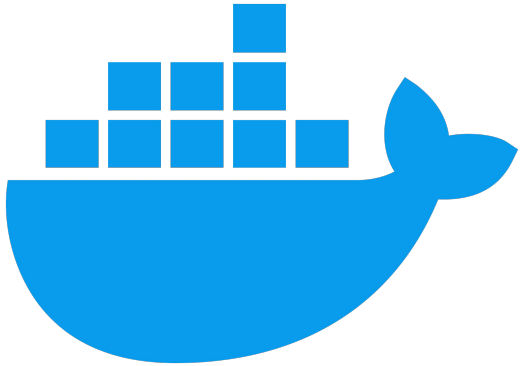


Docker

Build, Ship, and Run Applications Anywhere.

Roberto Molero.

What is Docker?

Docker is a software platform that automates the deployment of applications within software **containers**. It provides an additional layer of abstraction and automation of operating-system-level virtualization, allowing an application and all its dependencies to be packaged into a standardized and isolated environment.

The Power of Docker: Solving Deployment Chaos

The Problem: "It Works on My Machine"

The traditional software development cycle faces three critical friction points:

- **Environmental Inconsistency:** Differences in Operating Systems, libraries, and configurations between a developer's laptop and the production server.
- **Dependency Hell:** Manually installing runtimes (Python, Node, Java) is slow, error-prone, and leads to version conflicts.
- **The Deployment Gap:** Fear that code working in development will break when deployed due to a missing "hidden" setting.

The Solution: Standardized Containers

Docker solves these issues by introducing a **standard unit of software**:

- **Encapsulation:** It packages code, libraries, and dependencies into a single **Image**.
- **Isolation:** The application runs in its own sandbox, independent of the host OS settings.
- **Immutability:** The exact same bits you test locally are the bits that run in the cloud.

Key Benefits

Predictability: Guaranteed behavior across Dev, Test, and Production.

Portability: "Write once, run anywhere"—on any cloud or local server.

Efficiency: Rapid startup times and automated deployments with a single command.

Docker key concepts

Image: An **immutable template** that contains your application’s code, libraries, and dependencies. It acts as the "blueprint" or "read-only snapshot" of your environment.

Container: A **running instance** of an image. If the image is the recipe, the container is the meal. It is a lightweight, isolated, and executable package.

Dockerfile: A text file containing the **set of instructions** used to build an image. It automates the environment setup step-by-step.

Docker key concepts

Registry: A **storage and distribution service** for images. Examples include **Docker Hub** (public), **GHCR** (GitHub Container Registry), or private enterprise registries.

Volume: A mechanism for **persistent data storage**. Since containers are ephemeral (data is lost when they are deleted), volumes allow you to save data outside the container.

Network: The infrastructure that enables **communication between containers**. It allows different services (like a web app and a database) to talk to each other securely.

How Docker Works: Build, Ship, and Run

Build (The Dockerfile):

- The process starts with a **Dockerfile**, where the developer defines the environment.
- Running the `docker build` command transforms this text file into a **Docker Image**.

Ship (The Registry):

- The image is uploaded (pushed) to a **Registry** (like Docker Hub).
- This makes the application portable and accessible from anywhere in the world.

Run (The Container):

- The image is downloaded (pulled) onto any machine with Docker installed.
- Running the `docker run` command creates a **Container**, which is the live, executing version of your software.

Developer → `Build` → **Image** → `Ship` → **Registry** → `Run` → **Container**

Local Workflow vs. Remote Workflow

Local Workflow (Build & Run):

- If you are developing or testing on your own, you simply execute `docker build` and the image is stored in your computer's **Local Image Store**.
- Immediately after, you can execute `docker run`; Docker will look for the image on your local drive before attempting to search for it on the internet.
- **Docker Desktop** acts as the manager for this entire cycle without ever leaving your machine.

Remote Workflow (Ship):

- Pushing (uploading) to a **Registry** (such as Docker Hub) is only necessary when:
  - You want to **share** the image with a teammate.
  - You want to **deploy** the application to a cloud server (AWS, Azure, Google Cloud, etc.).
  - You are using a **Continuous Integration / Continuous Deployment (CI/CD)** pipeline.

Images vs. Containers: The Core Difference

State: Static vs. Dynamic

Docker Image (Immutable): Once an image is created, it **cannot be changed**. If you need to update the application or change a configuration, you must build a new image. This ensures that the environment is always identical.

