval: 15s
      timeout: 5s
      retries: 3
    secrets:
      - db_password
    networks:
      - app-network

secrets:
  db_password:
    external: true

networks:
  app-network:
    driver: overlay
```

That's the entire deployment system. One CI workflow file. One compose file. One SSH connection. The developer touches two files in their repo (source code and the workflow YAML), and the rest happens automatically.

How teams manage secrets in Swarm:

Swarm has built-in secrets management. You create secrets on the manager node, and they're encrypted in the Raft log, mounted as files inside containers at `/run/secrets/`. They're never exposed as environment variables and are wiped from memory when the container stops.

```
# Create a secret on the manager
echo "s3cure_p@ssw0rd" | docker secret create db_password -

# Or from a file
docker secret create tls_cert ./cert.pem

# List secrets
docker secret ls

# Secrets are mounted as files inside the container
# Your app reads: /run/secrets/db_password
```

This is actually more secure by default than Kubernetes secrets (which are base64-encoded, not encrypted, and can be passed as environment variables where they're visible in process listings and logs).

The Kubernetes Deployment Workflow

The Kubernetes workflow has significantly more moving parts. A typical modern setup involves: a CI system, a container registry, a manifest repository (separate from app code), a templating tool (Helm or Kustomize), and a GitOps controller (ArgoCD or Flux). Five components where Swarm needs three.

The full cycle looks like this:

```
Developer pushes to main (app repo)
        |
        v
CI pipeline triggers
        |
        |-- 1. Build image:     docker build -t registry/app:a3f8c2d .
        |-- 2. Push image:      docker push registry/app:a3f8c2d
        |-- 3. Clone GitOps repo
        |-- 4. Update image tag in Helm values.yaml or Kustomize overlay
        \-- 5. Commit and push to GitOps repo
        |
        v
ArgoCD detects change in GitOps repo
        |
        |-- 6. Renders Helm templates / applies Kustomize patches
        |-- 7. Compares rendered manifests against cluster state
        \-- 8. Applies diff to cluster (creates/updates Deployments, Services,
        |
        v
Kubernetes performs rolling update
```

A real GitHub Actions workflow for Kubernetes with ArgoCD:

```yaml
name: CI
on:
  push:
    branches: [main]
    paths-ignore:
      - "helm/**"
      - "k8s/**"

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set Docker Tag
        id: vars
        run: echo "docker_tag=${{ github.sha }}" >> $GITHUB_OUTPUT

      - uses: docker/setup-buildx-action@v3

      - uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and Push
        uses: docker/build-push-action@v6
        with:
          push: true
          tags: |
            yourorg/app:${{ steps.vars.outputs.docker_tag }}
            yourorg/app:latest

      - name: Update Helm Values
        run: |
          sed -i "s/tag: .*/tag: \"${{ steps.vars.outputs.docker_tag }}\"/" \
            helm/app-chart/values.yaml

      - name: Push Helm Update
        run: |
          git config user.email "ci@yourorg.com"
          git config user.name "CI Bot"
          git add helm/app-chart/values.yaml
          git commit -m "Update image tag to ${{ steps.vars.outputs.docker_tag
          git push
```

Notice the `[skip-ci]` in the commit message - without it, the CI pipeline triggers again when it pushes the updated Helm values, creating an infinite loop. This is a gotcha that every Kubernetes team learns the hard way. The `paths-ignore` filter at the top also helps prevent this.

The Helm chart structure this pipeline updates:

```
helm/app-chart/
|-- Chart.yaml
|-- values.yaml              # <- CI updates the tag here
|-- templates/
|   |-- deployment.yaml      # ~60 lines
|   |-- service.yaml         # ~20 lines
|   |-- ingress.yaml         # ~30 lines
|   |-- configmap.yaml       # ~15 lines
|   |-- secret.yaml          # ~15 lines
|   |-- hpa.yaml             # ~20 lines
|   |-- serviceaccount.yaml  # ~10 lines
|   \-- _helpers.tpl         # ~30 lines
\-- charts/                  # sub-chart dependencies
```

That's 8+ template files generating around 200 lines of Kubernetes manifests for a single service. In Swarm, the equivalent is one compose file of roughly 30-50 lines.

ArgoCD then monitors the repo and syncs:

ArgoCD is a separate system running inside your Kubernetes cluster. You install it, configure it to watch your GitOps repository, and it continuously reconciles the cluster state with what's in Git. If someone uses `kubectl` to change something directly, ArgoCD detects the drift and reverts it. This is powerful - but it's also another system you need to install, configure, monitor, and keep updated.

The File Count Problem

This is where the difference becomes visceral. Here's what a developer manages for a single service:

WHAT YOU MANAGE	DOCKER SWARM	KUBERNETES (HELM + ARGOCD)
Application code	Yes Same	Yes Same
Dockerfile	1 file	1 file
Compose / manifests	1 file (~40 lines)	8-12 template files (~200+ lines)
CI workflow	1 file	1 file (but more complex)
GitOps controller config	Not needed	ArgoCD Application manifest
Values per environment	Environment variables or `.env`	values-dev.yaml, values-staging.yaml, values-prod.yaml
Total config files	**2-3**	**12-18**

Multiply this by the number of services in your application. A team with 5 microservices in Swarm manages maybe 10-15 config files total. The same team in Kubernetes manages 60-90 files. One developer quoted in community discussions counted 47 YAML files for a basic web application with a database.

Local Development: Where the Gap Is Widest

This is the most painful difference for developers working day-to-day.

Swarm local development:

```
# Your entire local dev workflow
docker compose up -d
# Edit code, see changes via bind mounts
# Done

# Want to test the Swarm-specific deploy behavior?
docker swarm init
docker stack deploy -c docker-compose.yml myapp
# Same compose file. Same commands you already know.
docker swarm leave --force
```

Docker Compose is the local development tool, and the same file (with minor additions in the `deploy:` block) becomes the production deployment manifest. The developer never leaves the Docker ecosystem.

Kubernetes local development:

To develop locally against Kubernetes, teams need to choose and set up one of: Minikube, Kind (Kubernetes IN Docker), k3d (k3s in Docker), or Docker Desktop's built-in Kubernetes. Then they need a development workflow tool on top of that: Skaffold, Tilt, DevSpace, or Telepresence. Each has its own configuration format, its own learning curve, and its own quirks.

```
# Start local cluster (pick one)
minikube start          # VM-based, heavier
kind create cluster     # Container-based, lighter
k3d cluster create      # k3s in Docker, lightest

# Then you need a dev workflow tool (pick one)
skaffold dev            # Google's tool, watches files, rebuilds, redeploys
tilt up                 # Dashboard-based, uses Starlark config
devspace dev            # File sync, hot reload, dev containers

# Or do it manually every time you change code:
docker build -t myapp:dev .
kind load docker-image myapp:dev    # Load into Kind's image store
kubectl apply -f k8s/               # Apply all manifests
kubectl rollout restart deployment/myapp  # Force pod restart
```

The manual cycle - build, load, apply, restart - takes 30-60 seconds per code change. Tools like Skaffold and Tilt automate this, but they're additional systems to learn and configure. Bret Fisher, a well-known Docker educator, put it this way: Compose is designed around the developer workflow, while Kubernetes is an ops tool for managing clusters. The compose file is typically a quarter the size of equivalent Kubernetes manifests.

Resource usage is also a factor. Running Kubernetes locally - even with Kind - requires the Kubernetes control plane components (API server, etcd, scheduler, controller-manager, CoreDNS, kube-proxy). A 2024 Spectro Cloud report found that over 77% of Kubernetes practitioners still have issues with their clusters, up from 66% in 2022.

Pain Points: What People Actually Complain About

Docker Swarm Pain Points

1. "Swarm feels abandoned"

This is the single most common complaint. Docker Inc. shifted focus to Docker Desktop and Docker Business, and Swarm development slowed dramatically. The GitHub roadmap issue asking Docker to clarify Swarm's status has been open since 2020. Community members worry about using a tool that might not get critical security fixes or new features.

The reality is nuanced: Swarm is part of the Docker Engine (specifically Moby/SwarmKit), which is still actively maintained. It gets bug fixes and security patches. But it doesn't get new features, and Docker's documentation increasingly de-emphasizes it. For teams evaluating orchestrators, the perception of abandonment is itself a risk - it makes hiring harder and makes management nervous.

2. "No autoscaling"

Swarm will maintain your declared replica count. If you say 5 replicas and 2 crash, it brings them back to 5. But it won't automatically scale from 3 to 10 based on CPU load. You either set a fixed replica count or write your own scaling script.

For many workloads, this is fine - you provision for your peak load and run at that level. But for spiky traffic patterns, it means you're either over-provisioning or accepting degraded performance during spikes.

3. "No fine-grained scheduling"

Swarm spreads services across nodes but gives you limited control over placement. You can use constraints (`node.labels.type == worker`) and placement preferences, but you can't do things like: force all containers in a stack to run on the same node (with failover), ensure two services are always co-located, or implement pod affinity/anti-affinity rules.

One community member described the frustration of having a web server and its database split across different nodes, adding network latency for every request, with no way to keep them co-located while preserving high availability.

4. "The overlay network loses client IPs"

Swarm's ingress routing mesh, which load-balances incoming requests across all nodes, replaces the client's source IP with an internal overlay network IP. Your application sees Docker's internal IP instead of the real client. The workaround is running your reverse proxy (NGINX, Caddy, Traefik) outside the Swarm with host networking, which defeats some of the point of orchestration.

5. "No built-in monitoring or logging"

Swarm provides `docker service logs` and `docker stats`, but there's no built-in dashboard, no metrics collection, no alerting. You have to set up Prometheus, Grafana, and a log aggregator yourself. Kubernetes doesn't come with these built-in either, but the ecosystem of Helm charts for deploying them is vastly larger and better-maintained.

6. "Secret rotation requires service restarts"

When you update a Swarm secret, you can't just update the existing secret - you create a new secret version and update the service to use it. This means a service restart. Kubernetes has the same limitation for secrets mounted as volumes, but some patterns (like external secret operators syncing from Vault) handle rotation more gracefully.

Kubernetes Pain Points

1. "YAML hell"

This is the most universal Kubernetes complaint. A simple web application deployment requires a Deployment manifest, a Service manifest, an Ingress manifest, possibly a ConfigMap, a Secret, a HorizontalPodAutoscaler, a ServiceAccount, and PodDisruptionBudget. Each is a separate YAML file with its own schema, its own apiVersion, and its own set of gotchas.

Community members describe counting 47 YAML files for a basic web app with a database. The Helm templating system - designed to reduce this - adds Go template syntax on top of YAML, creating what one developer called "a shitty meta programming language on top of YAML." Kustomize takes a different approach (patching base manifests), but adds its own complexity.

2. "The learning cliff"

The average time for a competent developer to become productive with Kubernetes has grown from 2-3 months in 2019 to 6-8 months in 2024, according to consulting reports. This isn't just about learning `kubectl` - it's about understanding Pods, Deployments, ReplicaSets, Services, Ingress, ConfigMaps, Secrets, Namespaces, RBAC, PersistentVolumes, StorageClasses, NetworkPolicies, ServiceAccounts, and the dozens of CRDs that most production clusters depend on.

Teams end up hiring dedicated Kubernetes engineers whose full-time job is operating the cluster. At that point, the infrastructure intended to simplify operations has become the thing that requires the most operations.

3. "Resource overhead"

Before you deploy a single application container, Kubernetes is already running: etcd (1-3 instances), kube-apiserver, kube-scheduler, kube-controller-manager, kube-proxy (on every node), CoreDNS (usually 2 replicas), and CNI plugin pods (on every node). Most production deployments add an ingress controller, cert-manager, and metrics-server.

One experienced Swarm user compared it to renting a house where Kubernetes takes 5 of the 8 rooms for its own furniture and tells you to use what's left. For small teams on modest infrastructure, this overhead is significant.

4. "Local development is a second job"

As described above, developing locally against Kubernetes requires choosing and configuring a local cluster tool, then choosing and configuring a development workflow tool on top of it. The gap between `docker compose up` and the equivalent Kubernetes local setup is immense.

Teams frequently end up maintaining two entirely separate configurations: Docker Compose for local development and Kubernetes manifests for staging/production. This means what you test locally isn't what runs in production, undermining one of the core promises of containerization.

5. "Networking is still unintuitive"

Services, Ingress, IngressClasses, LoadBalancers, ClusterIP, NodePort, NetworkPolicies, CNI plugins - Kubernetes networking requires understanding more abstractions than many developers will encounter in their entire career. Even experienced network engineers get tripped up. Community discussions regularly feature developers asking why something as fundamental as routing traffic between services remains so difficult.

6. "Tool sprawl and decision fatigue"

Kubernetes doesn't just require learning Kubernetes. A production cluster typically requires choosing and learning: a package manager (Helm vs Kustomize), a GitOps controller (ArgoCD vs Flux), a service mesh (Istio vs Linkerd vs none), a monitoring stack (Prometheus + Grafana + AlertManager), a logging solution (Loki, ELK, Fluentd), an ingress controller (NGINX vs Traefik vs Ambassador), a secrets manager (External Secrets Operator, Sealed Secrets, Vault), and a CI/CD integration.

Every one of these choices has trade-offs, every one requires configuration, and every one can break independently. The ecosystem's richness is simultaneously its greatest strength and its greatest burden.

The Workflow Comparison at a Glance

DIMENSION	DOCKER SWARM	KUBERNETES
Files to deploy one service	1 compose file + 1 CI workflow	8-12 manifests + Helm chart + CI workflow + ArgoCD config
Config language	Docker Compose YAML (same as development)	Kubernetes YAML + Helm Go templates or Kustomize patches
Deployment command	`docker stack deploy -c compose.yml app`	`kubectl apply`, `helm upgrade`, or ArgoCD auto-sync
Deploy mechanism	SSH to manager + stack deploy	CI pushes to GitOps repo -> ArgoCD syncs
Image tag update	Environment variable in compose file	`sed` values.yaml -> commit -> push -> ArgoCD detects
Local development	`docker compose up` (same file)	Minikube/Kind + Skaffold/Tilt (different toolchain)
Secrets management	Built-in, encrypted at rest, file-mount only	Base64-encoded by default, requires external tools for real security
Rollback	`docker service rollback` or redeploy previous tag	`kubectl rollout undo`, `helm rollback`, or ArgoCD revert commit
Learning curve	Hours to days (extends Docker knowledge)	Months (new concepts, new tools, new ecosystem)
Ecosystem tools needed	Registry + CI + SSH	Registry + CI + Helm/Kustomize + ArgoCD/Flux + Ingress controller + monitoring stack
Team size sweet spot	1-10 developers, < 20 nodes	10+ developers, or when you need autoscaling/multi-tenancy/advanced scheduling
Risk	Perceived abandonment, smaller community	Complexity, operational overhead, tool sprawl

When Each Workflow Makes Sense

Use Swarm when your team is small, your infrastructure is modest, your deployment needs are straightforward, and you value simplicity over flexibility. If you're a team of 1-5 developers running a handful of services on a few servers, Swarm's workflow will get you from code to production in a fraction of the time Kubernetes would take to set up. The single compose file for development and production means less configuration drift, fewer moving parts, and fewer things that can break between your laptop and production.

Use Kubernetes when you need autoscaling, multi-tenant isolation, advanced scheduling policies, or when your organization has (or can hire) dedicated platform engineers to operate the cluster. Kubernetes earns its complexity at scale - when you're running dozens of services across multiple teams, the abstractions that feel like overkill for a small project become essential governance tools. The GitOps workflow with ArgoCD provides audit trails, automatic drift correction, and multi-environment promotion that Swarm can't match without significant custom tooling.

The uncomfortable truth is that many teams adopt Kubernetes before they need it. They pay the full complexity cost - the YAML, the tooling, the operational overhead, the hiring - for workloads that would run perfectly well on a 3-node Swarm cluster. The decision should be based on your actual operational requirements, not on what looks best on a resume or what the industry hype cycle says you should be using.

DEVELOPMENT WORKFLOW PATTERNS THAT WORK

Pattern 1: Bind Mounts for Hot Reload

In development, you don't want to rebuild the image every time you change code. Use bind mounts to map your local source code into the container:

