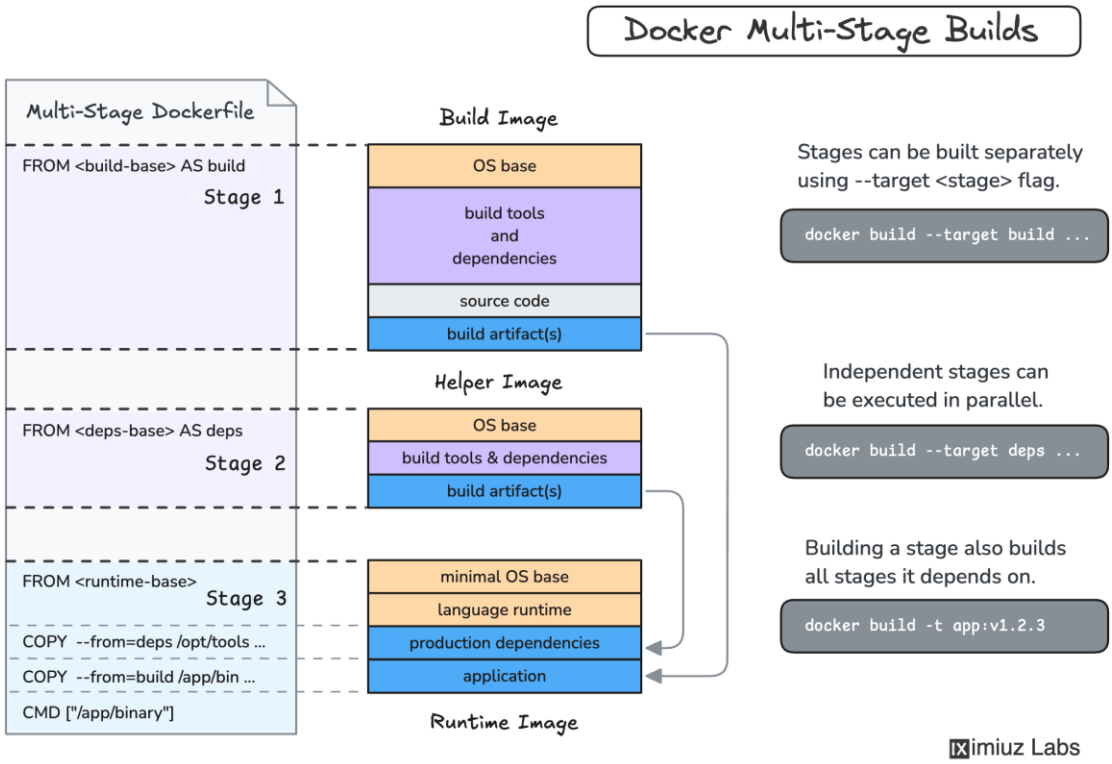
Module 1: Advanced Docker — Detailed Notes for Participants

1. Multi-Stage Builds & Image Optimization



```
# Base image Stage 1
FROM ubuntu:16.04 as stage1

RUN apt-get update
RUN apt-get -y install make curl
RUN curl http://xyz.com/abc.tar.gz -O
RUN tar zxf abc.tar.gz && cd abc
RUN make DESTDIR=/tmp install

# Stage 2
FROM alpine:3.10

COPY --from=stage1 /tmp /abc

ENTRYPOINT ["/abc/app"]
```

1.1 Why Multi-Stage Builds?

Multi-stage builds help you:

- **Reduce image size**
- **Speed up deployments**
- **Improve security** by excluding unnecessary dev tools
- **Separate build & runtime environments**

Traditional Dockerfiles produced large images because they included compilers, dependencies, and runtime components in a single image.

1.2 How Multi-Stage Builds Work

You use multiple FROM statements in the same Dockerfile—each stage creates intermediate images.

Example:

Stage 1: Builder

FROM golang:1.21 AS builder

```
WORKDIR /app

COPY . .

RUN go build -o app .


# Stage 2: Runtime

FROM alpine:3.18

WORKDIR /app

COPY --from=builder /app/app .

CMD ["./app"]
```

1.3 Benefits

- Final image contains **only what is required** to run the application.
- No need to install compilers in the runtime image.
- Helps in **CI/CD pipelines** where speed and security matter.

