WEB DEPLOYMENT

ONE LOVE. ONE FUTURE.

Learning Outcomes

By the end of this lesson, students will be able to:

- Explain the end-to-end web deployment workflow.
- Containerize applications using Docker
- Build CI/CD pipelines with GitHub Actions
- Deploy applications in production environments.
- Apply basic DevOps practices (Nginx, PaaS, monitoring)

Content

1. DevOps Mindset & Deployment Overview
2. Docker & Environment Standardization
3. CI/CD Pipelines with GitHub Actions
4. Production Deployment and Operation

HUST

1. DevOps Mindset & Deployment Flow

1.1. DevOps Mindset

1.2. Deployment Flow

1.1. DevOps Introduction

- DevOps is a combination of Culture, Automation, and Feedback to deliver value to users.
  - Culture
    - Developers care about deployment and production
    - Operations are involved early
    - Mindset: You build it, you run it
  - Automation
    - Automate build, test, deploy, and infrastructure setup
    - Mindset: use CI/CD pipelines, Docker
  - Feedback
    - Fast feedback from production to development
    - Use logs, monitoring, and alerts to detect issues
    - Mindset: Production is a learning environment, not just a destination.

DevOps – What it is NOT

❌ **Not a Tool**

- DevOps is **not** Docker, Kubernetes
- Tools **support** DevOps, but **do not define** it

❌ **Not a Job Title**

- DevOps is a **shared responsibility** across the team
- DevOps means developers **care about production outcomes**

❌ **Not Only CI/CD**

- CI/CD is **part of DevOps**, not DevOps itself

DevOps is about how teams work together, not just the tools they use.

Why Developers need to understand DevOps

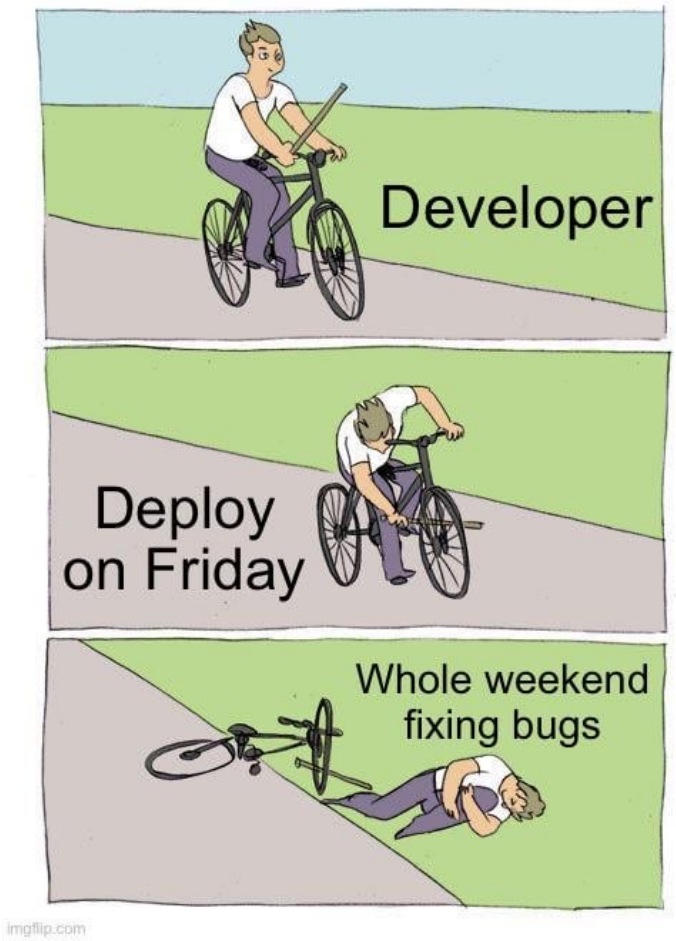
Modern software development does not end with writing code.

- Code runs in production, not in IDE
- Faster and safer deployment
- Better debugging and problem solving understanding
- Industry expectations

Good developers write code.
Excellent developers deliver and operate software.

1.2. Deployment Flow

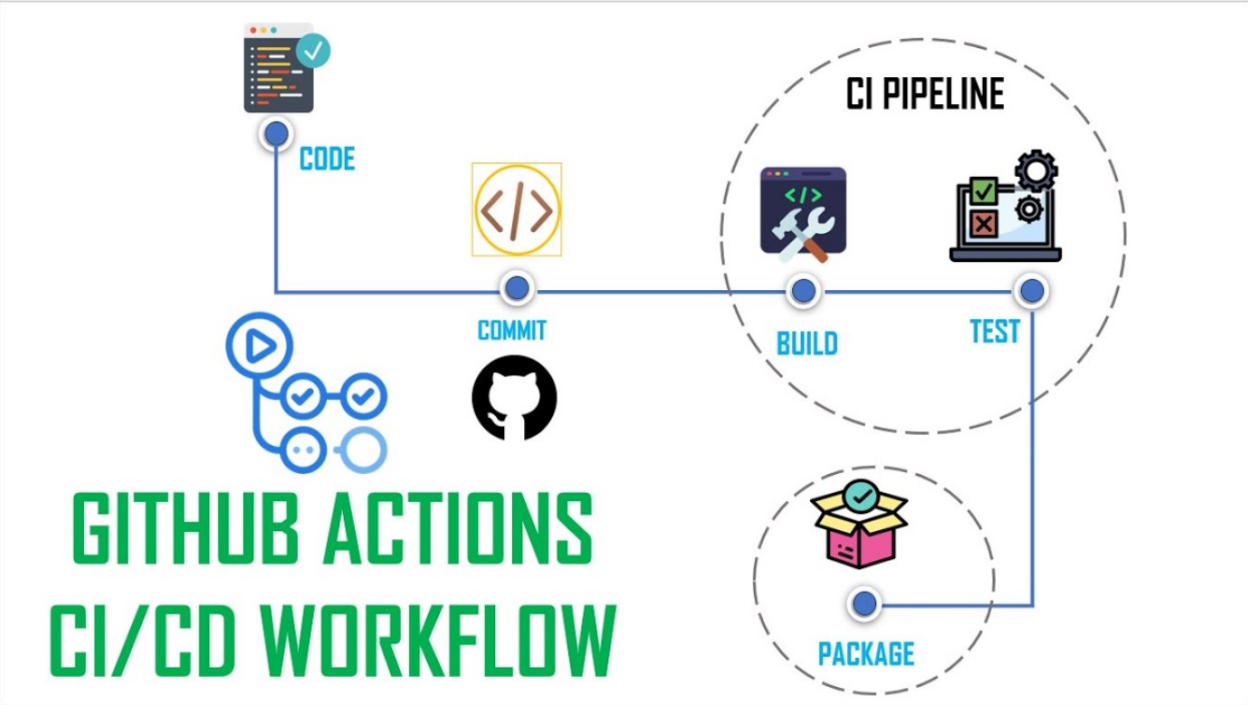
- Deployment is where real problems appear — not during coding.
  - Works on my machine, fails in production.
  - Different environments: local ≠ server ≠ cloud.
  - Manual deployment causes errors and downtime.
  - Hard to debug without logs and monitoring.