```
services:
  nodeserver:
    build: ./nodeserver
    image: yourregistry/nodeserver:latest
    volumes:
      # Bind mount source code for live editing
      - ./nodeserver:/app
      # But DON'T override node_modules from the image
      - /app/node_modules
    command: ["npx", "nodemon", "server.js"]
```

The anonymous volume `/app/node_modules` is critical. Without it, your local (possibly empty or wrong-platform) `node_modules` folder overwrites the one built inside the container. The anonymous volume preserves the container's `node_modules` while your source code comes from the host.

Important: This pattern is for development only. In production, everything is baked into the image. There are no bind mounts. Remove the `volumes` and `command` overrides in your production compose file.

Pattern 2: Environment-Specific Compose Files

Docker Compose supports file overrides. Use a base file for shared configuration and override files for environment specifics:

```
# Base configuration
docker-compose.yml

# Development overrides
docker-compose.dev.yml

# Production overrides
docker-compose.prod.yml
```

```
# docker-compose.yml (base)
services:
  nodeserver:
    image: yourregistry/nodeserver:latest
    init: true
    networks:
      - app-network
    deploy:
      resources:
        limits:
          memory: 400M

# docker-compose.dev.yml (development overrides)
services:
  nodeserver:
    build: ./nodeserver
    volumes:
      - ./nodeserver:/app
      - /app/node_modules
    environment:
      - NODE_ENV=development
      - LOG_LEVEL=debug
    ports:
      - "3000:3000"
      - "9229:9229"  # Node.js debugger port
    command: ["node", "--inspect=0.0.0.0:9229", "server.js"]
```

Run with:

```
# Development
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d

# Production (Swarm)
docker stack deploy -c docker-compose.yml -c docker-compose.prod.yml mystack
```

Pattern 3: Health Checks from Day One

Add health checks in development, not as an afterthought when deploying to Swarm. Swarm relies on health checks for rolling updates and automatic rollbacks. If your service doesn't have a health check, Swarm can't tell the difference between a healthy container and a broken one.

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 30s
```

Your `/health` endpoint doesn't need to be complicated:

```
app.get('/health', (req, res) => {
  // Check critical dependencies
  const mongoConnected = mongoose.connection.readyState === 1;

  if (mongoConnected) {
    res.status(200).json({ status: 'healthy', mongo: 'connected' });
  } else {
    res.status(503).json({ status: 'unhealthy', mongo: 'disconnected' });
  }
});
```

When Swarm does a rolling update, it starts the new container, waits for the `start_period`, then begins checking the health endpoint every `interval`. If the endpoint returns a non-200 status `retries` times, the container is marked unhealthy and Swarm rolls back. This entire mechanism is invisible to you - but only if you wrote the health check.

Pattern 4: Why You Must Handle Exit Codes (And What Ctrl+C Actually Does)

This is the section most Docker guides skip entirely, and it's the one that will save you from the most insidious production bugs. If you don't handle process signals and exit codes properly, you will leak memory, orphan database connections, corrupt data, and create failures that are impossible to reproduce.

What Happens When You Press Ctrl+C

When you're developing locally and you press Ctrl+C to stop your Node.js app, here's what *actually* happens at the operating system level:

1. Your terminal sends `SIGINT` (signal interrupt) to the foreground process

2. Node.js receives `SIGINT` and, by default, immediately terminates
3. The operating system reclaims all memory allocated to that process
4. All open file descriptors are closed
5. All TCP sockets are forcibly reset (not gracefully closed)

On your local machine, this is fine. The OS cleans up after you. Your MongoDB is running separately, so it handles the dropped connection. Your Redis reconnects automatically next time. No harm done.

But inside a Docker container, the rules change.

When Docker stops a container - whether from `docker compose down`, a rolling update, a health check failure, or a Swarm rebalance - it sends `SIGTERM` to PID 1 inside the container. Your application has exactly 10 seconds (the default `stop_grace_period`) to shut down. After 10 seconds, Docker sends `SIGKILL` - an uncatchable, unblockable kill signal. The process is dead. No cleanup. No "wait, I'm almost done." Dead.

Here's the problem: **if PID 1 in your container is not your application, SIGTERM never reaches your application.**

```
# WRONG - PID 1 is /bin/sh, not node
CMD npm start
# Docker runs: /bin/sh -c "npm start"
# /bin/sh receives SIGTERM but doesn't forward it to the node process
# After 10 seconds, Docker sends SIGKILL to the shell
# Node process is orphaned and killed without any signal

# CORRECT - PID 1 is node
CMD ["node", "server.js"]
# Docker runs: node server.js (directly, no shell wrapper)
# Node process IS PID 1 and receives SIGTERM directly
```

The difference between `CMD npm start` (shell form) and `CMD ["node", "server.js"]` (exec form) is the difference between your application receiving a graceful shutdown signal and being violently killed without warning.

What Actually Breaks Without Graceful Shutdown

This isn't theoretical. Here's what happens to real systems when your app gets killed without cleanup:

Database connection pool leak. Your application opens 10 connections to MongoDB at startup. During a rolling update, Docker kills the old container. Those 10 connections are now orphaned - MongoDB still thinks they're alive because the TCP keepalive timer hasn't expired yet (typically 2 hours). The new container starts and opens 10 more connections. After 5 rolling updates, you have 50 phantom connections and MongoDB starts rejecting new ones because you've hit `maxPoolSize`. Your application throws "MongoServerError: connection pool is full" and users see 500 errors.

```
// Without graceful shutdown:
// MongoDB connection pool stays allocated server-side
// TCP FIN/RST never sent - connection is just... gone
// MongoDB waits up to 2 hours before cleaning up the orphan
// Meanwhile, you've eaten through your connection limit
```

In-flight requests return errors to users. A user submits a form. Your app receives the POST request and starts processing it - validating data, writing to the database, sending a confirmation email. Halfway through, Docker kills the container. The user gets a connection reset error. The database write might have completed but the email didn't send. The user doesn't know if their submission worked. They submit again. Now you have duplicate data.

Redis pub/sub subscribers vanish. Your app subscribes to a Redis channel for real-time notifications. Docker kills the container. The subscription is gone, but the publisher doesn't know. Messages published to that channel during the window between the kill and the new container starting are lost forever. Nobody gets notified.

File writes are corrupted. Your app writes to a log file or a data file. It writes 500 bytes of a 1,000-byte record. Docker kills the process. The file now has a partial record. Next time the app starts and reads that file, it either crashes on the corrupt data or silently ignores it.

Memory-mapped state is lost. If your app uses an in-memory cache (like node-cache or a custom Map), everything in that cache disappears instantly. If your code expected to flush that cache to disk or to Redis on shutdown, it never got the chance.

The Complete Graceful Shutdown Pattern

Here's the pattern that handles all of these cases. This isn't optional - it's as fundamental as error handling.

```
const http = require('http');
const mongoose = require('mongoose');
const redis = require('./redis-client');

const app = require('./app'); // Your Express app
const server = http.createServer(app);

// Track whether we're shutting down
let isShuttingDown = false;

// Middleware to reject new requests during shutdown
app.use((req, res, next) => {
  if (isShuttingDown) {
    res.set('Connection', 'close');
    return res.status(503).json({
      error: 'Server is shutting down',
      retryAfter: 5
    });
  }
  next();
});

async function gracefulShutdown(signal) {
  console.log(`[shutdown] Received ${signal}. Starting graceful shutdown...`);
  isShuttingDown = true;

  // 1. Stop accepting new connections
  //    In-flight requests continue to completion
  server.close(async () => {
    console.log('[shutdown] HTTP server closed. No new connections.');

    try {
      // 2. Close database connections properly
      //    This sends TCP FIN to MongoDB, releasing the connection server-side
      await mongoose.connection.close();
      console.log('[shutdown] MongoDB connections closed.');

      // 3. Close Redis connections
      //    Unsubscribes from pub/sub, releases connection
      await redis.quit();
      console.log('[shutdown] Redis connections closed.');

      // 4. Flush any pending writes
      //    If you buffer logs or data, flush them now
      // await logger.flush();
```

```
      // 5. Exit with success code
      //    Exit code 0 tells Docker (and Swarm) this was a clean shutdown
      console.log('[shutdown] Cleanup complete. Exiting.');
      process.exit(0);
    } catch (err) {
      console.error('[shutdown] Error during cleanup:', err);
      process.exit(1);
    }
  });

  // Safety net: if graceful shutdown takes too long, force exit
  // This should be LESS than Docker's stop_grace_period (default 10s)
  // so your app exits cleanly before Docker sends SIGKILL
  setTimeout(() => {
    console.error('[shutdown] Graceful shutdown timed out. Forcing exit.');
    process.exit(1);
  }, 8000); // 8 seconds - leaves 2 seconds before Docker's SIGKILL
}

// Listen for both signals
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));  // Docker sends this
process.on('SIGINT', () => gracefulShutdown('SIGINT'));    // Ctrl+C sends this

// Also handle uncaught exceptions and unhandled rejections
process.on('uncaughtException', (err) => {
  console.error('[fatal] Uncaught exception:', err);
  gracefulShutdown('uncaughtException');
});

process.on('unhandledRejection', (reason) => {
  console.error('[fatal] Unhandled rejection:', reason);
  gracefulShutdown('unhandledRejection');
});

server.listen(3000, '0.0.0.0', () => {
  console.log('Server running on port 3000');
});
```

The Docker Side: Make Signals Work

Your application code is only half the equation. Docker has to actually deliver the signal.

Use exec form in CMD (no shell wrapper):

```
# WRONG - shell form, signals go to /bin/sh, not your app
CMD npm start

# CORRECT - exec form, signals go directly to node
CMD ["node", "server.js"]
```

Use `init: true` in your compose file:

```
services:
  nodeserver:
    image: yourregistry/nodeserver:latest
    init: true  # Runs tini as PID 1, forwards signals to your app

    # Set stop_grace_period to match your shutdown timeout
    stop_grace_period: 15s  # Give your app 15 seconds instead of default 10

    deploy:
      update_config:
        order: start-first      # New container starts before old stops
        failure_action: rollback
      rollback_config:
        parallelism: 1
```

The `init: true` directive runs `tini` as PID 1. Tini is a tiny init system that does exactly two things: forwards signals to your application process, and reaps zombie child processes. It's one line in your compose file and it eliminates an entire class of shutdown bugs.

Set `stop_grace_period` to match your shutdown timeout. If your app needs 15 seconds to drain connections and flush writes, set `stop_grace_period: 15s`. The default is 10 seconds, which might not be enough for applications with long-running requests or large connection pools.

Exit Codes Matter to Swarm

Swarm uses exit codes to decide what to do next:

EXIT CODE	WHAT SWARM DOES	WHAT IT MEANS
0	Considers the task completed successfully	Clean shutdown. If `restart_policy` says "on-failure", Swarm does NOT restart
1	Considers the task failed	Application error. Swarm restarts based on restart policy
137	Considers the task killed (OOMKilled or SIGKILL)	Container ran out of memory or didn't shut down in time
143	Considers the task terminated (SIGTERM received)	Normal during rolling updates - Swarm sent SIGTERM and the app exited

If your application exits with code 0 during a rolling update, Swarm records it as a clean transition. If it exits with code 1 or 137, Swarm may flag the update as failing and trigger a rollback (if you have `rollback_config` set). The difference between `process.exit(0)` and your process being killed without a signal handler is the difference between a smooth rolling update and a rollback that disrupts your users.

A development-time habit that prevents production problems: Stop pressing Ctrl+C to kill your containers. Use `docker compose down` instead. This sends SIGTERM to your containers and gives your shutdown handlers a chance to run, just like production. If you always Ctrl+C, you never test your shutdown code, and the first time it runs is during a production deployment - exactly when you don't want surprises.

```
# Instead of Ctrl+C on docker compose up (foreground):
# Open a new terminal and run:
docker compose down

# Or run detached from the start:
docker compose up -d
# Then stop with:
docker compose down
```

THE DEVELOPER DAILY GRIND: PROBLEMS NOBODY WARNS YOU ABOUT

The sections above cover the big topics - Dockerfiles, compose files, builds, networking, deployment. This section covers the small, daily frustrations that eat 20 minutes here, 30 minutes there, every single day. They're not dramatic enough for blog posts, but they're the reason developers say "Docker slows me down." Each one has a fix.

Attaching a Debugger (Breakpoints Inside Containers)

The number one reason developers resist Docker for local development is that they can't figure out how to hit breakpoints. Their code runs inside a container, but VS Code's debugger connects to `localhost`. How do you bridge the gap?

Step 1 - Expose the debug port and start Node.js in inspect mode:

```
# docker-compose.dev.yml
services:
  nodeserver:
    command: ["node", "--inspect=0.0.0.0:9229", "server.js"]
    ports:
      - "3000:3000"    # Application port
      - "9229:9229"    # Debugger port
```

The `--inspect=0.0.0.0:9229` flag is critical. Without `0.0.0.0`, Node.js only listens for debug connections on `127.0.0.1` *inside the container* - which is unreachable from your host machine. You need it to listen on all interfaces so the port mapping works.

Step 2 - Configure VS Code to attach to the container:

Create `.vscode/launch.json` in your project:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Docker: Attach to Node",
      "type": "node",
      "request": "attach",
      "port": 9229,
      "address": "localhost",
      "localRoot": "${workspaceFolder}/nodeserver",
      "remoteRoot": "/app",
      "restart": true,
      "skipFiles": ["<node_internals>/**"]
    }
  ]
}
```

The `localRoot` / `remoteRoot` mapping is where people get stuck. `localRoot` is the path to your source code on your machine. `remoteRoot` is the path inside the container (whatever you set as `WORKDIR` in your Dockerfile). Without this mapping, VS Code can't match the file in the container to the file on your screen, and breakpoints silently fail.

The `"restart": true` setting automatically reconnects the debugger when the container restarts (like after nodemon detects a file change).

Step 3 - Use it:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d
```

Then in VS Code, press F5 (or Run -> Start Debugging) and select "Docker: Attach to Node." Set breakpoints normally. They work.

For nodemon (hot reload + debugging):

```
command: ["npx", "nodemon", "--inspect=0.0.0.0:9229", "server.js"]
```

Nodemon restarts the process on file changes, and the `--inspect` flag is passed through to each restart. Combined with `"restart": true` in your launch.json, the debugger reconnects automatically after each restart.