Docker Container (Mutable): While a container is running, it is "mutable." It adds a thin **Writable Layer** on top of the static image layers. This allows the application to write logs or temporary files without ever altering the original image.

An **Image** is what you build and share; a **Container** is what you run and scale.

Images vs. Containers: The Core Difference

Lifecycle: Permanent vs. Ephemeral

Docker Image (Permanent): An image exists as long as you don't manually delete it from your disk. It acts as a permanent version of your application.

Docker Container (Ephemeral): Containers are designed to be temporary. When a container is deleted, its internal state is destroyed, but the **Image remains untouched**, ready to launch a fresh new container.

An **Image** is what you build and share; a **Container** is what you run and scale.

Images vs. Containers: The Core Difference

Purpose: Packaging vs. Execution

Docker Image (The Package): Its primary goal is **distribution**. It is what you upload to a Registry to share your software.

Docker Container (The Runtime): Its primary goal is **execution**. It provides the isolated sandbox where your code actually runs.

An **Image** is what you build and share; a **Container** is what you run and scale.

Images vs. Containers: The Core Difference

Resource Usage: Storage vs. Compute

Docker Image (Storage): It only occupies **Disk Space**. The size depends on the layers and dependencies it contains.

Docker Container (Compute): It occupies **RAM and CPU**, depending on the application's activity at that moment.

An **Image** is what you build and share; a **Container** is what you run and scale.

The Dockerfile: Building Your Image

What is it?

A **Dockerfile** is a simple text document that contains a list of commands. Think of it as a **scripted recipe** or an automated "To-Do" list that tells Docker exactly how to package your application.

What is it for?

- **Automation:** It replaces manual installation steps. No more "follow this 10-step guide to set up your environment."
- **Documentation:** The Dockerfile *is* the documentation of your infrastructure.
- **Portability:** It allows any developer to recreate the exact same image by just having this file.

The Dockerfile: Line-by-Line Breakdown

1. FROM: The Foundation

- **The Command:** `FROM python:3.11-slim`
- **What it does:** It tells Docker which base to build on. Instead of installing Linux and Python manually, you start with an official, pre-configured image.
- **Analogy:** Choosing the plot of land for your house.

2. WORKDIR: The Worksite

- **The Command:** `WORKDIR /app`
- **What it does:** It creates a directory named `/app` inside the image and moves into it. All following commands will happen here.
- **Analogy:** Walking into the specific room where you are going to work.

The Dockerfile: Line-by-Line Breakdown

3. COPY: The Materials

- **The Command:** `COPY . .`
- **What it does:** It takes all the files from your local folder (the first `.`) and puts them inside the image's current folder (the second `.`).
- **Analogy:** Moving your furniture and tools into the room.

4. RUN: The Preparation

- **The Command:** `RUN pip install -r requirements.txt`
- **What it does:** It executes a command **during the build process**. This installs the libraries your code needs to work.
- **Analogy:** Installing the electricity and plumbing before the house is finished.

The Dockerfile: Line-by-Line Breakdown

5. ENV: The Settings

- **The Command:** `ENV APP_COLOR=blue`
- **What it does:** It defines environment variables. These are configuration settings that your application can read while it is running.
- **Analogy:** Setting the thermostat or the light levels of the room.

6. CMD: The Ignition

- **The Command:** `CMD ["python", "main.py"]`
- **What it does:** This is the **most important** line. It tells the container: "The moment you start, run this specific command."
- **Analogy:** Turning on the lights and opening for business.

The Dockerfile: Line-by-Line Breakdown

7. EXPOSE: The Communication Port

The Command: `EXPOSE 5000`

- **What it does:** It documents which port the application uses inside the container. It informs Docker and the user that the container listens on this specific port at runtime for internal networking.
- **Analogy:** Putting a sign in a building’s lobby that says **"Reception is in Room 5000."** It tells visitors where the service is located, even if the building's front door isn't unlocked for the public yet.