From Git Push → CI (Continuous Integration)

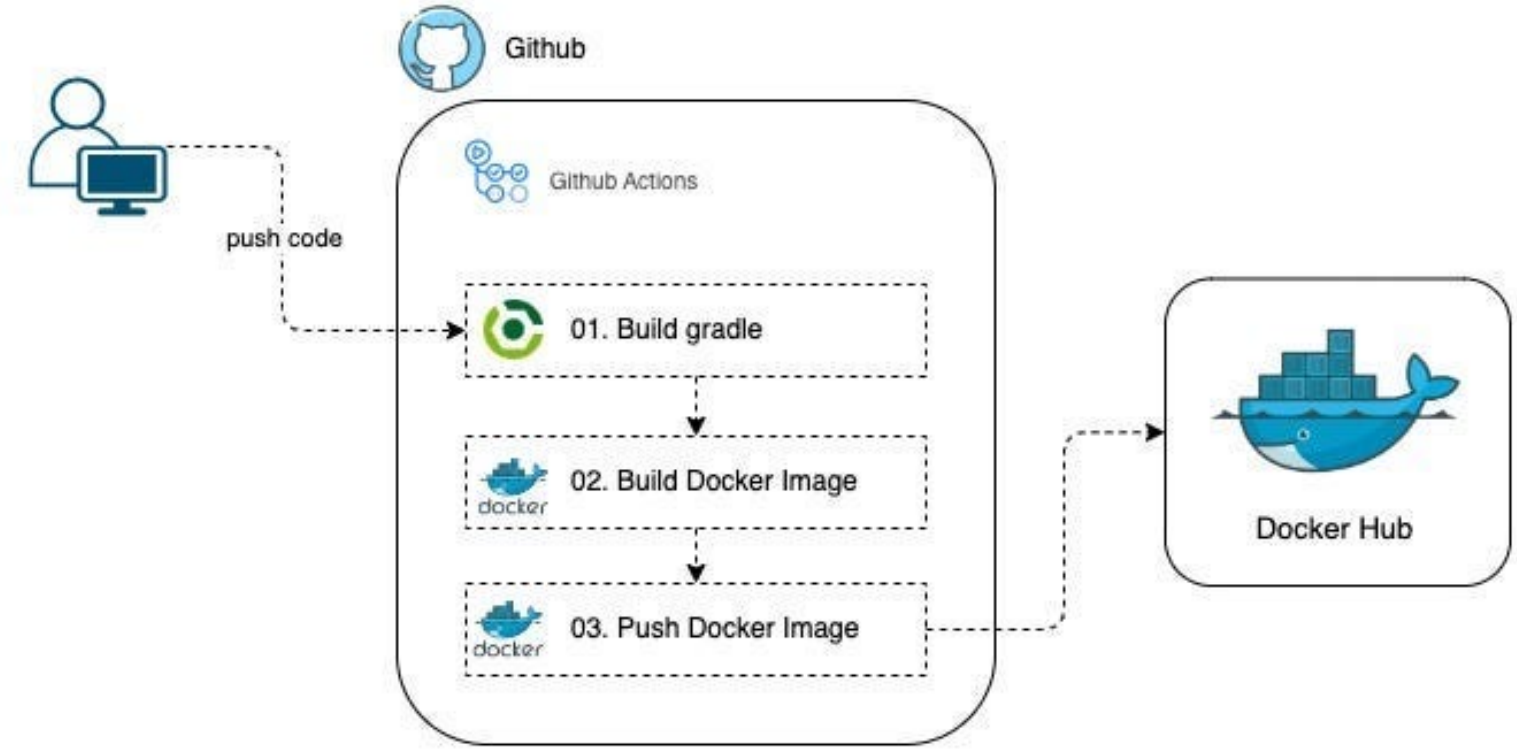
Every git push is a quality checkpoint, not just a code upload

- Developer pushes code to the Git repository.
- The push automatically triggers a CI workflow.
- CI server builds and tests the application.
- Errors are detected early, before deployment.