Chrome DevTools for Performance and Memory Profiling

VS Code breakpoints are great for stepping through logic. But when your container is eating 800 MB of RAM and you don't know why, or API responses take 3 seconds and you can't figure out where the time goes, you need Chrome DevTools. This is the tool that lets you take heap snapshots, record CPU profiles, track memory allocation over time, and see exactly which functions are burning cycles.

Most developers don't realize Chrome DevTools can connect to *any* Node.js process - not just browser JavaScript. Node's `--inspect` flag speaks the same Chrome DevTools Protocol (CDP) that Chrome's built-in debugger uses. The same port you exposed for VS Code breakpoints (9229) gives you access to the full profiling suite.

Step 1 - Expose the inspect port (same setup as breakpoints):

If you already set up the debugger from the section above, you're most of the way there. Make sure your container exposes port 9229:

```
# docker-compose.dev.yml (or docker-compose.override.yml)
services:
  nodeserver:
    command: ["node", "--inspect=0.0.0.0:9229", "server.js"]
    ports:
      - "3000:3000"
      - "9229:9229"
```

For production-like profiling where you want to investigate under realistic conditions, you can temporarily use `--inspect` on a staging container. But never leave `--inspect` exposed on a production container with port 9229 published - it gives anyone who can reach that port full access to execute arbitrary code in your process.

Step 2 - Connect Chrome DevTools:

1. Open Chrome (or any Chromium-based browser - Edge, Brave, etc.)
2. Navigate to `chrome://inspect`
3. Under "Remote Target," you should see your Node.js process listed (it shows the `--inspect` address and the script name)
4. Click **"inspect"** to open the full DevTools panel

If your container doesn't appear, click "Configure" and add `localhost:9229` to the target discovery list. Docker's port mapping makes the container's debug port appear as localhost on your machine.

If nothing appears at all, the most common causes are:

- You used `--inspect` instead of `--inspect=0.0.0.0:9229` (Node defaults to 127.0.0.1, unreachable from outside the container)
- The port mapping is missing from your compose file
- Another process on your host is already using port 9229 (check with `lsof -i :9229` on Mac/Linux or `netstat -ano | findstr 9229` on Windows)
- Your container hasn't started yet or the process crashed on startup - check `docker compose logs nodeserver`

What you get once connected:

The DevTools panel for a Node.js process looks different from the browser version. You won't see the Elements or Network tabs (there's no DOM). What you get is:

- **Console** - Direct REPL into your running container's Node.js process. You can inspect variables, call functions, check `process.memoryUsage()` live.
- **Sources** - Your application source files, with the ability to set breakpoints (like VS Code, but in the browser).
- **Memory** - Heap snapshots, allocation timelines, allocation sampling. This is the killer feature for containers.
- **Profiler (Performance)** - CPU profiling to find hot functions and slow code paths.

CPU Profiling - Finding Where Time Goes

When your API endpoint takes 2 seconds and you've ruled out database queries (by checking your query logs), the bottleneck is somewhere in your JavaScript. CPU profiling shows you exactly where.

Recording a profile:

1. Go to the **Profiler** tab (some Chrome versions label it **Performance**)
2. Click **Start** (or the record button)

3. Trigger the slow operation - send a request to your API endpoint, run the job, whatever reproduces the slowness
4. Click **Stop**

DevTools shows a flame chart - a visualization of every function call during the recording, stacked by call depth. The x-axis is time. Wide bars are functions that took a long time.

What to look for:

- **Wide bars at the top of the stack** - these are your functions that are taking the most wall-clock time. Drill into them to see what they're calling.
- **Repeated thin bars** - the same function being called thousands of times. Each call is fast, but the aggregate is slow. This often indicates an N+1 loop where you're calling a function per item instead of batching.
- **`JSON.parse` or `JSON.stringify` with wide bars** - you're serializing/deserializing large objects. This is a common hidden cost. A 5 MB JSON response that gets `JSON.stringify()`'d on every request can dominate your CPU time. Consider streaming with `JSON.stringify` alternatives like `fast-json-stringify`, or restructure your data to send smaller payloads.
- **`RegExp` execution** - Catastrophic backtracking in regexes can make a single `test()` call take seconds. If you see regex functions dominating, the regex pattern itself is the problem.
- **Synchronous file operations** - `fs.readFileSync`, `fs.writeFileSync`, or any `*Sync` function blocks the entire event loop. In the profiler, these show as a single wide bar with no children - the thread is blocked waiting for I/O.
- **Garbage collection pauses** - Look for `GC` entries in the flame chart. Frequent or long GC pauses mean you're creating and discarding too many objects. The memory profiling section below helps you find the cause.

Taking action on results:

```
Scenario: Your /api/search endpoint takes 1.8 seconds.
Profile shows:
  - 200ms in your route handler
  - 50ms in MongoDB query
  - 1400ms in JSON.stringify
  - 150ms in Express middleware

The fix isn't optimizing the database query - it's the serialization.
Options:
  - Add pagination to return fewer results
  - Use a streaming JSON serializer
  - Cache the serialized response if the data doesn't change often
  - Return only the fields the client needs (projection)
```

Memory Profiling - Finding Leaks and Bloat

Memory issues in containers are more critical than on a regular server because containers have hard memory limits (`memory: 400M` in your compose file). When a Node.js process exceeds that limit, Docker kills it instantly with SIGKILL (exit code 137) - no graceful shutdown, no error handler, just dead. The process restarts, climbs back up, gets killed again. You see a sawtooth pattern in your monitoring, and your service is unreliable.

Understanding Node.js memory from inside the container:

Before using DevTools, check what your container reports:

```
# Quick memory check from outside the container
docker stats --no-stream nodeserver
# Shows: MEM USAGE / LIMIT     MEM %

# Detailed breakdown from inside the container
docker compose exec nodeserver node -e "
  const mem = process.memoryUsage();
  console.log({
    rss: (mem.rss / 1024 / 1024).toFixed(1) + ' MB',
    heapTotal: (mem.heapTotal / 1024 / 1024).toFixed(1) + ' MB',
    heapUsed: (mem.heapUsed / 1024 / 1024).toFixed(1) + ' MB',
    external: (mem.external / 1024 / 1024).toFixed(1) + ' MB',
    arrayBuffers: (mem.arrayBuffers / 1024 / 1024).toFixed(1) + ' MB'
  });
"
```

What those numbers mean:

METRIC	WHAT IT MEASURES	WHEN TO WORRY
`rss` (Resident Set Size)	Total memory the OS has given to this process	This is what Docker's memory limit counts. If this approaches your container limit, you'll get OOM-killed
`heapTotal`	Memory V8 has allocated for JavaScript objects	If this keeps growing without leveling off, you have a leak
`heapUsed`	Memory V8 is actually using right now	The gap between heapTotal and heapUsed is free heap space
`external`	Memory used by C++ objects tied to JavaScript (Buffers, etc.)	Large values here mean you're holding many Buffers or native objects
`arrayBuffers`	Memory in ArrayBuffer and SharedArrayBuffer	Subset of external - large when processing binary data, files, or streams

Heap Snapshots - The Memory Leak Finder:

A heap snapshot captures every object in your JavaScript heap at a specific moment. By comparing two snapshots, you can see what's accumulating.

1. Open Chrome DevTools -> **Memory** tab
2. Select **"Heap snapshot"**
3. Click **"Take snapshot"** - this is your baseline
4. Use your application normally for a few minutes (or run a load test to accelerate the problem)
5. Take a **second snapshot**
6. In the second snapshot, change the view to **"Comparison"** (dropdown at the top) and select Snapshot 1 as the baseline

The comparison view shows objects that were allocated between snapshot 1 and snapshot 2 and are *still alive* (not garbage collected). Sort by **"Size Delta"** or **"# New"** to find what's accumulating.

Common leak patterns you'll find in the comparison:

Pattern 1 - Event listeners that are never removed:

```
// LEAKS: Every request adds a listener, none are removed
app.get('/stream', (req, res) => {
  const handler = (data) => res.write(data);
  eventEmitter.on('data', handler);
  // If the client disconnects, the handler stays attached forever
});

// FIXED: Clean up on disconnect
app.get('/stream', (req, res) => {
  const handler = (data) => res.write(data);
  eventEmitter.on('data', handler);
  req.on('close', () => eventEmitter.removeListener('data', handler));
});
```

In the heap snapshot comparison, this shows up as growing numbers of `(closure)` objects or the type of objects your handler captures in its closure scope. The heap will show a chain: the EventEmitter holds a reference to the handler function, which holds a reference to the `res` object, which holds a reference to the socket, which holds its buffers. One leaked listener can hold onto kilobytes of associated memory.

Pattern 2 - Caches that grow without bounds:

```
// LEAKS: Cache grows forever
const cache = {};
app.get('/user/:id', async (req, res) => {
  if (!cache[req.params.id]) {
    cache[req.params.id] = await db.users.findById(req.params.id);
  }
  res.json(cache[req.params.id]);
});

// FIXED: Use an LRU cache with a maximum size
const LRU = require('lru-cache');
const cache = new LRU({ max: 500 });  // At most 500 entries
```

In the heap snapshot, this shows as an ever-growing `(object properties)` count on whatever object you're using as the cache. You'll see thousands of string keys and their associated values.

Pattern 3 - Closures capturing more than they should:

```
// LEAKS: The closure captures the entire 'bigData' even though
// only 'bigData.id' is used in the callback
function processItem(bigData) {
  const id = bigData.id;
  setTimeout(() => {
    console.log(`Processing ${bigData.id}`);  // captures entire bigData
  }, 5000);
}

// FIXED: Extract what you need before the closure
function processItem(bigData) {
  const id = bigData.id;
  setTimeout(() => {
    console.log(`Processing ${id}`);  // captures only the string
  }, 5000);
}
```

Pattern 4 - Global arrays or maps that accumulate request data:

```
// LEAKS: Every request pushes to a global array
const requestLog = [];
app.use((req, res, next) => {
  requestLog.push({ url: req.url, time: Date.now(), headers: req.headers });
  next();
});
// After 100,000 requests, requestLog holds ~200 MB of header objects
```

This is obvious in a heap snapshot - you'll see a single huge Array object.

Allocation Timeline - Watching Memory in Real Time:

Instead of comparing two snapshots, the Allocation Timeline records continuously and shows you *when* objects are allocated and whether they persist.

1. Open **Memory** tab -> select **"Allocation instrumentation on timeline"**
2. Click **Start**
3. Exercise your application (send requests, trigger jobs)
4. Click **Stop**

The timeline shows blue bars for objects that were allocated and are still alive, and gray bars for objects that were allocated and then garbage collected. A healthy application has mostly gray bars - objects get created, used, and cleaned up. If you see blue bars that keep accumulating at a steady rate, those are your leaks.

Warning about profiling overhead: Heap snapshots and allocation timelines pause your process while they capture data. A heap snapshot of a 500 MB process can freeze your application for 5-10 seconds. The allocation timeline adds continuous overhead that slows your application by 10-30%. Use these in development or on a staging container - not on a production container serving traffic, unless you've already diverted traffic away from it.

Practical Container Memory Settings

Once you've profiled and optimized, set memory limits that match reality:

```
services:
  nodeserver:
    deploy:
      resources:
        limits:
          memory: 400M     # Hard ceiling - OOM-killed above this
        reservations:
          memory: 200M     # Guaranteed minimum from Swarm
```

How to choose the limit: Run your container under realistic load and watch `docker stats` for 10-15 minutes. Note the peak memory usage. Set your limit to 1.5-2x the peak to give headroom for spikes. If your app peaks at 220 MB under load, `memory: 400M` is reasonable. If it peaks at 350 MB, you need `memory: 512M` or you need to optimize.

Telling Node.js about its memory constraint:

By default, V8 (Node's JavaScript engine) decides how much heap to use based on the system's total memory. Inside a container, V8 might see the *host's* RAM (8 GB, 16 GB, 32 GB) rather than the container's 400 MB limit, depending on your Node.js version and Docker configuration. This means V8 happily grows its heap past the container limit, and Docker kills the process.

```
services:
  nodeserver:
    command: ["node", "--max-old-space-size=300", "server.js"]
    deploy:
      resources:
        limits:
          memory: 400M
```

`--max-old-space-size=300` tells V8 to cap its old generation heap at 300 MB. You set this *lower* than your container's memory limit because RSS includes more than just the V8 heap - it includes the Node.js runtime itself, native addon memory, thread stacks, buffers, and memory-mapped files. A good rule of thumb: set `--max-old-space-size` to about 75% of your container's memory limit.

With this flag, V8 will aggressively garbage-collect as the heap approaches 300 MB instead of growing past the container limit and getting killed. Your application might slow down from more frequent GC, but it stays alive - and that slowdown is a signal to optimize, not a crash you discover at 3 AM.

Combining `--inspect` with `--max-old-space-size` for development profiling:

```
# docker-compose.override.yml - local development with profiling
services:
  nodeserver:
    command: [
      "node",
      "--inspect=0.0.0.0:9229",
      "--max-old-space-size=300",
      "server.js"
    ]
    ports:
      - "3000:3000"
      - "9229:9229"
```

This gives you DevTools profiling *while* your container is constrained to production-like memory limits. You'll see GC pressure and memory behavior that matches production, rather than running with unlimited memory locally and only discovering problems after deployment.

Quick Reference: When to Use What

SYMPTOM	TOOL	WHAT TO DO
API endpoint is slow	CPU Profiler	Record a profile while triggering the endpoint. Look for wide bars in the flame chart
Memory grows over hours/days	Heap Snapshot Comparison	Take snapshot, wait, take another. Compare to find what's accumulating
Memory spikes during specific operations	Allocation Timeline	Record during the operation. Look for blue (persistent) bars
Container keeps getting OOM-killed (exit 137)	`docker stats` + `--max-old-space-size`	Check if V8 is growing past container limit. Set the flag
App is slow but you don't know if it's CPU or I/O	`process.memoryUsage()` + `docker stats`	If CPU is high, use CPU profiler. If memory is high, use heap snapshot. If neither is high, it's I/O wait (database, network, filesystem)
GC pauses causing latency spikes	CPU Profiler + `--trace-gc` flag	Add `--trace-gc` to your node command to log every GC event with timing. Profile to see GC in the flame chart

Hot Reload Stops Working (And You Don't Know Why)

You set up bind mounts, installed nodemon, and hot reload worked perfectly... until it didn't. You change a file, nothing happens. You restart the container, it picks up the changes. What's going on?

Cause 1 (Windows) - Your project files are on the Windows filesystem, not inside WSL.

This is the number one cause of hot reload failures on Windows, and it's also the number one cause of "Docker is slow on Windows." It affects everything: file watching, build speed, npm install performance, and general I/O throughput.

Here's what's happening. Docker Desktop on Windows uses WSL 2 (Windows Subsystem for Linux) as its backend. Your containers run inside a real Linux kernel inside WSL 2. When your project files live on the Windows filesystem (any path starting with `C:\`, `D:\`, or `/mnt/c/`, `/mnt/d/` when accessed from WSL), every file operation has to cross the boundary between the Windows NTFS filesystem and the WSL 2 Linux filesystem. This translation is slow - 5-10x slower for I/O-heavy operations - and **file system event notifications (inotify) don't cross this boundary at all.**

That's why hot reload breaks. Nodemon, webpack, Vite, and every other file watcher rely on inotify events to detect changes. When your files are on the Windows side, those events never fire inside the Linux container. Your file changes. Nothing happens. The watcher never knows.