1.4 Best Practices

- Use **Alpine** or other lightweight base images.
- Use .dockerignore to avoid copying unnecessary files.
- Pin versions: node:20-alpine instead of node:latest.

---

2. Docker Compose for Production

Docker Compose is not only for local development. With proper configurations, it works well for **small/medium-scale production** setups.

2.1 YAML Structure Refresher

A Compose file (docker-compose.yml) describes:

- Services
- Networks
- Volumes
- Dependencies
- Environment variables
- Build settings

2.2 Example Production-Ready Compose

```
version: "3.9"

services:
  web:
    image: myapp:1.0.0
    ports:
      - "80:80"
    environment:
      - APP_ENV=prod
    depends_on:
      - db
    restart: unless-stopped
    networks:
      - app-net

  db:
    image: postgres:15
    volumes:
      - dbdata:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: StrongP@ss
    networks:
      - app-net

volumes:
  dbdata:

networks:
  app-net:
```

2.3 Production Tips

- Always use **restart policies** (e.g., on-failure, unless-stopped)
- Store secrets using **Docker secrets**, not environment variables.
- Use **named volumes** to persist data.
- Configure **healthcheck** to restart unhealthy containers.

Example:

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost/health"]

interval: 30s

timeout: 10s

retries: 3

2.4 Scaling

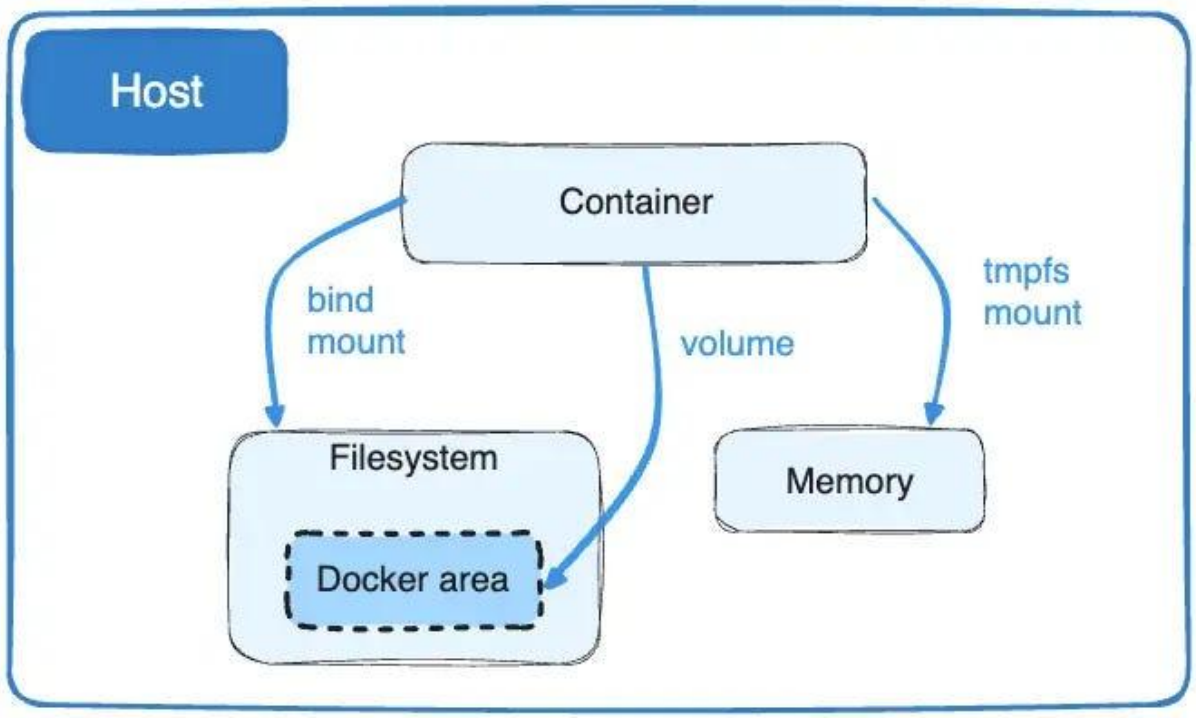
Use:

docker compose up --scale web=3 -d

This launches **horizontal scaling**, useful for load balancing.