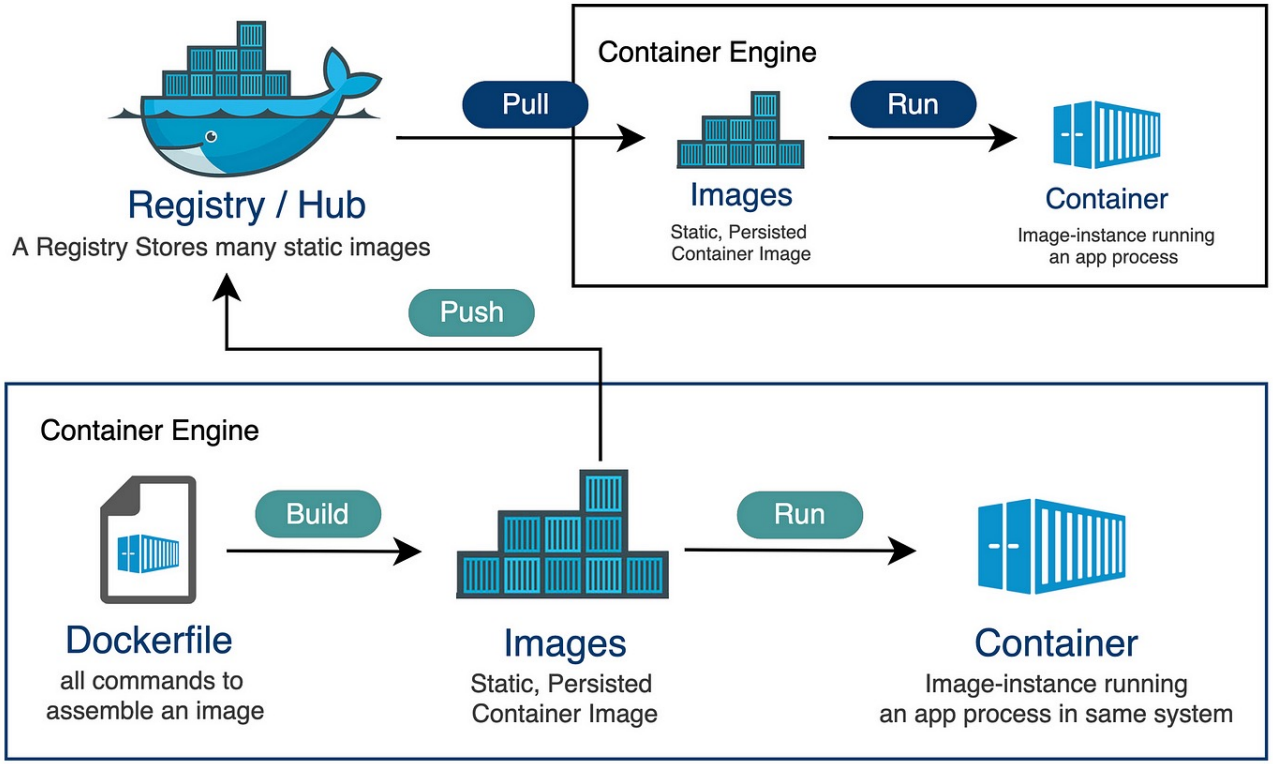
From CI → Docker Image

- CI turns source code into a deployable Docker image.
  - A Docker image is created using a Dockerfile.
  - The image is tagged and stored in a container registry.
  - The image becomes a deployable artifact.



From Docker Image → Deploy

- The server pulls the Docker image from the registry.
- A container is started from the image.
- The application runs consistently in production.
- Deployment is repeatable and predictable.



From Deploy → Monitoring & Feedback

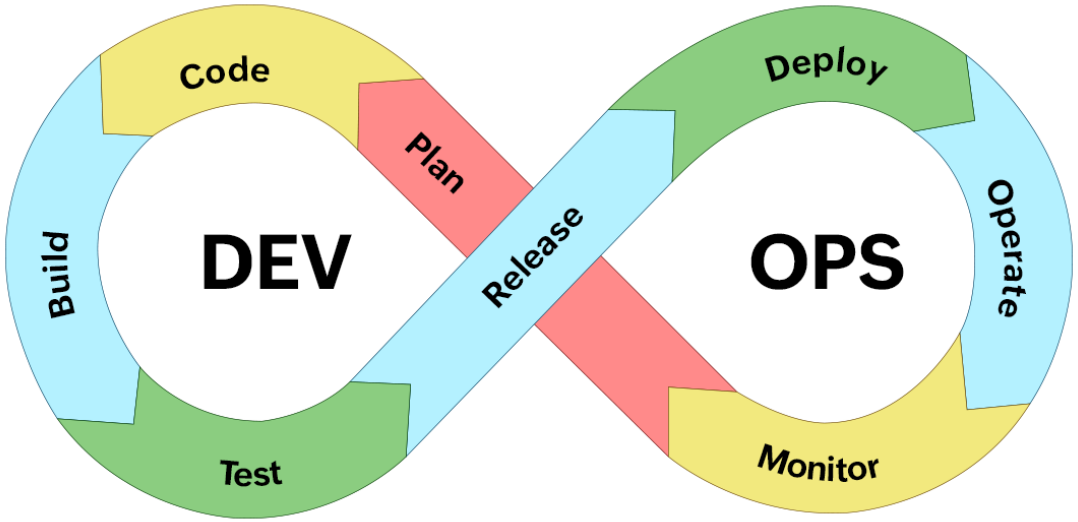
Deployment is not the end — it is the start of feedback and improvement.

- After deployment, the system is continuously monitored.
- Logs and metrics reveal errors, performance, and usage.
- Alerts notify the team before users complain.
- Feedback is used to improve the next release.

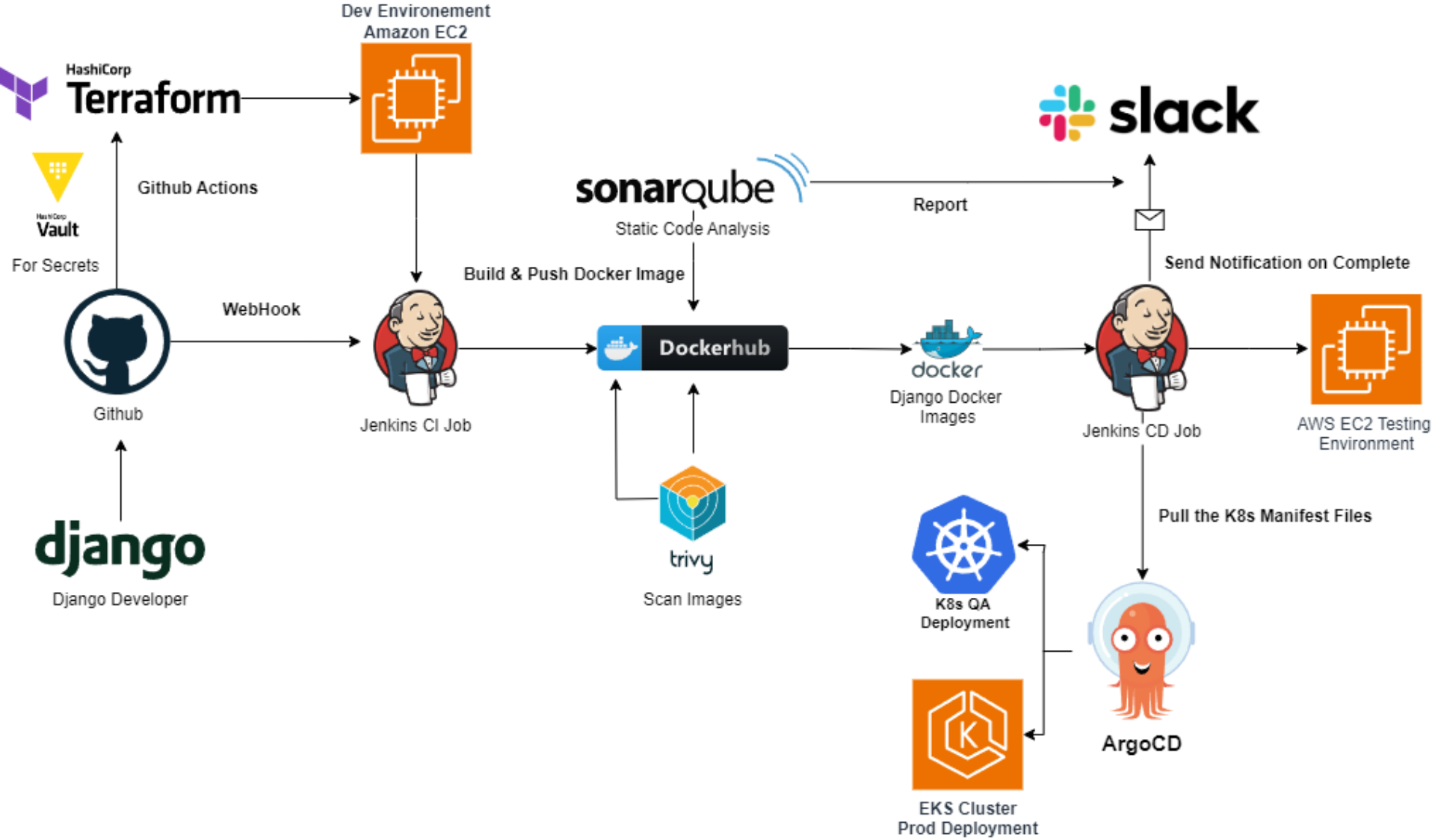
End-to-End DevOps Flow

Git Push → CI → Docker Image → Deploy → Monitoring → Feedback

- Code changes start the pipeline (Git push).
- CI builds, tests, and produces a Docker image.
- The image is deployed consistently to production.
- Monitoring and feedback close the loop and improve the next release.