The fix is not polling. The fix is moving your project into WSL.

```
# WRONG - project on Windows filesystem, accessed through /mnt/c
# Hot reload broken, builds slow, everything slow
cd /mnt/c/Users/yourname/projects/myapp
docker compose up -d
#  File watchers don't work, npm install takes forever

# CORRECT - project on the WSL native filesystem
# Hot reload works, builds are fast, inotify works natively
cd ~/projects/myapp
docker compose up -d
# Yes File watchers work instantly, native Linux I/O speed
```

Your project must live on the WSL filesystem - meaning a path like `~/projects/myapp` or `/home/yourname/projects/myapp` - NOT in `/mnt/c/...`. The `/mnt/c` path is a mount of the Windows filesystem, and it carries all the same cross-boundary penalties.

But I open my project in VS Code on Windows - how do I edit files inside WSL?

This is where people get confused. You don't need to run VS Code inside a Linux terminal. VS Code has a built-in WSL extension that lets you edit files on the WSL filesystem directly from the Windows VS Code interface:

1. Install the "WSL" extension in VS Code (by Microsoft)

2. Open VS Code
3. Press `Ctrl+Shift+P` -> "WSL: Connect to WSL"
4. Open your project folder (which is now inside WSL)
5. VS Code's file explorer, terminal, and extensions all operate inside WSL

When connected this way, VS Code's integrated terminal runs inside WSL, your file edits happen on the WSL filesystem (no cross-boundary translation), and inotify events fire correctly. Docker commands run natively. Everything is fast.

How to tell if you're in WSL mode: Look at the bottom-left corner of VS Code. If it says `WSL: Ubuntu` (or whatever your distro is), you're connected. If it doesn't show anything or says `Local`, you're on the Windows filesystem and Docker will be slow.

Moving an existing project into WSL:

```
# From a WSL terminal (Ubuntu/Debian):
mkdir -p ~/projects
cp -r /mnt/c/Users/yourname/projects/myapp ~/projects/myapp
cd ~/projects/myapp

# Or clone fresh from git (preferred - avoids the node_modules problem entirely
cd ~/projects
git clone https://github.com/yourorg/myapp.git
cd myapp
docker compose up -d
```

Critical: delete `node_modules` and reinstall after moving. If you copied your project from the Windows filesystem instead of cloning fresh, your `node_modules` directory contains packages installed for Windows. Native modules like `bcrypt`, `sharp`, `esbuild`, `better-sqlite3`, and anything with C/C++ bindings were compiled against Windows binaries. Inside WSL, you're now running Linux. Those Windows-compiled native modules will either segfault, throw "invalid ELF header" errors, or fail silently with wrong behavior.

```
# After copying into WSL - ALWAYS do this:
cd ~/projects/myapp
rm -rf node_modules
rm -rf .next .nuxt dist build    # Framework build caches may also have platfor
npm ci                            # Fresh install, now compiled for Linux

# If you're using Docker for everything (which you should be),
# just rebuild the image - npm ci runs inside the container:
docker compose down -v
docker compose up -d --build
```

This applies to any language with platform-specific compiled dependencies - Python's `.so` files, Ruby's native gems, Go binaries, Rust `.so` libraries. If you copied instead of cloning, nuke the dependency directory and reinstall.

After moving, update your VS Code workspace to point to the WSL path. Your git operations, npm commands, and Docker commands all run inside WSL now, and they're all faster.

Cause 2 - macOS file system events don't cross the VM boundary reliably.

Docker on macOS runs containers inside a Linux VM (using Apple's Virtualization.framework). File system events from your Mac sometimes don't propagate through the VirtioFS sharing layer into the Linux VM. This is less severe than the Windows/WSL issue - macOS VirtioFS does forward most events - but it can still cause intermittent missed reloads, especially with large projects or rapid successive edits.

Fix: Enable polling mode as a fallback:

```
services:
  nodeserver:
    environment:
      # For nodemon / chokidar-based watchers
      - CHOKIDAR_USEPOLLING=true
      # For webpack / Next.js
      - WATCHPACK_POLLING=true
      # For Vite
      - VITE_HMR_POLL=true
    volumes:
      - ./nodeserver:/app
      - /app/node_modules
```

Polling is slower than event-based watching (it checks files on a timer instead of reacting to events), but it works reliably. On native Linux, you never need polling - inotify works directly because there's no VM.

Cause 3 - `node_modules` anonymous volume is masking your changes.

If you're using the anonymous volume pattern (`- /app/node_modules`) to protect the container's `node_modules`, but you change `package.json` and run `npm install` on your host, the anonymous volume still has the *old* `node_modules`. The container doesn't see your new dependencies.

Fix: Rebuild when dependencies change:

```
# When you change package.json:
docker compose down
docker compose up -d --build

# Or, to preserve the database volume but rebuild the app:
docker compose up -d --build nodeserver
```

Cause 4 - Your framework ignores certain directories.

Many frameworks have a `watchIgnore` or `ignored` setting that excludes directories from the watcher. Check your `nodemon.json`, `webpack.config.js`, or `next.config.js` for patterns that might accidentally exclude the directory you're editing.

The `.env` File Precedence Nightmare

You change a variable in your `.env` file, restart the container, and the old value is still there. Or you set a variable in the compose file AND the `.env` file and get confused about which one wins. This is the environment variable precedence order in Docker Compose, from highest to lowest priority:

```
1. Shell environment variable (export VAR=value before running docker compose)
2. Value in docker-compose.yml under environment:
3. Value from env_file: reference
4. Value in .env file (auto-loaded by Docker Compose)
5. ENV instruction in Dockerfile
```

The common trap: You set `POSTGRES_DB=myapp` in your `.env` file, but you also have `POSTGRES_DB=production_db` set as a shell environment variable from some previous debugging session. Docker Compose uses the shell variable, not the `.env` file. Your container connects to the wrong database and you spend an hour checking your compose file.

How to debug it:

```
# See what Docker Compose actually resolves for all variables
docker compose config

# This renders the FINAL compose file with all variables substituted
# If a variable shows the wrong value, you know it's being overridden
# somewhere higher in the precedence chain

# Check your shell for leftover environment variables
env | grep POSTGRES
env | grep NODE

# See what a running container actually has
docker compose exec nodeserver env | sort
```

`docker compose config` is the single most useful command for debugging environment variables. It shows you exactly what Docker Compose will send to your containers, with all variable substitution resolved. If the value looks wrong there, the problem is in your precedence chain. If it looks right there but wrong inside the container, the problem is in your Dockerfile or entrypoint script overriding it.

The `.env.example` pattern - commit a `.env.example` to your repo with all required variables (using placeholder values), and add `.env` to `.gitignore`. New team members copy the example, fill in their values, and they're up and running. Without this, onboarding is "ask someone for the secret values on Slack":

```
# .env.example (committed to git)
POSTGRES_USER=myapp
POSTGRES_PASSWORD=change_me_locally
POSTGRES_DB=myapp_dev
NODE_ENV=development
REDIS_URL=redis://redis:6379
JWT_SECRET=any_random_string_for_dev

# .env (in .gitignore, never committed)
POSTGRES_USER=myapp
POSTGRES_PASSWORD=my_actual_local_password
POSTGRES_DB=myapp_dev
NODE_ENV=development
REDIS_URL=redis://redis:6379
JWT_SECRET=dev_secret_key_12345
```

Database Initialization, Migrations, and the Stale Volume Trap

You spin up a fresh Postgres or MySQL container. It creates the default database from the `POSTGRES_DB` environment variable. Great. But then you need to run migrations to create your tables. And seed data for development. And when you change the schema, you need to run migrations again. And sometimes you need to blow everything away and start fresh.

The startup race condition: Your app container starts and immediately tries to connect to the database. The database container is running, but it's still initializing - creating users, setting up the database, running init scripts. Your app gets "connection refused" and crashes.

Fix with health checks (the only reliable solution):

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: myapp
      POSTGRES_PASSWORD: devpassword
      POSTGRES_DB: myapp_dev
    volumes:
      - postgres-data:/var/lib/postgresql/data
      - ./db/init:/docker-entrypoint-initdb.d  # SQL files run on FIRST start
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U myapp -d myapp_dev"]
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 10s

  nodeserver:
    depends_on:
      db:
        condition: service_healthy  # Waits for health check to pass
    # ...
```

The `condition: service_healthy` makes Docker Compose wait until the database is actually accepting connections before starting your app.

The stale volume trap: You change `POSTGRES_DB` from `myapp_dev` to `myapp_v2`. You restart. But Postgres ignores the change because the data directory already exists in the volume. Postgres init scripts (and `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD`) only run when the data directory is empty - meaning the first time the container starts with a fresh volume.

```
# Your .env change won't take effect until you destroy the volume:
docker compose down -v    # -v removes named volumes
docker compose up -d      # Fresh start, init scripts run again
```

This applies to any database that uses a volume: Postgres, MySQL, MongoDB. If you change initialization parameters, you must delete the volume. Otherwise the container starts, sees existing data, and skips initialization entirely.

Running migrations as part of your development workflow:

```
services:
  migrate:
    image: yourregistry/nodeserver:latest
    build: ./nodeserver
    command: ["npm", "run", "migrate"]
    depends_on:
      db:
        condition: service_healthy
    restart: "no"  # Run once and stop
    networks:
      - app-network

  nodeserver:
    depends_on:
      migrate:
        condition: service_completed_successfully
      db:
        condition: service_healthy
    # ...
```

The `condition: service_completed_successfully` makes the app wait not just for the database to be healthy, but for migrations to finish. The `restart: "no"` ensures the migration container runs once and exits.

Logs Are Overwhelming (And You Can't Find Anything)

When you run 4-5 services in Docker Compose, the log output becomes a wall of text. Health checks fire every 30 seconds, database query logs scroll by, and the one error message you're looking for is buried somewhere in the middle.

Disable health check logging in development:

```
# Use a no-op health check in development
services:
  nodeserver:
    healthcheck:
      test: /bin/true    # Always passes, produces no log output
      interval: 60s      # Check rarely in development
```

Or use the environment variable pattern from Nick Janetakis - define the health check command as a variable so you can swap between a real check in production and `/bin/true` in development.

Follow logs for one service only:

```
# Instead of seeing all services:
docker compose logs -f

# Follow only what you care about:
docker compose logs -f nodeserver

# Last 50 lines with timestamps:
docker compose logs --tail 50 -t nodeserver

# Multiple specific services:
docker compose logs -f nodeserver redis
```

Add log levels to your application and control them via environment variables:

```
const LOG_LEVEL = process.env.LOG_LEVEL || 'info';

function log(level, ...args) {
  const levels = { debug: 0, info: 1, warn: 2, error: 3 };
  if (levels[level] >= levels[LOG_LEVEL]) {
    console.log(`[${new Date().toISOString()}] [${level.toUpperCase()}]`, ...ar
  }
}
```

```
# In your dev compose override:
environment:
  - LOG_LEVEL=debug    # Verbose for development
```

Changes Not Taking Effect (The Stale Container Problem)

You change an environment variable. You change a Dockerfile instruction. You install a new npm package. You restart with `docker compose up -d`. Nothing changes. The old behavior persists. You add more logging. Still old. You question your sanity.

This is the stale container problem, and it's the most common daily frustration with Docker Compose. Here's why it happens and how to fix each case:

Environment variable change not taking effect:

`docker compose up -d` reuses existing containers if the compose configuration hasn't changed. But Docker Compose doesn't always detect `.env` file changes as a configuration change.

```
# Force recreation of containers (even if config seems unchanged):
docker compose up -d --force-recreate

# Verify what the container actually has:
docker compose exec nodeserver env | grep YOUR_VARIABLE
```

Dockerfile change not taking effect:

`docker compose up -d` doesn't rebuild images automatically. If you changed a `RUN`, `COPY`, or `ENV` instruction in your Dockerfile, you need to explicitly rebuild:

```
# Rebuild AND recreate:
docker compose up -d --build

# Rebuild without cache (when cache is stale):
docker compose build --no-cache nodeserver
docker compose up -d
```

New npm package not appearing:

If you use the anonymous volume pattern for `node_modules`, the volume persists across container recreations. It still has the old `node_modules` even after a rebuild - because Docker restores the anonymous volume from the previous container.

```
# Nuclear option - remove anonymous volumes and rebuild:
docker compose down -v
docker compose up -d --build

# Less destructive - remove just the anonymous volume:
docker compose rm -f nodeserver
docker volume ls | grep node_modules  # Find the volume name
docker volume rm <volume_name>
docker compose up -d --build nodeserver
```

The diagnostic checklist when "changes aren't taking effect":

```
# 1. Is the compose file correct?
docker compose config | grep -A5 nodeserver

# 2. Is the container actually using the new image?
docker compose ps    # Check if container was recreated
docker inspect <container> | grep Image  # Check image hash

# 3. Are environment variables correct inside the container?
docker compose exec nodeserver env | sort

# 4. Are the right files inside the container?
docker compose exec nodeserver ls -la /app/
docker compose exec nodeserver cat /app/package.json

# 5. Is the bind mount working?
docker compose exec nodeserver cat /app/server.js | head -5
# Does it match your local file?
```

The `docker-compose.override.yml` Shortcut

Docker Compose automatically loads `docker-compose.override.yml` if it exists in the same directory. You don't need `-f` flags. This means you can have:

```
docker-compose.yml          # Base config (committed to git)
docker-compose.override.yml  # Dev overrides (in .gitignore)
```

When you run `docker compose up`, it automatically merges both files. Your base file has the production-ready config, and your override has your personal development tweaks: bind mounts, debug ports, verbose logging, local-only services.