The basic Dockerfile example

```dockerfile
# 1. THE FOUNDATION: Start with Python installed
FROM python:3.11-slim

# 2. THE WORKSITE: Create and move to the /app folder
WORKDIR /app

# 3. THE PREPARATION: Install dependencies
# (We copy just the requirements file first to optimize the cache)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4. THE MATERIALS: Copy the rest of your source code
COPY . .

# 5. THE SETTINGS: Set a configuration variable
ENV APP_COLOR=blue

# 6. THE GATE: Document that the app uses port 5000
EXPOSE 5000

# 7. THE IGNITION: Run the application
CMD ["python", "main.py"]
```

Building and Running Your Image

Step 1: The Build (`docker build`)

This command reads the Dockerfile and creates the Image (the blueprint).

- **The Command:** `docker build -t my-first-app .`
- **What it does:**
  - **`-t` (Tag):** Gives your image a human-readable name (`my-first-app`) so you don't have to remember a random ID number.
  - **`.` (The Context):** Tells Docker to look for the Dockerfile in the current folder.
- **Analogy:** Taking the paper blueprint and using it to manufacture the actual machinery.

To transform your **Dockerfile** into a working application, you follow a simple two-step process: **Build** and **Run**.

Building and Running Your Image

Step 2: The Run (`docker run`)

This command takes the Image and starts a Container (the living process).

- **The Command:** `docker run my-first-app`
- **What it does:** It creates a fresh, isolated sandbox and executes the `CMD` you defined in your Dockerfile.
- **The "Hello World" Result:** If your `main.py` says `print("Hello from my custom image")`, your terminal will show:
  `Hello from my custom image`

To transform your **Dockerfile** into a working application, you follow a simple two-step process: **Build** and **Run**.

Building and Running Your Image

Step 3: Accessing the App (`-p` Flag)

If your app is a website or an API (using the `EXPOSE` port we discussed), you need to "map" the ports.

- **The Command:** `docker run -p 8080:5000 my-first-app`
- **Explanation:** "Take port **8080** on my laptop and connect it to port **5000** inside the container."
- **Result:** You can now open your browser and go to `localhost:8080` to see your app running.

To transform your **Dockerfile** into a working application, you follow a simple two-step process: **Build** and **Run**.

Understanding Image Layers

A Docker Image is not a single big file; it is a stack of **read-only layers**. Each instruction in your Dockerfile (FROM, RUN, COPY, etc.) creates a new layer.

- **Layer 1 (FROM):** The base OS.
- **Layer 2 (RUN):** The installed libraries.
- **Layer 3 (COPY):** Your application code.

Understanding Image Layers

The Power of Caching

Docker is intelligent. When you rebuild an image, it checks if the instructions have changed.

- **If nothing changed:** Docker reuses the existing layer from the **Cache** (Instant).
- **If something changed:** Docker recreates that layer **and every layer above it**.

  The "Small Changes" Rule: If you only change one line of code in your script, Docker doesn't reinstall Python or your libraries; it only replaces the very top layer where your code lives. This makes builds take **milliseconds** instead of minutes.

Understanding Image Layers

Think of a Docker Image as a **Layer Cake**:

1. **Instruction 1 (FROM):** The bottom sponge cake.
2. **Instruction 2 (RUN):** The cream filling.
3. **Instruction 3 (COPY):** The frosting on top.

If you want to change the frosting, you don't need to bake a new sponge or make new cream. You just swap the top layer.

Understanding Image Layers

Best Practices for Speed

To keep your builds fast, follow these two golden rules:

1. **Order Matters:** Put the things that **change the least** (like installing Python or libraries) at the **top** of the Dockerfile. Put the things that **change the most** (your code) at the **bottom**.
2. **Specific Copies:** Instead of copying everything at once, copy your dependency files (like `requirements.txt`) first, install them, and *then* copy your code.

Understanding Image Layers

The WRONG way, SLOW.

```
FROM python:3.11
COPY . .                # <--- If this changes (your code)...
RUN pip install flask   # <--- ...this layer is destroyed and re-run (SLOW)
CMD ["python", "app.py"]
```

The RIGHT way, FAST.

```
FROM python:3.11
COPY requirements.txt .  # 1. Only runs if you add a new library
RUN pip install flask    # 2. Only runs if line 1 changed
COPY . .                 # 3. Your code (Changes often, but it's the last step!)
CMD ["python", "app.py"]
```

Running Containers Like a Pro

Image Creation (Build)

- **Command:** `docker build -t my-app .`
- **Action:** Compiles the Dockerfile into a read-only image.
- **-t (Tag):** Assigns a name and version to the image.
- **"." (Context):** Specifies the directory containing the source files and Dockerfile.