End-to-End DevOps Flow (Example)



HUST

2. Docker & Environment Standardization

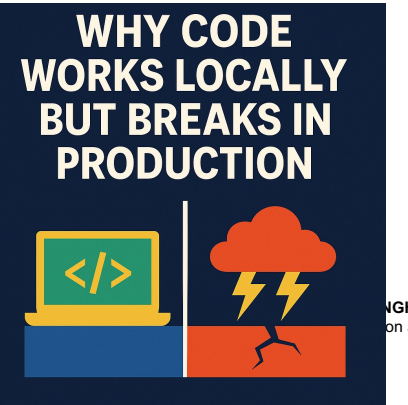
2.1. Docker for Web Applications

2.2. Dockerfile & Build Image

2.3. Docker Compose & Production Mindset

2.1. Docker for Web Applications

- The application works locally, but fails in production
  - Different environments
    - different OS, runtime versions (Node, Java)
    - missing or incompatible system libraries
  - Configuration & secrets environment
    - variables not set in production
    - hard-coded configs that only work locally
  - Infrastructure differences
    - local single process vs production services
    - database, network, or port differences



If the environment is not standardized,
the application behavior will not be predictable

How do we fix “works on my machine”?

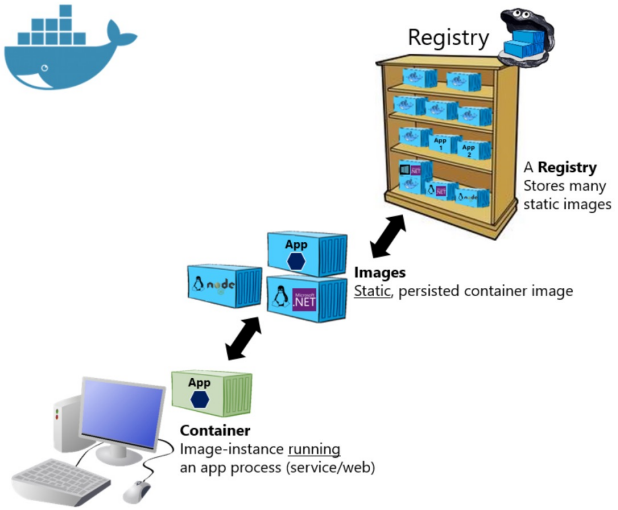
The solution is to **standardize the runtime environment**.

- Standardize the environment
  - use the same OS, runtime, and libraries everywhere
  - eliminate differences between local, test, and production
- Package everything together
  - Application code
  - Runtime (Node.js, Python, …)
  - Dependencies and system libraries
- Make same setup
  - No manual installation steps
  - Predictable behavior across environments

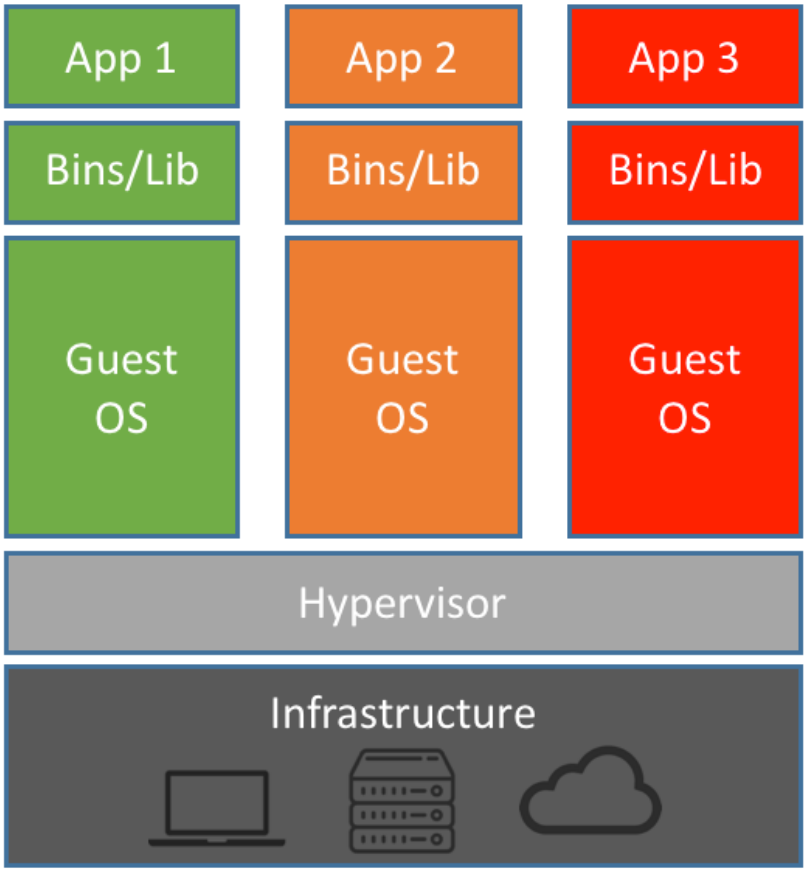
This is exactly the problem Docker was designed to solve