```
# docker-compose.override.yml (in .gitignore - personal dev config)
services:
  nodeserver:
    build: ./nodeserver
    volumes:
      - ./nodeserver:/app
      - /app/node_modules
    command: ["npx", "nodemon", "--inspect=0.0.0.0:9229", "server.js"]
    ports:
      - "3000:3000"
      - "9229:9229"
    environment:
      - NODE_ENV=development
      - LOG_LEVEL=debug
      - CHOKIDAR_USEPOLLING=true

  # Add a tool that only exists in your dev setup
  mailhog:
    image: mailhog/mailhog
    ports:
      - "8025:8025"  # Web UI for catching emails
```

The benefit: everyone on the team has their own override file with their own preferences (ports, debug settings, extra services), and it never causes merge conflicts because it's gitignored. New team members copy a `docker-compose.override.example.yml` (committed to git) and customize it.

Warning: If you use `-f` flags explicitly (`docker compose -f docker-compose.yml -f docker-compose.prod.yml`), the automatic override loading is disabled. It only works when you run plain `docker compose up` without `-f`.

Docker is messy. Every build leaves behind intermediate layers. Every `docker compose down` leaves dangling volumes. Every failed build leaves images. Over weeks of development, you accumulate gigabytes of orphaned data.

Build a cleanup habit. I run this weekly:

```
# See what's using disk space
docker system df

# OUTPUT:
# TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
# Images          45        5         12.3GB    10.1GB (82%)
# Containers      12        3         150MB     120MB (80%)
# Local Volumes   8         3         5.2GB     3.8GB (73%)
# Build Cache     -         -         2.1GB     2.1GB

# Safe cleanup - removes stopped containers, dangling images, unused networks
docker system prune

# Aggressive cleanup - also removes all unused images
docker system prune -a

# Remove unused volumes (CAREFUL - check what's in them first)
docker volume ls
docker volume prune
```

Before running `docker volume prune`, check what volumes exist and what's in them. Development database volumes contain your test data. If you don't care about it, prune away. If you do, leave them alone.

CONTAINER USERS, PERMISSIONS, AND WHY ROOT IS THE WRONG DEFAULT

Every Docker container runs as root by default. This is one of Docker's worst defaults and most developers never change it. Your Node.js app, your Express server, your background workers - all running as UID 0 with full superuser privileges inside the container.

Why This Matters More Than You Think

"But it's inside a container, it's isolated." Partially true. Container isolation is not perfect. Container escape vulnerabilities are discovered regularly (CVE-2024-21626 in runc, CVE-2022-0847 "Dirty Pipe" in the kernel, CVE-2019-5736 in runc). When

one of these is exploited, the attacker gets whatever privilege level the container process had. If your process was root in the container, the attacker potentially has root on the host.

Even without escape vulnerabilities, running as root has real consequences:

- If your application has a path traversal bug, root can read `/etc/shadow` inside the container
- If you bind-mount a host directory, root in the container can modify or delete files on the host
- If a dependency has a remote code execution vulnerability (and with hundreds of npm packages, the odds are non-zero), the attacker's shell is root
- Container orchestrators like Kubernetes can enforce PodSecurityPolicies or SecurityContexts that reject root containers entirely. If you haven't set up a non-root user, your images can't run in those clusters

The Node.js Built-in User

The official `node:20-bookworm-slim` image already ships with a non-root user called `node` (UID 1000, GID 1000). You don't need to create one from scratch. Most developers don't know this user exists.

```
FROM node:20-bookworm-slim

WORKDIR /app

# Copy package files and install dependencies as root
# (root is needed for npm ci because it might need to write to system directori
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Copy application source with correct ownership
COPY --chown=node:node . .

# Switch to the non-root user for the rest
USER node

EXPOSE 3000
ENTRYPOINT ["node", "server.js"]
```

The key ordering here matters: install dependencies as root (because `npm ci` may need permissions for native module compilation), then `COPY --chown=node:node` your source code so the files are owned by the non-root user, then `USER node` to switch before the ENTRYPOINT.

When You Need a Custom User

If you're not using the `node` base image, or if you need a specific UID to match your host system (common on Linux where Docker doesn't remap UIDs), create your own:

```
FROM ubuntu:22.04

# Create group and user with explicit IDs
RUN groupadd --gid 1000 appgroup && \
    useradd --uid 1000 --gid appgroup --shell /bin/sh --create-home appuser

WORKDIR /app

# Install dependencies as root
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Change ownership of the app directory
RUN chown -R appuser:appgroup /app

# Copy source with correct ownership
COPY --chown=appuser:appgroup . .

USER appuser

ENTRYPOINT ["node", "server.js"]
```

The --chown Flag on COPY (The Part Everyone Misses)

Without `--chown`, `COPY` creates files owned by root. Your app runs as `node` or `appuser` but the files it needs to read belong to root. It usually works because the default file mode allows read access, but the moment your app needs to write (logs, uploads, temp files, SQLite databases), it fails with `EACCES: permission denied`.

```
# BAD - files owned by root, app runs as node
COPY . .
USER node
# App can READ its own source files but can't WRITE anywhere

# GOOD - files owned by the user that will run them
COPY --chown=node:node . .
USER node
# App owns its files and can write where needed
```

`--chown` works on both `COPY` and `ADD` instructions. Always use it for your application source code when running as a non-root user.

The Volume Permission Problem (Linux-Specific)

On macOS and Windows, Docker Desktop remaps UIDs transparently, so this usually isn't an issue. On Linux, container UIDs are the same as host UIDs - there's no remapping. This creates problems with volumes:

```
services:
  nodeserver:
    user: "1000:1000"    # Runs as UID 1000
    volumes:
      - app-data:/app/data    # Docker creates this volume as root
```

The named volume `app-data` is created with root ownership. Your app runs as UID 1000. It can't write to its own data directory. The fix is an entrypoint script that adjusts permissions before dropping to the non-root user:

```
COPY --chown=root:root entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh

# Note: ENTRYPOINT runs as whatever USER is set to.
# We use the entrypoint script to fix permissions then exec as the app user.
ENTRYPOINT ["/entrypoint.sh"]
```

```
#!/bin/sh
# entrypoint.sh
# Fix volume permissions (only needed on first run or when volumes are recreate
chown -R appuser:appgroup /app/data 2>/dev/null || true

# Drop to non-root and exec the main process
exec su-exec appuser node server.js
```

For Alpine-based images, use `su-exec`. For Debian-based images, use `gosu`. Both do the same thing - run a command as a different user without the overhead of `su` (which starts a new shell session).

When Root IS Required

There are a few legitimate cases where you need root inside the container:

- **Installing system packages** - `apt-get install`, `apk add` require root. Do this early in the Dockerfile before `USER`.
- **Binding to ports below 1024** - Port 80 and 443 require root on Linux. Instead of running your entire app as root, put NGINX or Caddy in a separate container as the reverse proxy, and have your app listen on port 3000+ as a non-root user.
- **Modifying `/etc/` files** - Timezone changes, host entries, etc. Do this in the Dockerfile build steps, not at runtime.
- **Some monitoring/debugging containers** - Tools like `tcpdump`, `netshoot`, `strace` need elevated permissions. These should be temporary diagnostic containers, not your application.

Quick Pattern for Multi-Stage with Non-Root User

```
# Stage 1: Build as root (needs compiler tools)
FROM node:20-bookworm-slim AS builder
WORKDIR /app
RUN apt-get update && apt-get install -y python3 make g++ && \
    apt-get clean && rm -rf /var/lib/apt/lists/*
COPY package.json package-lock.json ./
RUN npm ci --omit=dev

# Stage 2: Production as non-root
FROM node:20-bookworm-slim
WORKDIR /app
ENV NODE_ENV=production

COPY --from=builder --chown=node:node /app/node_modules ./node_modules
COPY --chown=node:node . .

USER node
EXPOSE 3000
ENTRYPOINT ["node", "server.js"]
```

The builder stage runs as root because it needs `apt-get` and the compilation toolchain. The production stage switches to `node` immediately. The `--chown=node:node` on `COPY --from=builder` ensures even the compiled `node_modules` from the builder stage are owned by the correct user.

SECRETS: WHY ENVIRONMENT VARIABLES ARE NOT SECRET

This section is specifically about handling sensitive data - database passwords, API keys, TLS certificates, encryption keys - in a Docker environment. Most developers put these values in environment variables and assume they're safe. They're not.

The Problem with Environment Variables for Secrets

Environment variables are the most common way to pass configuration to containers. For non-sensitive data like `NODE_ENV=production` or `PORT=3000`, they're fine. For secrets, they have serious problems:

1. Environment variables are visible in process listings.

Anyone with access to the Docker host can see every environment variable in every running container:

```
# Anyone with docker access can see your secrets
docker inspect mycontainer --format '{{json .Config.Env}}'
# Output: ["MONGO_URI=mongodb://admin:s3cretPassw0rd@mongo:27017/mydb","API_KEY

# Or just exec into the container
docker exec mycontainer env
# Prints every environment variable including all your secrets
```

In Swarm, `docker service inspect` shows the environment variables of every running service. If your ops team member, your CI/CD logs, or your monitoring dashboard can see `docker inspect` output, they can see your database password.

2. Environment variables leak into child processes.

Every child process your application spawns inherits the full environment. If your Node.js app runs a shell command (`child_process.exec()`), the child process gets every secret. If a dependency spawns a subprocess for any reason, that subprocess sees every secret. If that subprocess crashes and generates a core dump, the secrets are in the dump.

3. Environment variables end up in logs.

It's extremely common for frameworks, error handlers, and crash reporters to log the process environment. Express error handlers, Sentry, PM2, and many npm packages dump `process.env` when reporting errors. Your database password is now in your Elasticsearch cluster, your CloudWatch logs, or your Sentry dashboard.

4. Environment variables persist in image layers.

If you set a secret with `ENV` in your Dockerfile:

```
# NEVER DO THIS
ENV DATABASE_PASSWORD=s3cretPassw0rd
```

That value is baked into the image layer permanently. Anyone who pulls your image can extract it. Even if you later delete the environment variable in a subsequent layer, the earlier layer still contains it. Docker images are additive - you can't truly remove something from a previous layer.

5. Environment variables are visible in your compose file.

```
# NEVER DO THIS in a file committed to git
services:
  nodeserver:
    environment:
      - DATABASE_PASSWORD=s3cretPassw0rd
```

Even if you use `.env` files (which are better), the values are stored in plaintext on disk, and `docker compose config` will print them to stdout for anyone watching.

Docker Secrets: The Correct Approach

Docker Swarm has a built-in secrets management system. Secrets are encrypted at rest in the Swarm's Raft log, encrypted in transit between nodes, and mounted as in-memory files (tmpfs) inside containers - they never touch the container's filesystem. When the container stops, the secrets are wiped from memory.

Creating secrets:

```
# From a string (the hyphen reads from stdin)
echo "s3cretPassw0rd" | docker secret create db_password -

# CAREFUL: echo adds a trailing newline character to your secret.
# This silently breaks passwords. Use printf instead:
printf "s3cretPassw0rd" | docker secret create db_password -

# From a file (for certificates, multi-line config, etc.)
docker secret create tls_cert ./cert.pem
docker secret create tls_key ./private-key.pem

# From stdin without leaving the value in shell history:
# Type the value and press Ctrl+D when done
cat /dev/stdin | docker secret create api_key -

# List all secrets (values are NEVER shown)
docker secret ls

# Inspect metadata only (the value is never revealed)
docker secret inspect db_password
```

The newline trap: `echo "password"` outputs `password\n` (with a trailing newline). Your secret is now `"password\n"`, not `"password"`. When your app reads the file and does `.trim()`, it works fine. But if another service reads the raw bytes (like a database image using `_FILE` convention), authentication fails with "invalid password" and you'll spend hours debugging. Always use `printf` or `echo -n` for secret creation.

You cannot retrieve the value of a secret once it's created. This is intentional. If you need to see what a secret contains, you have to read it from inside a running container that has it mounted.

Using secrets in your compose file:

```
version: "3.8"

services:
  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    secrets:
      - db_password
      - api_key
      - tls_cert
    # NO environment variables for secrets

secrets:
  db_password:
    external: true    # Already created via docker secret create
  api_key:
    external: true
  tls_cert:
    external: true
```

Reading secrets in your application:

Inside the container, secrets are mounted as files at `/run/secrets/`. Your application reads them as files, not environment variables:

```
const fs = require('fs');
const path = require('path');

// Helper function to read a Docker secret
function getSecret(secretName) {
  const secretPath = path.join('/run/secrets', secretName);
  try {
    return fs.readFileSync(secretPath, 'utf8').trim();
  } catch (err) {
    // Fall back to environment variable for local development
    // (where you're not running in Swarm)
    const envName = secretName.toUpperCase();
    if (process.env[envName]) {
      return process.env[envName];
    }
    throw new Error(
      `Secret "${secretName}" not found at ${secretPath} ` +
      `and no ${envName} environment variable set`
    );
  }
}

// Usage
const dbPassword = getSecret('db_password');
const apiKey = getSecret('api_key');
const mongoUri = `mongodb://admin:${dbPassword}@mongo:27017/mydb`;
```

This pattern gives you the best of both worlds: in Swarm, secrets are read from encrypted, in-memory files. In local development (where you might not have Swarm secrets), it falls back to environment variables. The key is that production never exposes the secret as an environment variable.

Why File-Based Secrets Are Fundamentally More Secure

The file-based approach isn't just a preference - it's architecturally different:

- **Files have access controls.** You can set the file mode to `0400` (owner read only). Environment variables have no access controls at all - any process in the container can read them.
- **Files aren't inherited by child processes.** A child process spawned by your app doesn't automatically get access to `/run/secrets/db_password` unless you explicitly pass the path or file descriptor.

- **Files don't show up in `docker inspect`.** The secret's value is never part of the container's configuration metadata. Only the secret name appears.
- **Files aren't logged by error handlers.** No framework dumps the contents of `/run/secrets/` when an error occurs. But nearly every crash reporter includes `process.env` in error reports.
- **tmpfs-mounted files are never written to disk.** In Swarm, `/run/secrets` is a tmpfs mount - it exists only in RAM. When the container stops, the data is gone. Environment variables, on the other hand, can persist in container metadata on the Docker host.

Connecting Your Database with Secrets (Complete Pattern)

Here's how a real application connects to MongoDB using file-based secrets with zero secrets in environment variables:

```
# docker-compose.production.yml
version: "3.8"

services:
  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    environment:
      - NODE_ENV=production
      - MONGO_HOST=mongo
      - MONGO_PORT=27017
      - MONGO_DB=myapp
      # Notice: NO password in environment variables.
      # The only env vars here are non-sensitive connection metadata.
    secrets:
      - mongo_user
      - mongo_password
    deploy:
      replicas: 3

  mongo:
    image: mongo:7
    secrets:
      - mongo_user
      - mongo_password
    # MongoDB also reads secrets from files using environment variables
    # that point to the file paths
    environment:
      - MONGO_INITDB_ROOT_USERNAME_FILE=/run/secrets/mongo_user
      - MONGO_INITDB_ROOT_PASSWORD_FILE=/run/secrets/mongo_password

secrets:
  mongo_user:
    external: true
  mongo_password:
    external: true
```

Notice the `_FILE` suffix pattern on the MongoDB environment variables. Many official Docker images (MongoDB, PostgreSQL, MySQL, Redis) support this convention: instead of `MONGO_INITDB_ROOT_PASSWORD=thepassword`, you use `MONGO_INITDB_ROOT_PASSWORD_FILE=/run/secrets/mongo_password` and the image reads the file contents at startup. This means even the database container never has the password as an environment variable.

Your Node.js application:

```
const fs = require('fs');
const { MongoClient } = require('mongodb');

function readSecret(name) {
  try {
    return fs.readFileSync(`/run/secrets/${name}`, 'utf8').trim();
  } catch {
    return process.env[name.toUpperCase()]; // local dev fallback
  }
}

const user = readSecret('mongo_user');
const password = readSecret('mongo_password');
const host = process.env.MONGO_HOST || 'mongo';
const port = process.env.MONGO_PORT || '27017';
const db = process.env.MONGO_DB || 'myapp';

const uri = `mongodb://${user}:${password}@${host}:${port}/${db}`;
const client = new MongoClient(uri);
```

Simulating Secrets in Local Development

In local development you're probably running `docker compose up`, not `docker stack deploy`. Compose doesn't have Swarm's encrypted secret storage, but it can mount files as secrets:

```
# docker-compose.yml (local development)
services:
  nodeserver:
    build: .
    secrets:
      - db_password

secrets:
  db_password:
    file: ./secrets/db_password.txt    # Plain text file on your machine
```

Create a `./secrets/` directory, add it to `.gitignore`, and put your development secrets there as plain text files (one value per file). The mounting path inside the container is identical (`/run/secrets/db_password`), so your application code works the same way in development and production.