Running Containers Like a Pro

- **-d Detached**
  Runs the container in background mode (prints container ID).

- **-p Publish**
  Maps host ports to container ports (host_port:container_port).

- **--name Name**
  Assigns a custom identifier to the container for easier management.

```
docker run -d -p 8080:5000 --name my_app --rm -e DEBUG=true my-image
```

Running Containers Like a Pro

- **--rm Remove**
  Automatically deletes the container's file system upon exit.

- **-v Volume**
  Mounts a host directory into the container (Data Persistence).

- **-e Env**
  Passes environment variables to the container at runtime.

- **-it Interactive + TTY**
  Allocates a pseudo-terminal and keeps STDIN open for interaction.

```
docker run -d -p 8080:5000 --name my_app --rm -e DEBUG=true my-image
```

Understanding the Base Image

Understanding the Base Image

What it is: The foundation of your container. It is a pre-configured template that contains the Operating System and the runtime environment (like Python, Node, or Java) so you don't have to build them from scratch.

The Role of the Base Image

- **The Starting Point:** Every Dockerfile must start with a `FROM` instruction.
- **Standardization:** It ensures that your app runs on the same "virtual ground," whether it's on a developer's laptop or a cloud server.
- **Layer 0:** It provides the system libraries and the language engine that your code needs to execute.

Choosing the Right Base Image (The "Flavors")

Standard (The Default)

- **The Command:** `FROM python:3.11`
- **What it is:** A full Operating System (usually Debian) with all common tools and libraries included.
- **Pros/Cons:** It’s the easiest to use but very heavy (~900MB). It contains many tools you don't need, which increases the security risk.

Choosing the Right Base Image (The "Flavors")

Slim (The Balanced Choice)

- **The Command:** `FROM python:3.11-slim`
- **What it is:** A stripped-down version of the standard image. It removes documentation and extra packages that aren't needed to run Python.
- **Pros/Cons:** Much smaller (~150MB) and very stable. It is the **recommended** choice for most production environments.

Choosing the Right Base Image (The "Flavors")

Alpine (The Ultra-Lightweight)

- **The Command:** `FROM python:3.11-alpine`
- **What it is:** A tiny image based on Alpine Linux, a security-oriented, minimal distribution.
- **Pros/Cons:** Incredibly small (~50MB). However, it uses different system libraries (`musl`), which can sometimes cause compatibility issues or slower installations for complex Python packages like Pandas or NumPy.

Understanding Volumes (-v)

The Problem: Containers are **ephemeral**. This means that if you save a file or a database record inside a container and then delete the container, **your data is gone forever.**

The Solution: Volumes. A volume is a link between a folder on your **Host (PC)** and a folder inside the **Container**.

```
docker run -v my_data:/var/lib/mysql mysql
```

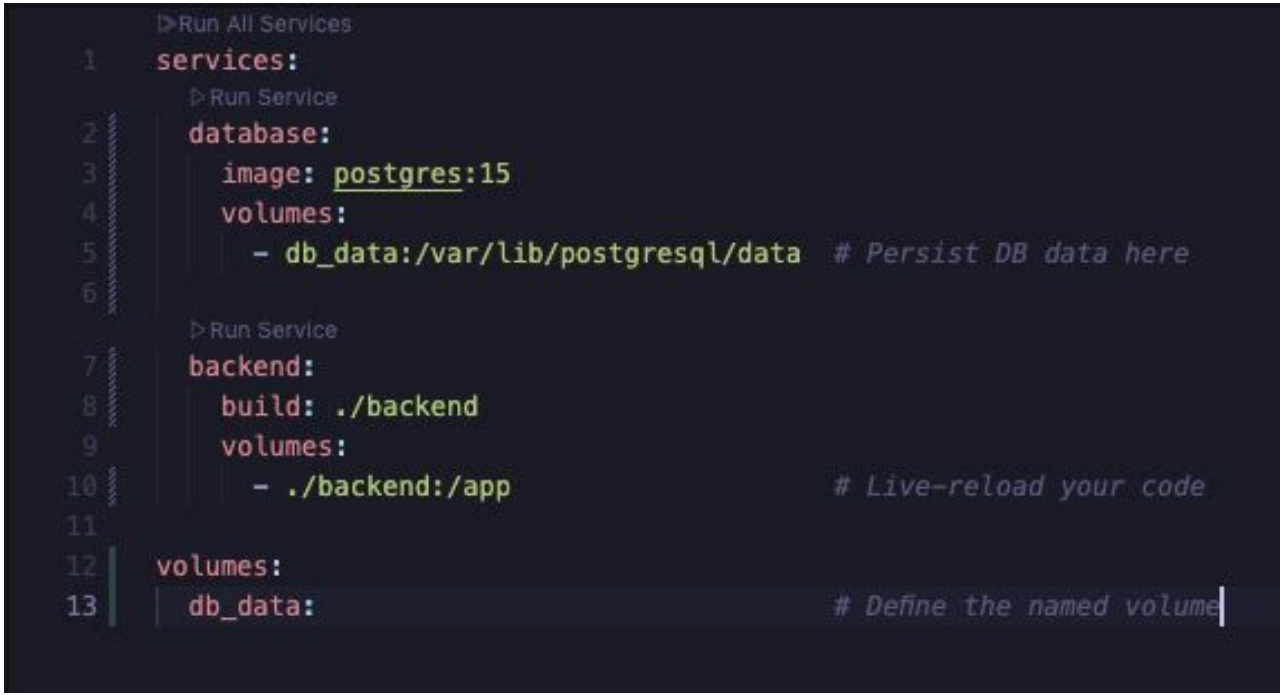
Understanding Volumes (-v)