Docker Concepts: Image – Container – Registry

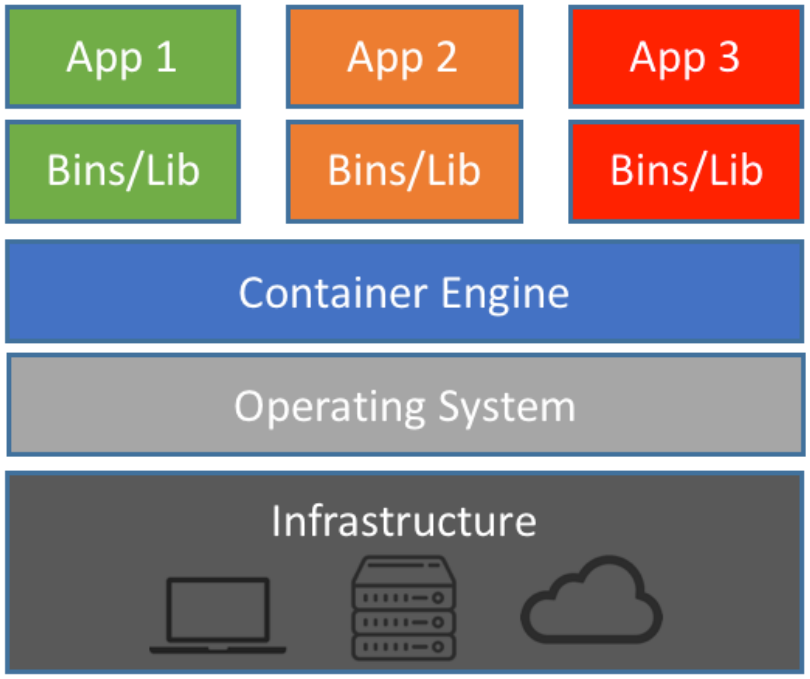
- Docker Image
  - A blueprint of the application
  - Contains code, runtime, libraries, and configuration
  - Immutable: built once, reused everywhere
- Docker Container
  - A running instance of an image
  - Isolated, lightweight, and reproducible
  - Multiple containers can run from the same image
- Docker Registry
  - A place to store and share Docker images
  - Examples: Docker Hub, GitHub Container Registry



Docker vs. Virtual Machine



Machine Virtualization



Containers

Docker vs. Virtual Machine

Aspect	Docker (Containers)	Virtual Machines
Virtualization level	Application-level	Hardware-level
Operating system	Shares host OS kernel	Each VM has its own OS
Startup time	Seconds	Minutes
Resource usage	Lightweight	Heavy
Isolation	Process-level	OS-level
Deployment speed	Very fast	Slower
Typical use	CI/CD, APIs	Full OS, legacy systems

Does Docker replace Virtual Machines?

2.2. Dockerfile & Build Image

- Dockerfile is a text file that defines how to build a Docker Image.
- It describes
  - Base environment (OS, runtime)
  - Application dependencies
  - Build & run commands
- Dockerfile standardizes how an application is built and deployed.
  - Ensures the app runs the same everywhere
  - Enables automation in CI/CD

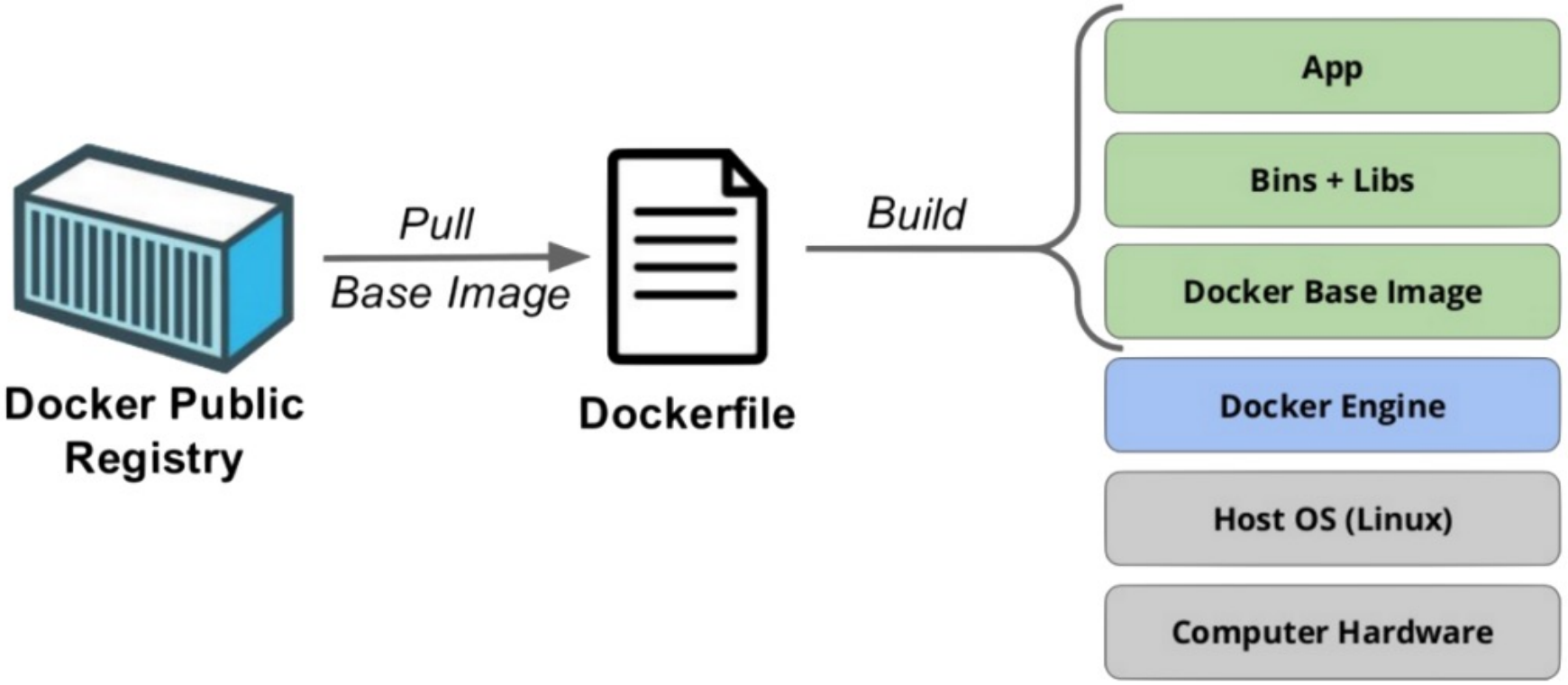
```
Source Code
     ↓
Dockerfile
     ↓
Docker Image
     ↓
Container
     ↓
Deployment (Server / Cloud)
```

Base Image & Best Practices

- What is a Base Image?
  - The starting point of a Docker image
  - Defines OS + runtime environment (e.g. node, python, nginx)
  - FROM node:20-alpine
- Dockerfile Best Practices
  - Use **specific versions** (avoid latest)
  - Install only what is needed
  - Clean cache & temp files
- A good base image improves security, performance, and reliability.

Why do we need base image?