```
# Set up local secrets directory
mkdir -p secrets
echo "dev_password_123" > secrets/db_password.txt
echo "dev_api_key_abc" > secrets/api_key.txt
echo "secrets/" >> .gitignore
```

Rotating Secrets Without Downtime

Secrets in Swarm are immutable - you can't update a secret's value. Instead, you create a new version and update the service:

```
# Create new version of the secret
echo "newS3cretPassw0rd" | docker secret create db_password_v2 -

# Update the service: remove old secret, add new one with the same target name
docker service update \
  --secret-rm db_password \
  --secret-add source=db_password_v2,target=db_password \
  mystack_nodeserver
```

The `target=db_password` part is important - it means the new secret still appears as `/run/secrets/db_password` inside the container, so your application code doesn't change. Swarm performs a rolling update, so there's no downtime. Old containers read the old secret, new containers read the new one.

DOCKER CONFIGS: NON-SENSITIVE CONFIGURATION IN SWARM

Docker Configs are the non-sensitive sibling of Docker Secrets. While secrets are encrypted and meant for passwords and keys, configs are for configuration files that need to be distributed across your Swarm but aren't confidential - things like NGINX configs, application properties files, feature flags, or logging configuration.

Why Use Configs Instead of Bind Mounts or Environment Variables?

Bind mounts don't work in Swarm. If your NGINX config is at `/home/deploy/nginx.conf` on Node 1, Swarm might schedule the container on Node 2 where that file doesn't exist. You'd have to copy the config file to every node manually and keep them in sync.

Environment variables have line limits and format restrictions. You can't easily pass a multi-line NGINX config or a JSON configuration file as an environment variable.

Baking configs into the image means rebuilding to change them. If your NGINX `proxy_pass` target changes, you don't want to rebuild and redeploy the entire NGINX image.

Docker Configs solve all three: they're stored in the Swarm raft store, available on every node, and mounted as files inside the container.

Creating and Using Configs

```
# Create a config from a file
docker config create nginx_config ./nginx.conf

# Create from stdin
echo '{"feature_flags": {"new_ui": true}}' | docker config create app_flags -

# List configs
docker config ls

# Unlike secrets, you CAN inspect config values
docker config inspect nginx_config --pretty
```

Using configs in your compose file:

```
version: "3.8"

services:
  nginx:
    image: nginx:stable-alpine
    configs:
      - source: nginx_config
        target: /etc/nginx/conf.d/default.conf
        mode: 0444    # Read-only for everyone
    ports:
      - "80:80"
      - "443:443"

  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    configs:
      - source: app_flags
        target: /app/config/flags.json
        uid: "1000"    # Owned by the node user
        gid: "1000"
        mode: 0440

configs:
  nginx_config:
    external: true
  app_flags:
    external: true
```

Your application reads the config as a regular file:

```
const flags = JSON.parse(fs.readFileSync('/app/config/flags.json', 'utf8'));
if (flags.feature_flags.new_ui) {
  // serve new UI
}
```

Updating Configs

Like secrets, configs are immutable. Create a new version and update the service:

```
# Edit your nginx.conf locally, then:
docker config create nginx_config_v2 ./nginx.conf

docker service update \
  --config-rm nginx_config \
  --config-add source=nginx_config_v2,target=/etc/nginx/conf.d/default.conf \
  mystack_nginx
```

Swarm rolling-updates the containers with the new config. No downtime.

Configs vs Secrets vs Environment Variables - When to Use Each

DATA TYPE	MECHANISM	EXAMPLE
Non-sensitive simple values	Environment variables	`NODE_ENV=production`, `PORT=3000`, `LOG_LEVEL=info`
Non-sensitive files	Docker Configs	nginx.conf, feature flags JSON, logging config
Sensitive values	Docker Secrets	Database passwords, API keys, TLS certificates, encryption keys
Sensitive files	Docker Secrets	Private keys, service account JSON files, `.pem` certificates

The rule is simple: if you wouldn't want it appearing in a log file, use a secret. If it's fine to be seen but needs to be a file (not a simple key-value), use a config. If it's a simple key-value that's not sensitive, use an environment variable.

LOGGING DRIVERS AND LOG MANAGEMENT

By default, Docker captures everything your application writes to stdout and stderr, stores it as JSON on disk, and lets it grow until your disk is full. Most developers don't know this is happening until they run `docker system df` and find 20 GB of log files, or their production server's disk fills up at 3 AM.

The Default: json-file Driver

Docker's default logging driver is `json-file`. Every line your container writes to stdout or stderr becomes a JSON entry in a file at `/var/lib/docker/containers//-json.log` on the host.

The problem: **there is no size limit by default.** A busy Node.js application that logs every request can generate gigabytes of log data in days. A verbose error loop can fill a disk in hours.

Configure Log Rotation (Do This Immediately)

Add log rotation limits either globally (for all containers) or per container.

Globally - edit the Docker daemon config:

On Docker Desktop: Settings -> Docker Engine, add to the JSON:

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

This limits each container's log to 3 files of 10 MB each - maximum 30 MB per container. When the current file reaches 10 MB, Docker rotates to a new file. When there are 3 files, the oldest gets deleted.

Per container in your compose file:

```
services:
  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

If you don't set these, a single container can consume your entire disk. This has brought down more production servers than most people realize.

Available Logging Drivers

Docker supports multiple logging drivers beyond `json-file`:

DRIVER	WHERE LOGS GO	WHEN TO USE
`json-file`	Local JSON files	Default. Good for development and small deployments
`journald`	systemd journal	Linux servers already using systemd for log management
`syslog`	syslog daemon	Integration with existing syslog infrastructure
`fluentd`	Fluentd collector	Centralized logging with Fluentd/Fluent Bit pipelines
`gelf`	Graylog/Logstash	GELF-compatible log aggregators
`awslogs`	AWS CloudWatch	AWS deployments
`splunk`	Splunk	Enterprise Splunk infrastructure
`none`	Nowhere	Intentionally discard logs (health check containers, etc.)

Important: `docker compose logs` and `docker logs` only work with the `json-file` and `journald` drivers. If you switch to `fluentd` or `awslogs`, running `docker logs mycontainer` returns an error. Docker 20.10+ has "dual logging" that keeps a local cache for `docker logs` regardless of driver, but check your version.

Setting the Driver Per Service

```
services:
  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "5"

  # Health check sidecar - we don't need its logs
  healthchecker:
    image: curlimages/curl
    logging:
      driver: none
```

What Your Application Should Log

Docker captures stdout and stderr. That's it. If your application writes logs to a file inside the container (like `app.log`), Docker's logging driver never sees it. Those file-based logs die with the container unless you mount a volume.

The best practice for containerized applications is: **write everything to stdout/stderr and let Docker's logging infrastructure handle the rest.** This means:

```
// GOOD - Docker captures this
console.log('Server started on port 3000');
console.error('Database connection failed:', err.message);

// GOOD - structured logging to stdout
const pino = require('pino');
const logger = pino({ level: process.env.LOG_LEVEL || 'info' });
logger.info({ port: 3000 }, 'Server started');
logger.error({ err }, 'Database connection failed');

// BAD - Docker never sees this, lost when container dies
const fs = require('fs');
fs.appendFileSync('/app/logs/app.log', 'Server started\n');
```

If you're using a logging library like Winston, Pino, or Bunyan, configure it to output to stdout, not to files. In Docker, stdout IS your log transport.

Structured Logging (JSON) Is Worth the Effort

When your logs are plain text, searching through them with `docker logs mycontainer | grep "error"` works for small volumes. When you have 6 replicas producing thousands of lines per minute, you need structure.

```
// Unstructured - hard to parse, hard to filter
console.log('2025-01-15 User login failed for user john@example.com from 203.0.

// Structured JSON - every field is queryable
logger.warn({
  event: 'login_failed',
  user: 'john@example.com',
  ip: '203.0.113.50',
  attempts: 3
}, 'Login failed');
// Output: {"level":40,"time":1705315200000,"event":"login_failed","user":"johr
```

When structured JSON logs flow into a log aggregator (ELK stack, Grafana Loki, CloudWatch), you can query `event:login_failed AND attempts:>5` instead of writing regex patterns against unstructured text. Set this up from the start - retrofitting structured logging into an existing app is painful.

THE `NETSHOOT` CONTAINER: NETWORK DEBUGGING WITHOUT INSTALLING ANYTHING

When you're debugging container networking issues - DNS resolution failures, connection timeouts between services, mysterious packet drops - you need tools like `dig`, `nslookup`, `tcpdump`, `traceroute`, `curl`, `iperf`, and `netstat`. Your production containers don't have any of these (and they shouldn't - lean images are the goal). Installing them into a running production container means running `apt-get update` on a production node, which is both slow and a bad practice.

The `nicolaka/netshoot` image, created by Nicola Kabar (a former Docker engineer), is a pre-built troubleshooting container with every networking tool you'd ever need. You spin it up, debug, and throw it away. Nothing gets installed on your production nodes or your application containers.

Running netshoot on the Same Network as Your Application

```
# Attach to the same Docker network as your application
docker run -it --rm --network myapp_app-network nicolaka/netshoot

# Now you can:
# Test DNS resolution
dig mongo
nslookup nodeserver

# Test connectivity to a service
curl -v http://nodeserver:3000/health

# See all containers on this network
arp-scan --localnet

# Trace the route to a service
traceroute mongo

# Watch live traffic (useful for debugging proxy headers, missing requests, etc
tcpdump -i any -A port 3000

# Test bandwidth between containers
iperf -c nodeserver -p 5001
```

Running netshoot in a Service's Network Namespace

If you need to see networking exactly as a specific container sees it (same IP, same DNS, same routing table), you can join its network namespace:

```
# See the network exactly as your nodeserver container sees it
docker run -it --rm --network container:mystack_nodeserver_1 nicolaka/netshoot

# Now 'ip addr' shows the SAME interfaces as the target container
ip addr
# 'ss -tlnp' shows the SAME listening ports
ss -tlnp
# DNS resolution works identically to the target container
dig mongo
```

This is incredibly useful when your application claims it can't reach the database but `curl` from your host machine works fine. By joining the container's network namespace, you see exactly what the application sees.

Running netshoot with Host Networking

For debugging overlay network issues in Swarm, you sometimes need to see the host's full network stack:

```
docker run -it --rm --network host nicolaka/netshoot

# See all veth pairs (each represents a container's network interface)
ip link show type veth

# See the Docker bridge networks
brctl show

# Watch traffic on the Docker overlay network
tcpdump -i docker_gwbridge
```

Common Debugging Scenarios with netshoot

"My app can't connect to the database":

```
# From inside netshoot on the same network:
# 1. Can DNS resolve the service name?
dig mongo +short
# If this returns nothing, the service isn't on this network

# 2. Is the port open?
nc -zv mongo 27017
# "Connection to mongo 27017 port [tcp/*] succeeded!" = port is open
# "Connection timed out" = firewall, wrong network, or service not running

# 3. Can you actually connect?
curl -v telnet://mongo:27017
```

"Requests are slow between services":

```
# Measure latency between containers
ping nodeserver -c 10

# Measure throughput
iperf3 -c nodeserver

# Watch the actual packets for unusual delays
tcpdump -i any -n host nodeserver
```

"I'm getting the wrong IP in my logs":

```
# Watch the actual HTTP headers arriving at your service
tcpdump -i any -A port 3000 | grep -i "x-forwarded\|x-real-ip\|host:"
```

Add `netshoot` to your mental toolbox the same way you'd keep a multimeter in your electrical toolbox. You don't need it every day, but when you need it, nothing else will do.

WHAT BELONGS IN YOUR CONTAINER (AND WHAT DOESN'T)

This is the section that will make the biggest difference to your container size, performance, and security, and it's the architecture decision most developers get wrong on day one.

The Mistake Everyone Makes

A typical Express application in the wild looks like this:

```
const express = require('express');
const helmet = require('helmet');
const compression = require('compression');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const morgan = require('morgan');
const hpp = require('hpp');
const xss = require('xss-clean');
const mongoSanitize = require('express-mongo-sanitize');

const app = express();

// "Security" middleware
app.use(helmet());                    // Security headers
app.use(cors());                      // CORS headers
app.use(hpp());                       // HTTP parameter pollution
app.use(xss());                       // XSS sanitization
app.use(mongoSanitize());             // NoSQL injection prevention

// "Performance" middleware
app.use(compression());               // Gzip compression
app.use(morgan('combined'));           // Access logging

// Rate limiting
app.use(rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  standardHeaders: true,
}));

// SSL termination (sometimes even this)
const https = require('https');
const fs = require('fs');
https.createServer({
  key: fs.readFileSync('./certs/key.pem'),
  cert: fs.readFileSync('./certs/cert.pem'),
}, app).listen(443);
```

This developer has turned Node.js into a web server, a firewall, a compression engine, a rate limiter, a logger, and an SSL terminator. Every single incoming request - including the thousands of malicious ones that hit any public-facing server daily - passes through all of these JavaScript middleware layers on the single-threaded event loop before reaching the actual business logic.

The result: the container needs 200-400 MB of RAM, has a `node_modules` directory stuffed with security packages, and still doesn't handle any of these concerns as well as purpose-built tools that have been doing this for decades.

The Right Architecture: Let Each Container Do One Thing

Containers are cheap. That's the whole point. Instead of one bloated container that does everything, you run multiple lean containers that each do one thing well:

```
Internet traffic (malicious + legitimate)
    |
    v
[NGINX container] - SSL termination
    |                 Gzip compression
    |                 Security headers
    |                 Rate limiting
    |                 Bot blocking
    |                 IP blocking
    |                 Request size limits
    |                 Slowloris protection
    |                 Static file serving
    |                 Access logging
    |
    v  (only clean, filtered, decompressed traffic)
[Node.js container] - Business logic ONLY
    |
    v
[MongoDB container] - Data storage ONLY
```

Your Node.js container receives only pre-filtered, pre-decompressed, already-authenticated requests from NGINX. It does one thing: run your application logic. That's why it can be 26-38 MB instead of 300 MB.

What NGINX Does Better Than Node (And Why)

NGINX is written in C. It's compiled, it's multi-process, and it was specifically designed to sit on the internet and absorb abuse. Here's what it handles better than any Node.js middleware:

SSL/TLS termination:

Node.js handles SSL in JavaScript on the event loop. Every TLS handshake (CPU-intensive RSA/ECDSA operations) blocks your business logic. Under high load, your API response times spike because the event loop is busy doing cryptographic math.

NGINX handles SSL in compiled C across multiple worker processes. The handshakes happen in parallel, and your Node process never sees any of it. It receives plain HTTP over the internal Docker network.

```
# NGINX handles all SSL
server {
    listen 443 ssl;
    ssl_certificate /etc/nginx/certs/server.pem;
    ssl_certificate_key /etc/nginx/certs/server.key;
    ssl_protocols TLSv1.2 TLSv1.3;

    location / {
        proxy_pass http://mystack_nodeserver:3000;
        # Node receives plain HTTP - zero SSL overhead
    }
}
```

Gzip compression:

The `compression` npm package compresses every response in JavaScript. For a 500 KB JSON response, that's real CPU work on the event loop. Multiply by concurrent requests.