How it works:

- It’s like a "portal" or a shared folder.
- If the container writes a file in its folder, it actually appears on your physical hard drive.
- If you delete the container and create a new one, you just reconnect it to the same folder, and **your data is still there.**

```
docker run -v my_data:/var/lib/mysql mysql
```

Understanding Volumes (-v)

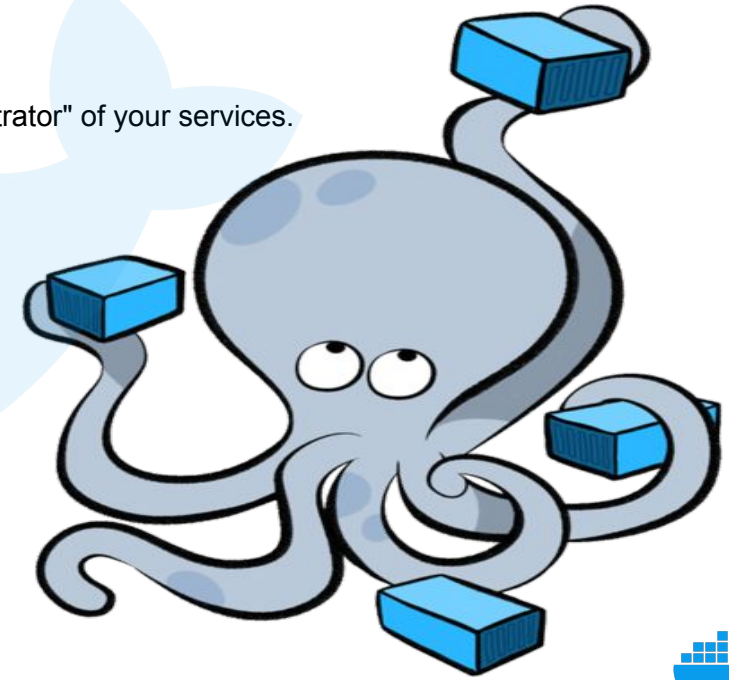


Docker Compose

What it is: A tool for defining and running multi-container applications. Instead of running 10 different `docker run` commands with 10 different flags, you define everything in a single file.

The Configuration: `docker-compose.yml`

Instead of a script, you use a **YAML** file. It is easy to read and acts as the "orchestrator" of your services.



Docker Compose

```yaml
version: '3.8'

▷Run All Services
services:
  # 1. THE FRONTEND (React / Vue / Angular)
  ▷Run Service
  frontend:
    build: ./frontend    # Looks for a Dockerfile inside the /frontend folder
    ports:
      - "3000:3000"      # You access the UI at localhost:3000
    depends_on:
      - backend

  # 2. THE BACKEND (Python / Node / Go)
  ▷Run Service
  backend:
    build: ./backend     # Looks for a Dockerfile inside the /backend folder
    ports:
      - "8000:8000"
    environment:
      - DB_URL=database  # It uses the service name as the address
    depends_on:
      - database

  # 3. THE DATABASE (PostgreSQL / MongoDB)
  ▷Run Service
  database:
    image: postgres:15   # Pre-made official image
    environment:
      POSTGRES_PASSWORD: secret_password
```

GitHub Actions for Docker (CI/CD)

What is CI/CD?