Base Image & Best Practices



Build Cache & Layer Order

- Docker builds images **layer by layer**
- Reuses cache if the layer **does not change**

⇒ put the least-changing steps first to maximize Docker build cache

⇒ dependency installation must come before copying application code

- Example

```
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "app.js"]
```

Docker File Example

```
FROM node:20-alpine AS builder    #Build stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM node:20-alpine               #Run stage
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules
./node_modules
CMD ["node", "dist/index.js"]
```

2.3. Docker Compose & Production Mindset

- Docker Compose?
  - Tool for **defining and running multi-container Docker applications**
  - Uses a single YAML configuration file
  - Manage dependencies between services
  - One command to start the whole stack

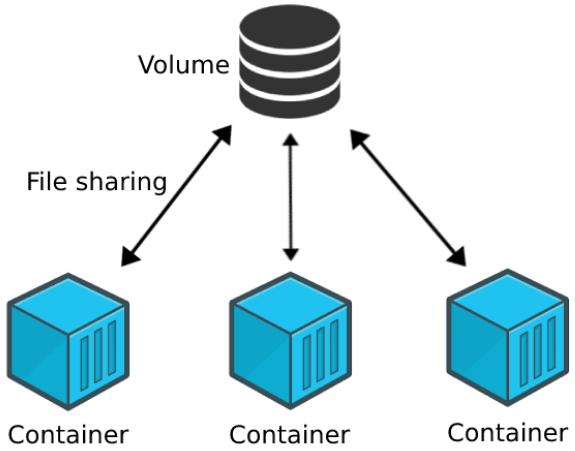
⇒ Perfect for backend + database

```yaml
version: "3.9"

services:
  backend:
    image: node:20-alpine
    command: ["node", "app.js"]
    ports:
      - "3000:3000"
    depends_on:
      - db

  db:
    image: postgres:15
```

Network & Volume

- **Network**
  - Allows containers to **communicate with each other**
  - Each service has a **service name** as hostname
  - Containers run in an **isolated virtual network**
  - Example: backend → db

- **Volume**
  - Used to **persist data**
  - Data survives container restart
  - Commonly used for **databases**



Restart Policy & Healthcheck

- Healthcheck detects problems, restart policy handles recovery.
- Healthcheck runs periodically
- Restart Policy: defines what Docker do when a container stops
  - no – never restart
  - always – always restart
  - unless-stopped – restart unless manually stopped
  - on-failure – restart only on error

test: ["CMD", "wget", "-qO-", "http://localhost:3000/health"]

=> wget -qO- http://localhost:3000/health

HUST

3. CI/CD Pipelines with GitHub Actions

3.1. CI Fundamentals

3.2. Docker in CI

3.3. CD Fundamentals

3.1. CI Fundamentals

- CI/CD: is a set of practices that automate the process from code changes to deployment.
- Continuous Integration (CI)
  - Merge code into a shared repository
  - Automatically build and test every change
  - Detect problems early
- Continuous Delivery / Deployment (CD)
  - Automatically prepare and deploy software
  - Continous Delivery: Deployment is manual (Delivery)
  - Continous Deployment: Deployment is automatic (Deployment)

GitHub Actions Overview

- GitHub Actions is a CI/CD platform built into GitHub.
- It automates build, test, and deployment workflows.
- Workflows are defined as YAML files in the repository.
- Actions run automatically on events (push, pull request, release).
- Components
  - Workflow: the CI/CD pipeline
  - Job: a set of steps
  - Step: an individual task
  - Runner: the machine that executes jobs

GitHub Actions turns GitHub events into automated CI/CD pipelines.

Workflow – Job – Step

- Workflow: the entire CI/CD pipeline
  - defined in a YAML file
  - triggered by GitHub events
- Job:
  - a set of tasks executed on the same runner
  - Jobs can run in parallel or sequentially
- Step:
  - an individual action or command
  - examples: checkout code, install dependencies, run tests
- Relationship Workflow → Job → Step
- A workflow is the pipeline, jobs are stages, and steps are actions.

Triggers & Secrets

- Triggers: define when a workflow runs
  - push – when code is pushed
  - pull request – when request is opened/updated
  - workflow_dispatch – manual trigger
- Secrets: store sensitive information securely
  - API keys
  - Database passwords
  - Not stored in source code

Triggers decide when CI runs,
Secrets protect what should not be exposed.

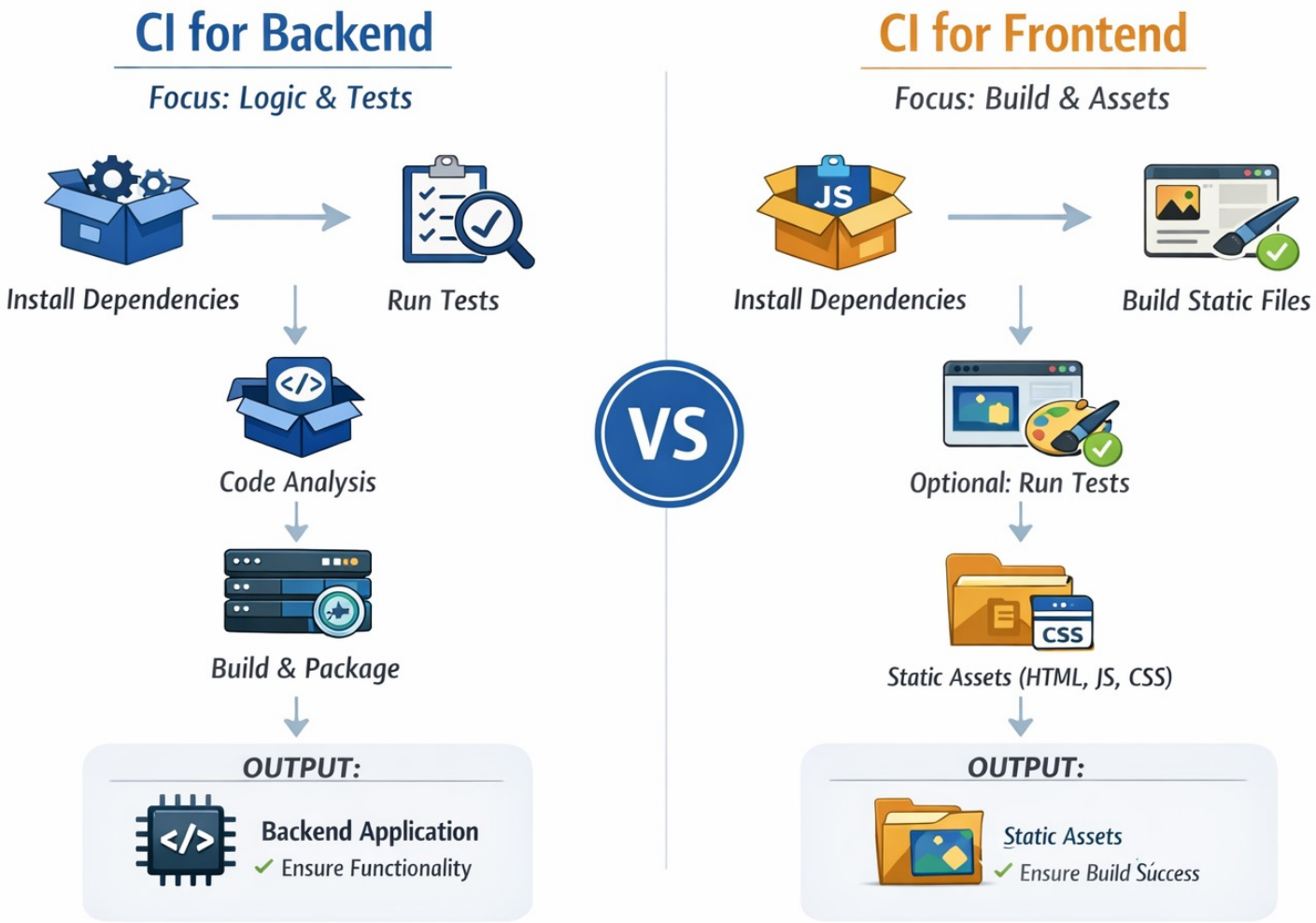
```yaml
on:
  push:
    branches: [ "main" ]
```

```yaml
env:
  DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
```