NGINX compresses in compiled C, in parallel, across worker processes. Your Node response leaves as plain JSON, NGINX compresses it before sending to the client.

```
gzip on;
gzip_comp_level 5;
gzip_min_length 256;
gzip_types text/plain application/json application/javascript text/css;
```

Rate limiting:

`express-rate-limit` stores request counts in Node's memory (or Redis). Every incoming request, including malicious ones, hits your event loop just to check if it should be rate-limited.

NGINX rate limiting happens before the request ever reaches Node. Abusive clients get rejected at the proxy level. Node never sees them, never allocates memory for them, never processes them.

```
limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;

location /api/ {
    limit_req zone=api burst=20 nodelay;
    proxy_pass http://mystack_nodeserver:3000;
}
```

Security headers:

`helmet` adds headers like `X-Frame-Options`, `Strict-Transport-Security`, `Content-Security-Policy` in JavaScript middleware. It works, but it's a JavaScript function call on every single response for something that should be a static configuration.

```
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Content-Type-Options "nosniff" always;
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" alwa
add_header Content-Security-Policy "default-src 'self'" always;
add_header Referrer-Policy "strict-origin-when-cross-origin" always;
```

Static config, zero per-request cost. Done once, applied to every response by the proxy.

Request filtering and body size limits:

Before Node even parses the request body, NGINX can reject oversized payloads, block suspicious paths, and drop malformed requests:

```
client_max_body_size 10m;              # Reject uploads over 10 MB
client_body_timeout 12;                # Drop slow request bodies
client_header_timeout 12;              # Drop slow headers (slowloris defense)

# Block common attack paths
location ~* /(\.env|\.git|wp-admin|phpMyAdmin) {
    return 444;                        # Drop connection, no response
}
```

Node never allocates a buffer for a 2 GB malicious upload. Node never parses a request to `/.env`. NGINX drops them before they reach the application network.

Static file serving:

Node developers use `express.static()` to serve frontend assets. Every CSS file, every JS bundle, every image goes through the Node event loop. Under load, static file serving competes with API requests for the same single thread.

NGINX serves static files from disk using the kernel's `sendfile()` system call - zero-copy I/O that doesn't even pass through userspace. It serves thousands of concurrent static file requests without breaking a sweat.

```
location /static/ {
    alias /var/www/static/;
    expires 30d;
    add_header Cache-Control "public, immutable";
}

location /api/ {
    proxy_pass http://mystack_nodeserver:3000;
}
```

What This Means for Your Node Container

Once you offload all this to NGINX, your Node app becomes dramatically simpler:

```
const express = require('express');
const app = express();

// That's it. No helmet, no compression, no cors, no rate-limit,
// no morgan, no hpp, no xss-clean, no mongo-sanitize.
// The proxy handles all of it.

app.use(express.json());  // Parse JSON bodies (still needed)

// Your business routes
app.get('/api/users', usersController.list);
app.post('/api/orders', ordersController.create);
// ...

app.listen(3000);  // Plain HTTP, internal network only
```

Your `package.json` dependencies shrink. Your `node_modules` shrinks. Your image shrinks. Your attack surface shrinks. Your event loop has nothing to do except run your business logic.

```
Before (Node does everything):
  dependencies: 47 packages
  node_modules: 180 MB
  Image size: 340 MB
  RAM usage: 250 MB
  CPU: SSL + compression + rate limiting + business logic

After (Node does business logic only):
  dependencies: 12 packages
  node_modules: 40 MB
  Image size: 65 MB
  RAM usage: 50 MB
  CPU: business logic only
```

The Compose File for This Architecture

```
version: "3.8"

services:
  nginx:
    image: yourregistry/nginx:1.4.72
    ports:
      - "80:80"
      - "443:443"
    configs:
      - source: nginx_conf
        target: /etc/nginx/conf.d/default.conf
    secrets:
      - nginx_server_pem
      - nginx_server_key
    deploy:
      replicas: 2
    networks:
      - app-network

  nodeserver:
    image: yourregistry/nodeserver:1.4.72
    # NO published ports - only reachable via nginx on internal network
    secrets:
      - db_password
    environment:
      - NODE_ENV=production
    deploy:
      replicas: 3
    networks:
      - app-network

  mongo:
    image: mongo:7
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network

networks:
  app-network:
    driver: overlay
    attachable: true

volumes:
  mongo-data:
```

```
configs:
  nginx_conf:
    external: true

secrets:
  nginx_server_pem:
    external: true
  nginx_server_key:
    external: true
  db_password:
    external: true
```

Notice that `nodeserver` has **no published ports**. It's only accessible via the internal overlay network through NGINX. An attacker who scans your server's public ports sees port 80 and 443 - both served by NGINX. They can't reach Node directly. They can't probe Node's debug port. They can't send malformed requests that bypass the proxy. The attack surface is NGINX, and NGINX was built to handle that.

As a General Rule

Never publish Node directly to the internet. Always put a reverse proxy in front - NGINX, Caddy, Traefik, whatever you prefer. Node is single-threaded. When you expose it directly to the web, that one thread handles SSL handshakes, gzip compression, rate limit calculations, bot detection, and your actual business logic on the same event loop. One slowloris attack holding connections open, one malformed payload that takes too long to parse, and the entire event loop blocks. Every user waits.

NGINX handles thousands of concurrent connections across multiple worker processes in compiled C. It was built to sit on the internet and take abuse. Node was not. Let compiled code handle the internet. Let Node handle your app.

What Your NGINX Container Needs

Your NGINX Dockerfile should be equally lean:

```
FROM nginx:stable-alpine

# Remove default config
RUN rm /etc/nginx/conf.d/default.conf

# Your config is injected via Docker Configs or COPY
COPY nginx.conf /etc/nginx/nginx.conf
COPY conf.d/ /etc/nginx/conf.d/

# Security: run as non-root (nginx image supports this)
# NGINX master process runs as root to bind port 80,
# worker processes run as the 'nginx' user automatically

EXPOSE 80 443
```

In production with Swarm, you'd use Docker Configs for the NGINX config and Docker Secrets for the SSL certificates, so you don't need to rebuild the NGINX image when your config changes. We covered both of these in the Configs and Secrets sections above.

The Exception: CORS

CORS is one area where you might still want to handle it in Node. The reason is that CORS responses need to be aware of your application's route structure - which origins are allowed for which endpoints, whether credentials are permitted, which headers are exposed. This is application logic, not infrastructure.

That said, if your CORS policy is simple (same origin for everything, or a static list of allowed origins), NGINX can handle it too:

```
location /api/ {
    add_header Access-Control-Allow-Origin "https://yourdomain.com" always;
    add_header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS" a
    add_header Access-Control-Allow-Headers "Authorization, Content-Type" alway

    if ($request_method = 'OPTIONS') {
        return 204;
    }

    proxy_pass http://mystack_nodeserver:3000;
}
```

If your CORS needs are dynamic (per-route, per-user, conditional), keep it in Node. For everything else, the proxy handles it.

LINE ENDINGS WILL BREAK YOUR CONTAINERS AND YOU WON'T KNOW WHY

This is the single most invisible, maddening problem in Docker development on Windows and mixed-OS teams. A shell script that works perfectly when you test it locally fails inside the container with a cryptic error. An entrypoint script exits immediately. A config file is "malformed." A secret value doesn't match. The file looks identical in every editor. Nothing is wrong except everything is broken.

The cause: Windows line endings.

The Problem

Windows uses `\r\n` (carriage return + line feed) as line endings. Linux uses `\n` (line feed only). When you edit a file on Windows - or when Git checks out a file and converts line endings on Windows - every line gets an invisible `\r` character appended.

Your text editor doesn't show it. `cat` doesn't show it. `diff` might not show it. But the Linux shell inside your container sees it, and it doesn't know what to do with it.

Here's what actually happens. You write an entrypoint script on Windows:

```
#!/bin/sh
echo "Starting application"
node server.js
```

What you think the file contains:

```
#!/bin/sh\n
echo "Starting application"\n
node server.js\n
```

What it actually contains after Windows (or Git) touches it:

```
#!/bin/sh\r\n
echo "Starting application"\r\n
node server.js\r\n
```

The shell reads the shebang line as `#!/bin/sh\r` - it's looking for an interpreter called `/bin/sh\r`, which doesn't exist. Or it tries to execute `node server.js\r` - a command with an invisible carriage return appended that the shell can't parse. The error messages you get are deliberately unhelpful:

```
/bin/sh: 1: /app/entrypoint.sh: not found
```

The file exists. It's right there. You can `ls` it. But the shell thinks it's "not found" because the shebang line points to a non-existent interpreter. Or you get:

```
exec format error
```

Or:

```
$'\r': command not found
```

Or the worst variant - no error at all. The script just exits silently with code 126 or 127, and you spend hours checking file permissions, PATH variables, and multi-stage COPY instructions when the actual problem is invisible characters.

Where This Bites You

Entrypoint scripts - The most common victim. You write `entrypoint.sh` on Windows, COPY it into the image, and the container won't start. Exit code 126 (permission denied) or 127 (not found), even though `chmod +x` was applied.

Shell scripts in CI/CD - Deployment scripts, health check scripts, database migration runners. They work in your Git Bash terminal on Windows. They fail inside the container.

Config files - NGINX configs, Postfix configs, cron files, any file that gets parsed line-by-line inside the container. A `\r` at the end of a directive can make the parser choke or silently misinterpret the value.

Secrets and environment files - You create a secret from a file edited on Windows. The password is now `"mypassword\r"` instead of `"mypassword"`. Authentication fails. The password "looks correct" in every editor because the `\r` is invisible.

.env files - Your `.env` has `DATABASE_URL=mongodb://mongo:27017/mydb\r`. Docker Compose passes the value including the `\r`. Your MongoDB driver tries to connect to a host that has a carriage return in its URL. Connection timeout.

Git autocrlf - Even if you're on Linux, if a teammate on Windows committed the file with `\r\n` endings, you might pull their line endings. Git's `core.autocrlf` setting controls this, but it's often misconfigured or inconsistent across the team.

The Fix: dos2unix in Your Dockerfile

`dos2unix` is a tiny utility that strips `\r` from files, converting Windows `\r\n` line endings to Linux `\n`. Install it and run it on every file that gets copied into your image that will be executed or parsed by the shell:

```
FROM node:20-bookworm-slim

# Install dos2unix (tiny package, minimal layer overhead)
RUN apt-get update && apt-get install -y --no-install-recommends dos2unix && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Copy entrypoint and config files
COPY entrypoint.sh ./
COPY config/ ./config/

# Convert line endings BEFORE chmod
RUN dos2unix entrypoint.sh && chmod +x entrypoint.sh
RUN dos2unix config/*

# Copy application (JS files are interpreted, not executed by shell - less crit
COPY --chown=node:node . .

USER node
ENTRYPOINT ["./entrypoint.sh"]
```

For Alpine-based images:

```
FROM node:20-alpine

RUN apk add --no-cache dos2unix

COPY entrypoint.sh ./
RUN dos2unix entrypoint.sh && chmod +x entrypoint.sh
```

What to Convert

You don't need to run `dos2unix` on everything. JavaScript files, JSON files, and most application code are fine because Node.js handles `\r\n` transparently. Focus on files that the Linux shell or system daemons will parse:

- `.sh` scripts (entrypoints, health checks, migration runners, CI scripts)
- NGINX `.conf` files
- Crontab files
- Postfix configuration files

- INI/properties files read by system tools
- Any file referenced in `ENTRYPOINT` or `CMD`
- Any file you `source` or pipe into another command

The Git-Level Fix (Belt and Suspenders)

You can also fix this at the Git level so Windows line endings never enter your repository in the first place. Create a `.gitattributes` file in your project root:

```
# Force LF endings for files that will run in containers
*.sh text eol=lf
*.conf text eol=lf
*.cfg text eol=lf
*.yml text eol=lf
*.yaml text eol=lf
Dockerfile text eol=lf
.dockerignore text eol=lf
.env* text eol=lf
entrypoint* text eol=lf
```

This tells Git to always check out these files with `\n` endings regardless of the platform. Even if a developer on Windows has `core.autocrlf=true`, Git will not convert these files.

Use both approaches. `.gitattributes` prevents the problem at the source. `dos2unix` in the Dockerfile catches anything that slips through - files not tracked by Git, files copied from other sources, files from dependencies, files pasted from Stack Overflow on a Windows machine.

How to Detect the Problem

If you suspect line endings are causing an issue, check from inside the container:

```
# Method 1: cat with visible control characters
docker exec mycontainer cat -A /app/entrypoint.sh
# Lines ending with ^M$ have Windows line endings (\r\n)
# Lines ending with just $ have correct Linux line endings (\n)

# Method 2: hexdump the first few lines
docker exec mycontainer hexdump -C /app/entrypoint.sh | head -5
# Look for "0d 0a" (CRLF) vs just "0a" (LF)

# Method 3: file command
docker exec mycontainer file /app/entrypoint.sh
# Will say "with CRLF line terminators" if Windows endings are present
```

If you see `^M` in `cat -A` output or `0d 0a` in the hex dump, that's your problem. Run `dos2unix` and rebuild.

THE DOCKER PITFALLS DEEP DIVE: EVERY COMPLAINT, ADDRESSED

I spend a lot of time reading what developers complain about on Reddit, Hacker News, and Stack Overflow. The same frustrations come up over and over. Some are legitimate problems with real solutions that nobody talks about. Some are the developer's own fault - patterns they carried over from the VM era that don't work in containers. And some are fundamental misunderstandings of how Docker works under the hood.

Let's go through every major one.

"Docker is slow on my Mac"

This is the number one complaint, and it's legitimate - but it's not Docker's fault, and understanding *why* it's slow immediately tells you how to fix it.

Docker containers are Linux. They use the Linux kernel's cgroups and namespaces. On a Linux machine, Docker runs natively - containers share the host kernel, there's no abstraction layer, and performance is essentially identical to running the process directly on the host.

On macOS (and Windows), there is no Linux kernel. Docker Desktop runs a lightweight Linux VM using Apple's Virtualization.framework (or Hyper-V on Windows). Your containers run inside that VM. This is invisible to you - Docker makes it feel native - but the VM is there, and it has consequences.

The biggest consequence is **file system performance on bind mounts**. When you mount your source code from your Mac into a container with `volumes: - ./src:/app`, here's what actually happens: your file lives on macOS's APFS filesystem. Docker has to share that file across a virtual boundary into the Linux VM's ext4 filesystem. Every file read and every file write crosses that boundary. Docker Desktop uses VirtioFS (the fastest option currently) to do this, but it's still a translation layer.

For a Node.js project with 50,000 files in `node_modules`, this is brutal. Every `require()` call is a file read. Every file watch event from nodemon has to cross the boundary. Every build step that touches thousands of files grinds through the translation layer.