- **CI (Continuous Integration):** Every time you "Push" code, it is automatically built and tested.
- **CD (Continuous Deployment):** If the tests pass, the code is automatically sent to the server.

The Automated Workflow

Instead of building and pushing images manually from your laptop, a **Robot** (GitHub Actions) does it for you in the cloud.

The Workflow Steps:

1. **Push:** You send your code to GitHub.
2. **Build:** A GitHub server runs `docker build`.
3. **Test:** The server runs your tests inside the container.
4. **Push Image:** If tests pass, the image is sent to **Docker Hub** or **GHCR** (GitHub Container Registry).
5. **Deploy:** The production server pulls the new image and restarts.

Simple GitHub Action Workflow

```yaml
jobs:
  # 'build' is the name of the job. You can have multiple jobs running in parallel.
  build:
    # Specifies the virtual machine (runner) provided by GitHub to execute the commands
    runs-on: ubuntu-latest

    steps:
      # Step 1: Downloads a copy of your code from the repository into the runner
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Executes the 'docker build' command using the Dockerfile in your repo
      # It tags the image as 'latest' for identification
      - name: Build the Docker image
        run: docker build -t my-user/my-app:latest .

      # Step 3: Starts a temporary container to run automated tests (e.g., pytest)
      # If the tests fail, the workflow stops here and does NOT push the image
      - name: Run Tests
        run: docker run my-user/my-app:latest pytest

      # Step 4: Authenticates with Docker Hub using encrypted credentials (Secrets)
      # This ensures your password is never visible in the logs
      - name: Login to Docker Hub
        run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USER }}" --password-stdin

      # Step 5: Uploads the verified, tested image to the cloud (Docker Hub)
      # Now, your production server can 'pull' this specific version
      - name: Push to Docker Hub
        run: docker push my-user/my-app:latest
```

.github/workflows/main.yml

Real-World Use Cases

Microservices Architecture

In the past, apps were "Monoliths" (one giant block of code). If one part broke, everything died.

- **The Docker Way:** You split the app into small, independent services (Auth, Payments, Inventory).
- **The Benefit:** Each service can use a different language (Python for AI, Node for API) and can be scaled or updated without touching the others.

Real-World Use Cases

Testing Environments (The "Discardable" Lab)

- **The Problem:** Setting up a testing database or a specific OS version is slow and messy.
- **The Docker Way:** QA engineers can spin up an exact replica of the production environment in seconds using `docker-compose`.
- **The Benefit:** Once the test is finished, you delete the containers (`--rm`), and your machine stays clean.

Real-World Use Cases

Local Development (The End of "It works on my machine")

- **The Problem:** A new developer joins the team and spends 2 days installing dependencies, databases, and specific versions of Python.
- **The Docker Way:** The developer clones the repo and runs `docker-compose up`.
- **The Benefit:** Onboarding takes 5 minutes instead of 2 days. Everyone on the team uses the exact same environment.

Real-World Use Cases

CI/CD Pipelines

- **The Problem:** Code that works on a laptop often fails when deployed to a server.
- **The Docker Way:** As we saw in GitHub Actions, the code is packaged into an image, tested, and shipped automatically.
- **The Benefit:** High-frequency deployments. Companies like Netflix or Amazon deploy code thousands of times a day thanks to this automation.

Real-World Use Cases

Cloud Deployments (Portability)

- **The Problem:** Moving from AWS to Google Cloud or Azure used to be a nightmare.
- **The Docker Way:** Since the app is inside a container, it doesn't care where it's running.
- **The Benefit: Cloud Agnostic.** You can move your containers to any provider or even to your own on-premise servers with zero code changes.

Your code, anywhere.

It doesn't matter if you are on a MacBook in a coffee shop, a Windows PC at home, or a massive Linux server in the cloud.

If it runs in Docker, it runs everywhere.

"Docker doesn't just fix 'It works on my machine.' It scales your machine to the entire world."