Example - YAML configuration file

```yaml
name: Simple CI

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout source code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test
```

CI: Backend vs Frontend



3.2. Docker in CI

- CI turns source code into a versioned Docker image ready for deployment.
- Code → CI → Docker Build → Image Tag → Registry

Image Tagging

- Tag identifies a specific version of a Docker image
- Same image name, different tags = different versions
- Common Tagging Strategies
  - Version-based, `myapp:1.2.0` => Good for releases
  - Git commit-based (Recommended)
    - `myapp:sha-3f9a1c2` => Perfect for CI/CD & rollback
  - Environment-based, `myapp:staging`, `myapp:prod`
    - Easy to understand
    - ❌ Not immutable
  - latest (Use carefully), `myapp:latest`
    - Always points to newest image
    - ❌ Hard to debug

Container Registry

- A container registry stores and distributes Docker images.
- CI pipelines push images to the registry after a successful build.
- Servers and platforms pull images from the registry to deploy.
- Common registries: Docker Hub, GitHub Container Registry (GHCR).

A registry is the bridge between CI and production deployment.

3.3. CD Fundamentals

- CD (Continuous Delivery / Continuous Deployment) is the practice of automatically delivering code to production-ready environments.

Level	Deploy Trigger	Risk
Manual	Human	Low
Continuous Delivery	Approval	Medium
Continuous Deployment	Automatic	High

Deploy Containers to a Server

- The server pulls the Docker image from the registry.
- A container is started from the image.
- Configuration is provided via environment variables.
- The same image runs consistently in production.
- Deploy = pull image + run container

Registry → Server → Container

Example

```
name: Simple CI/CD
on: { push: { branches: [ "main" ] } }
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: docker/login-action@v3
        with: { username: ${{ secrets.DOCKER_USERNAME }},
password: ${{ secrets.DOCKER_PASSWORD }} }
      - run: docker build -t ${{ secrets.DOCKER_USERNAME }}/myapp .
      - run: docker push ${{ secrets.DOCKER_USERNAME }}/myapp
      - uses: appleboy/ssh-action@v1.0.0
        with: { host: ${{ secrets.SERVER_HOST }}, username: ${{
secrets.SERVER_USER }}, key: ${{ secrets.SERVER_SSH_KEY }},
script: docker pull ${{ secrets.DOCKER_USERNAME }}/myapp &&
docker rm -f myapp || true && docker run -d --name myapp -p
80:3000 ${{ secrets.DOCKER_USERNAME }}/myapp }
```

Restart & update container

- Restart Container
  - Restart is used to recover from crashes
  - Container restarts without rebuilding the image
  - `docker restart backend`

- Update Container
  - Build a new image
  - Replace the old container
  - Build new image → Stop old container → Run new container
  - `docker-compose build backend`
  - `docker-compose up -d backend`

HUST

4. Production Deployment and Operations

4.1. Nginx

4.2. IaaS, PaaS, SaaS

4.3. Monitoring

4.1. Nginx

- Why you should NOT expose backend directly
  - Exposing backend increases security risks (attacks, scans, exploits).
  - Backend cannot efficiently handle TLS, files
  - It makes scaling, routing, and maintenance harder
- *Backends should serve logic, not handle internet traffic directly*.
- Approach
  - Place a reverse proxy (e.g., Nginx) in front of the backend.
  - Let Nginx handle HTTPS, routing, caching, and load balancing.

What is Nginx

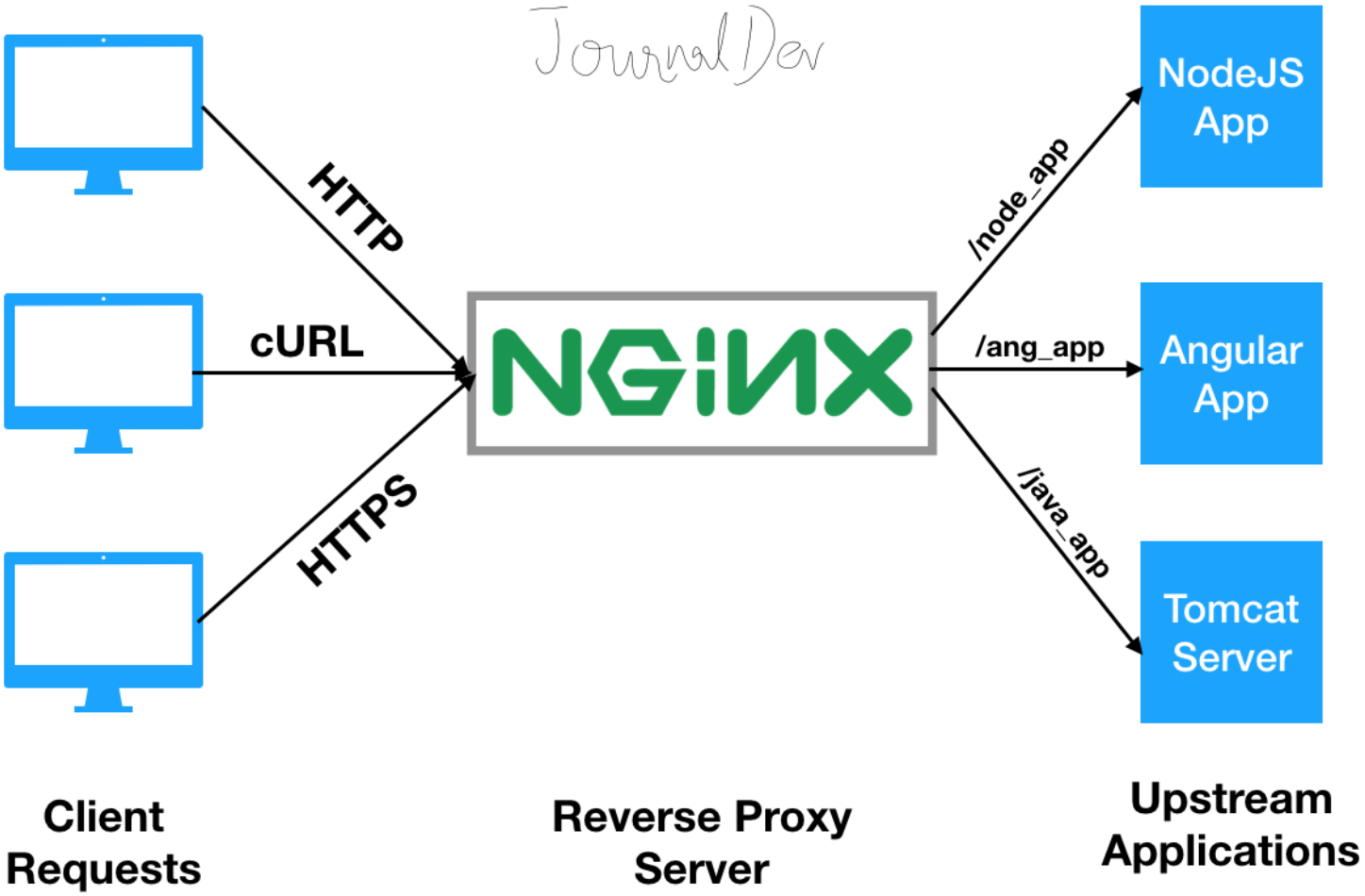
- Nginx is a high-performance web server and reverse proxy that sits in front of web applications.