The fix:

Don't mount `node_modules` from your host. This is the single biggest performance win on macOS:

```
services:
  app:
    volumes:
      - ./src:/app/src          # Mount only your source code
      - /app/node_modules       # Anonymous volume - stays inside the VM
```

The anonymous volume `/app/node_modules` tells Docker: "don't mount this from the host - use a volume inside the Linux VM." Your `node_modules` stays on ext4 inside the VM where file access is native-speed. Your source code is mounted from the host for live editing, but that's a much smaller number of files.

Other things that help: make sure Docker Desktop is set to use VirtioFS (Settings -> General -> file sharing implementation), allocate enough CPU and RAM (see the Docker Desktop section earlier in this article), and keep your bind mounts as narrow as possible - mount `./src` instead of `.` if you can.

On Linux, none of this matters. Bind mounts are native. File performance is identical to the host. If Docker performance is critical to your workflow and you're on macOS, the answer that nobody wants to hear is: develop on Linux, or accept the VirtioFS overhead and structure your mounts intelligently.

On Windows, the equivalent problem is even worse - and the fix is different. Docker Desktop uses WSL 2 as its backend. If your project files live on the Windows filesystem (`C:\Users\...` or `/mnt/c/...` from WSL), every file operation crosses the Windows-to-Linux boundary, which is even slower than the macOS VirtioFS layer. And critically, **file system watch events (inotify) don't cross this boundary at all**, which breaks hot reload completely. The fix isn't polling - it's moving your project into the WSL native filesystem (`~/projects/myapp`), not `/mnt/c/...`. Open it in VS Code using the WSL extension. See the "Hot Reload Stops Working" section under Developer Daily Grind for the full walkthrough.

"Docker eats all my disk space"

This is the second most common complaint, and it's 100% the developer's fault.

Every time you build an image, Docker creates cached layers. Every time a build fails halfway, the intermediate layers stay. Every time you pull a new version of `node:20-bookworm-slim`, the old version stays. Every `docker compose down` leaves the named volumes behind. Every stopped container sits there consuming space until you remove it.

After a few weeks of active development, it's completely normal to have 20-30 GB of Docker data you don't need. I've seen developers with 80+ GB of orphaned images and volumes.

```
# See exactly what's using space
docker system df

# Typical output after weeks of development:
# TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
# Images          47        4         15.2GB    13.1GB (86%)
# Containers      23        3         450MB     380MB (84%)
# Local Volumes   12        3         8.4GB     6.2GB (73%)
# Build Cache     -         -         3.8GB     3.8GB (100%)
```

86% of your image space is reclaimable. 100% of your build cache is reclaimable. That's 23 GB you can get back in one command:

```
# Safe cleanup - removes stopped containers, dangling images,
# unused networks, and build cache
docker system prune

# Aggressive cleanup - also removes all images not used by
# a currently running container
docker system prune -a

# The nuclear option - also removes unused volumes
# CAREFUL: this deletes your database data if it's in a volume
docker system prune -a --volumes
```

Build a habit. Run `docker system prune` weekly. Run `docker system prune -a` monthly. Check `docker system df` before complaining that your disk is full. This is housekeeping, the same way you'd clear old git branches or empty your trash.

"My container works locally but fails in production"

This is the "it works on my machine" problem that Docker was supposed to solve - and it does, as long as you don't undermine it. Here's why containers still behave differently between environments:

Different environment variables. Your local `.env` has `NODE_ENV=development` and `MONGO_URI=mongodb://localhost:27017`. Production has different values. If your app has a code path that only runs in production (and it almost certainly does), you've never tested it locally. `NODE_ENV=production` alone changes Express caching, error output, logging levels, and npm's install behavior - see the dedicated NODE_ENV section in the Dockerfile chapter for the full breakdown. Solution: run your local development with `NODE_ENV=production` occasionally to catch these.

Bind mounts masking the image contents. Locally, you mount `./src:/app` which overwrites what's inside the image. In production, there are no bind mounts - the container runs with whatever was `COPY`'d during the build. If you forgot to `COPY` a file, or your `.dockerignore` excludes something important, it works locally (because

the bind mount provides it) but fails in production (because it's not in the image). Solution: periodically test without bind mounts - just `docker compose up` without the volume overrides.

Platform differences. You're building on an Apple Silicon Mac (ARM64). Your production server is x86_64. If your image contains native binaries compiled for ARM, they won't run on x86. Solution: build multi-platform images with `docker buildx build --platform linux/amd64,linux/arm64`, or at minimum build with `--platform linux/amd64` when targeting x86 production servers.

Resource limits. Locally, your container might have access to 8 GB of RAM. In production, you set `memory: 400M`. Your app works fine locally but OOMs in production. Solution: set resource limits in development that match production (we covered this in the compose file section).

Missing health checks. Locally, you don't need health checks - you just look at the logs. In Swarm, without a health check, rolling updates can't verify the new container is healthy. A broken container gets marked as "running" and the old one gets killed. Solution: always have health checks, even locally.

"Docker networking is confusing"

It is. Here's the mental model that makes it click.

Docker has three networking modes that matter: **bridge**, **host**, and **overlay**.

Bridge (default for `docker compose up`) creates an isolated network on a single host. Containers on the same bridge network can talk to each other using service names. Containers on different bridge networks can't see each other. This is what you use for local development.

Host (rarely used) removes network isolation entirely - the container shares the host's network stack. Your container's port 3000 *is* the host's port 3000. No port mapping needed, but also no isolation. Don't use this unless you have a specific reason.

Overlay (required for Swarm) creates a network that spans multiple hosts. It works identically to bridge from the container's perspective - service names resolve, ports work - but under the hood it uses VXLAN tunnels to route traffic between physical machines. This is what you use in Swarm production.

The thing that confuses people is **why containers can't talk to each other**. The answer is almost always one of these:

1. They're not on the same network. Check with `docker network inspect`.
2. You're using the wrong name. In Compose, the service name is the DNS name. Not the container name, not the image name - the service name from your compose file.
3. The target container isn't ready yet. DNS resolves, but the application inside hasn't started listening. This is the `depends_on` problem - use retry logic.
4. You have a firewall or security group blocking the port. This is a host-level issue, not Docker.

If you can `ping` a service by name from inside a container but can't connect to its port, the problem is the application - not Docker's networking.

"I can't figure out volume permissions"

This is covered in detail in the Container Users and Permissions section above. The short version: on Linux, container UIDs are the same as host UIDs with no remapping. If your app runs as UID 1000 but the volume was created as root, your app can't write to it. The fix is defining a user in your Dockerfile with `--chown` on your COPY instructions and an entrypoint script that fixes volume ownership on startup. macOS remaps UIDs transparently so you might not see this locally, which makes it worse when it breaks in production.

"My builds are painfully slow"

We covered layer caching in the Dockerfile section, but here's the checklist when your builds are slow:

Is your `.dockerignore` correct? Run `docker build` and watch the "Sending build context" line. If it says "Sending build context to Docker daemon 1.2GB" - your build context is too large. You're probably sending `node_modules`, `.git`, or large data files. Fix your `.dockerignore`.

Is your COPY order wrong? If `COPY . .` comes before `RUN npm ci`, every code change triggers a full dependency install. Put package files first, install dependencies, then copy source code. This was explained in detail in the Dockerfile section.

Are you using `--no-cache` as a habit? Stop. If you need `--no-cache` regularly, your Dockerfile has a caching problem. Fix the Dockerfile.

Are you pulling base images every time? By default, Docker uses the local cache of base images. If you're doing `docker compose pull` before every build, you're downloading gigabytes of base images you already have. Only pull when you want to update the base image.

Did you lose your layer cache? After `docker system prune -a` or on a fresh CI runner, your build starts from scratch because there's no local cache. Use `cache_from` to pull layers from a previously pushed image:

```
services:
  nodeserver:
    build:
      context: ./nodeserver
      cache_from:
        - "yourregistry/nodeserver:latest"
    image: "yourregistry/nodeserver:${BUILD_VERSION:-latest}"
```

Docker pulls the `latest` image and reuses its layers as cache. If your `package.json` hasn't changed since the last push, the `npm ci` layer is cached even on a machine that has never built this image before. This can cut CI build times from 10 minutes to under a minute.

Is it a macOS file sharing issue? If your build context has thousands of files and you're on macOS, the file sharing layer adds overhead. Try moving the build context to a named volume, or narrow down what you're sending with a stricter `.dockerignore`.

"Docker Compose keeps recreating my containers"

Compose recreates a container when its configuration changes - environment variables, volumes, ports, image, or any setting in the compose file. If you're seeing unexpected recreations, it's usually because:

You changed an environment variable. Even adding a comment in your `.env` file can trigger this if the file is mounted or referenced.

You're using `latest` tag and pulled a new image. The image digest changed, so Compose sees it as a new configuration. Pin your versions.

Your build produced a new image. If you use `--build`, and *anything* in the build context changed (even a log file you forgot to `.dockerignore`), the image hash changes and Compose recreates the container.

Your compose file has a timestamp or dynamic value. If any environment variable references something that changes every run (like `BUILD_DATE=$(date)`), Compose sees a new config every time.

Use `docker compose up -d` without `--build` when you haven't changed the image. Use `--build` only when you've changed the Dockerfile or application code. And stop using `latest` - pin to specific versions so the image digest only changes when you intentionally push a new version.

"I keep running out of memory"

Two causes: either your containers don't have memory limits, or they have limits that are too low.

No limits: Without a memory limit, a container can consume all available memory in the Docker VM. One container with a memory leak takes down everything.

```
# ALWAYS set memory limits - even in development
deploy:
  resources:
    limits:
      memory: 400M
```

Too-low limits: If your limit is lower than what your application actually needs, Docker kills the container with exit code 137 (OOMKilled). Check the actual usage first:

```
docker stats
```

Watch the MEM USAGE column. If your app consistently uses 350 MB, don't set the limit to 256M. Set it to 512M - give yourself headroom for spikes. Memory limits should be a safety net, not a straightjacket.

For Node.js specifically, also set `--max-old-space-size` to match your container's limit:

```
environment:
  - NODE_OPTIONS=--max-old-space-size=350
```

This tells Node's garbage collector to start reclaiming memory before hitting the container limit. Without this, Node might think it has 4 GB available (the VM's total), allocate aggressively, and get OOMKilled by Docker before the garbage collector kicks in.

"Secrets end up in my image"

This is a security disaster that happens more often than people admit. You need an API key during the build - maybe to install a private npm package or download an artifact. So you put it in an environment variable or copy a file:

```
# CATASTROPHICALLY WRONG - the secret is baked into a layer
ENV NPM_TOKEN=abc123secret
COPY .npmrc /app/.npmrc
RUN npm ci
RUN rm /app/.npmrc   # This doesn't help - it's still in the previous layer
```

Deleting the file in a later layer doesn't remove it from the image. Docker images are layered. The `.npmrc` file exists in the layer where you `COPY`'d it. Anyone who pulls your image can extract that layer and read the file.

The fix - use BuildKit secrets:

```
# syntax=docker/dockerfile:1
RUN --mount=type=secret,id=npmrc,target=/app/.npmrc npm ci
```

Build with:

```
docker build --secret id=npmrc,src=$HOME/.npmrc .
```

The secret is mounted during the build step but never written to a layer. It exists only in memory during that single `RUN` instruction. The resulting image contains zero trace of it.

For runtime secrets in Swarm, use Docker Secrets - they're encrypted at rest, encrypted in transit, and mounted as in-memory files inside the container at `/run/secrets/`.

"Running everything as root inside containers"

This is covered extensively in the Container Users and Permissions section above. The short version: every Docker container runs as root by default, which means container escape vulnerabilities give the attacker root on the host. The official `node` image ships with a non-root `node` user (UID 1000) that most developers don't know exists. Use `COPY --chown=node:node`, then `USER node` before your ENTRYPOINT. The only operations that need root are installing system packages and binding to ports below 1024 (which you should handle with a reverse proxy container instead).

"Using `latest` tag in production"

`latest` is not a version. It's the default tag Docker applies when you don't specify one. It doesn't mean "the most recent build." It means "whatever was last tagged as latest."

Here's what goes wrong: you build and push `myapp:latest` on Monday. Everything works. On Wednesday, a teammate builds and pushes `myapp:latest` with a broken migration. On Thursday, your Swarm node restarts and pulls `myapp:latest` - it gets Wednesday's broken build. Your production is now running code you didn't deploy. You have no idea which version is running because there's no version.

```
# Tag with the git commit hash - always unique, always traceable
docker build -t myapp:$(git rev-parse --short HEAD) .
docker push myapp:$(git rev-parse --short HEAD)

# Or tag with a semantic version from your build pipeline
docker build -t myapp:${BUILD_VERSION} .
docker push myapp:${BUILD_VERSION}
```

When something breaks at 2 AM, you need to know exactly which version is running. `docker service inspect mystack_nodeserver` should give you `myapp:1.4.72` or `myapp:a3f8c2d` - not `myapp:latest`.

"Not using `.env` files properly"

Docker Compose reads a `.env` file automatically from the same directory as your compose file. Use it for environment-specific values:

```
# .env (for local development)
REGISTRY=yourregistry
NODE_ENV=development
MONGO_URI=mongodb://mongo:27017/mydb
API_KEY=dev-key-12345
REPLICAS=1
```

```
# docker-compose.yml
services:
  nodeserver:
    image: "${REGISTRY}/nodeserver:latest"
    environment:
      - NODE_ENV=${NODE_ENV}
      - MONGO_URI=${MONGO_URI}
      - API_KEY=${API_KEY}
    deploy:
      replicas: ${REPLICAS}
```

Two critical rules: **never add `.env` to your Docker image** (put it in `.dockerignore`) and **never commit `.env` to git** (put it in `.gitignore`). Commit a `.env.example` with placeholder values so new developers know which variables are needed. For production, use Docker Secrets or your CI/CD pipeline's secret management.

PUTTING IT ALL TOGETHER

The goal of all of this is one thing: **your development environment should mirror your production environment as closely as possible**. Same images. Same networking model. Same health checks. Same resource constraints. Same signal handling. Same secrets management. Same non-root user. Same NGINX-in-front architecture.

When you develop with these patterns from day one, deploying to Swarm isn't a migration - it's a configuration change. You swap `bridge` for `overlay`. You remove bind mounts. You push the image to a registry. You run `docker stack deploy`. That's it.

Most of the "Docker in production is hard" complaints I see come from teams that developed with completely different patterns than they deploy with. They used bind mounts and `depends_on` and `container_name` and `restart: always` - none of which exist in Swarm. They put passwords in environment variables. They ran everything as root. They installed `helmet` and `compression` in Node instead of putting NGINX in front. They never set up versioning, so when something breaks they can't trace it back. They skipped health checks, so Swarm doesn't know when a container is sick. Then they're surprised when things break.

The patterns in this guide aren't about Docker specifically. They're about building software that you can deploy, monitor, debug, and roll back with confidence. Docker is just the container that carries it. Get the fundamentals right - lean images, proper secrets, non-root users, structured logging, version tracking, proxy-based security, reproducible builds - and the orchestration layer becomes almost boring. Which is exactly what production should be.

Develop the way you deploy. The rest takes care of itself.

Tim Carter Clausen is a Danish-American full-stack architect and cryptographic researcher who writes at thedecipherist.com. His Docker Swarm Production Guide ranks #1 on Google, ahead of Docker's official documentation. He runs a live, dual-continent SaaS platform on $166/year with zero crashes.