=> Applications are **not exposed directly** to the Internet.

  - Reverse Proxy: routes requests to backend services
  - Static Content Server: serves HTML, CSS, JS, images efficiently
  - Load Balancer: distributes traffic across multiple backends
  - TLS Termination: handles HTTPS (SSL/TLS)

- Client → Nginx → Backend / Frontend App

Nginx - Reverse Proxy



Nginx serving Frontend

- **Serving static files**
  - Nginx efficiently serves **static frontend assets**: HTML, CSS, avaScript, Images
  - Nginx serves files **directly from disk**
  - Faster than backend server
- **SPA Problem**
  - Routes are handled **by JavaScript**
  - Direct access causes **404 error**

⇒Always return index.html

```
location / {
    root /usr/share/nginx/html;
    index index.html;
    try_files $uri $uri/ /index.html;
```

Nginx Configuration: API vs. Frontend

Aspect	API (Reverse Proxy)	Frontend (Static SPA)
Purpose	Forward requests to backend	Serve static files
Upstream	Backend service (e.g. Node.js)	No backend upstream
Typical path	/api/*	/
Main directive	proxy_pass	root / try_files
HTTPS handling	Yes	Yes
Common issue	Exposing backend port	SPA routing breaks

Nginx Configuration File

```
server {
    listen 80;
    # --- Frontend (SPA) ---
    location / {
        root /usr/share/nginx/html;
        index index.html;
        try_files $uri $uri/ /index.html;
    }

    # --- Backend API ---
    location /api/ {
        proxy_pass http://backend:3000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

4.2. IaaS, PaaS, SaaS

Layer	IaaS	PaaS	SaaS
Application	You	You	Provider
Runtime	You	Provider	Provider
OS	You	Provider	Provider
Server	You	Provider	Provider
Network	You	Provider	Provider
Example	AWS, Google, Azure	Render, Heroku, Vercel	Gmail, GitHub

PaaS

- PaaS (Platform as a Service) provides a managed platform to run applications.
- The platform handles servers, OS, runtime, scaling, and networking.
- Developers focus on code and configuration, not infrastructure.

PaaS lets developers deploy applications without managing servers.

What does PaaS do?

- From developers
  - Git push → App is live
  - Simple for developers
- PaaS automates what DevOps teams usually do manually
- PaaS = Docker
  + Nginx
  + CI/CD
  + Monitoring
  + Automation

Environment & Secrets on PaaS

- Configuration is set via environment variables.
- Secrets (API keys, DB passwords) are stored securely by the platform.
- No secrets in source code or Git.
- Each environment (dev/staging/prod) has its own values.

PaaS keeps configuration and secrets secure, separate from code.

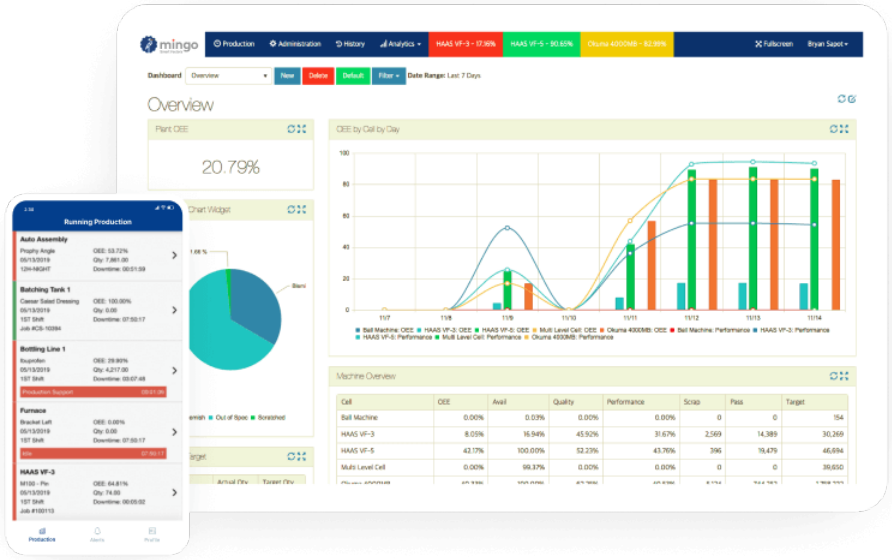
When to Use PaaS vs IaaS

Aspect	PaaS	iaaS (VPS)
Setup time	Minutes	Hours / days
DevOps effort	Low	High
Control	Limited	Full
Scaling	Automatic	Manual
Maintenance	Platform	You

Why Production needs monitoring

- Production is unpredictable
- Monitoring answers critical questions
- Detect problems early
- Enable continuous improvement

You cannot manage or improve what you cannot observe.



Minimum metrics to Monitor

- System Metrics
  - CPU usage
  - Memory usage
  - Disk space
- Application Metrics
  - Response time (latency)
  - Error rate (4xx / 5xx)
  - Request throughput
- Alerts
  - Notify when something is wrong
  - Triggered by metric thresholds or errors
- Logs
  - Application errors
  - Unexpected crashes
  - Startup / shutdown events

Metrics tell you there is a problem,
alerts wake you up,
logs tell you why.