---

3. Docker Volumes, Networking & Secrets Management



3.1 Docker Volumes

Volumes allow data persistence.

Types of volumes

1. **Named volumes** (Recommended)
2. **Anonymous volumes**
3. **Bind mounts** (For development)

Example:

docker volume create appdata

docker run -v appdata:/data alpine

When to use volumes

- Databases
- Config files
- Logs that must persist across restarts

3.2 Docker Networking

Docker provides 3 major network types:

Network Type	Description	Use Case
bridge	Default network	Multi-container apps
host	Shares host network	High performance apps
overlay	Multi-host networking	Docker Swarm/K8s

Inspect networks

docker network ls

docker network inspect bridge

Create a custom network

docker network create mynet

docker run --network=mynet nginx

Benefits:

- Better **isolation**
- DNS-based **service discovery**
- Container-to-container communication

3.3 Docker Secrets Management

Secrets include:

- API keys
- DB passwords
- TLS certificates
- OAuth tokens

Why not store secrets in environment variables?

- They appear in container metadata.
- Anyone with Docker access can view them.
- They may be logged accidentally.

Secure method (Docker Swarm required)

echo "SecretP@ss" | docker secret create db_pass -

docker service create --name myapp --secret db_pass nginx

Inside container, secrets are mounted at:

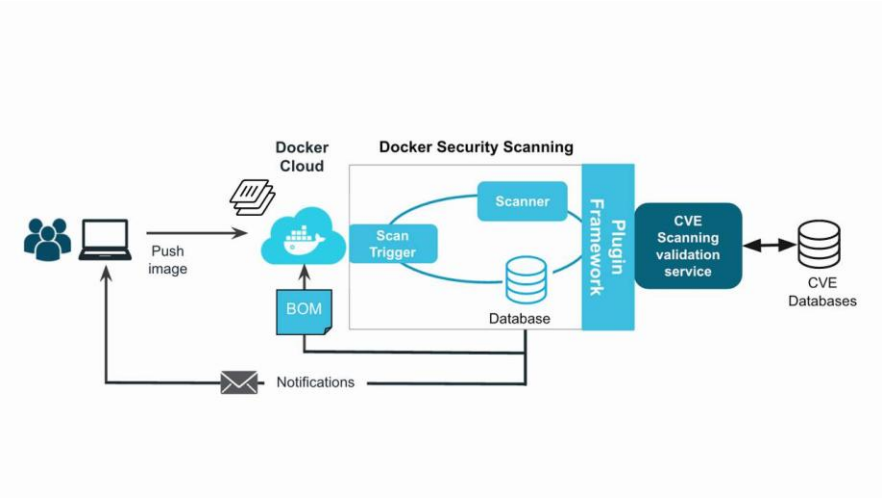
/run/secrets/db_pass

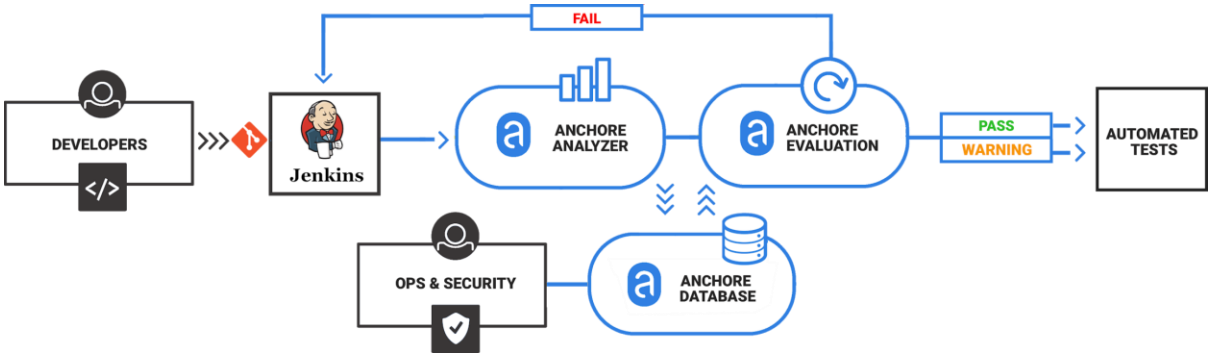
Best Practices

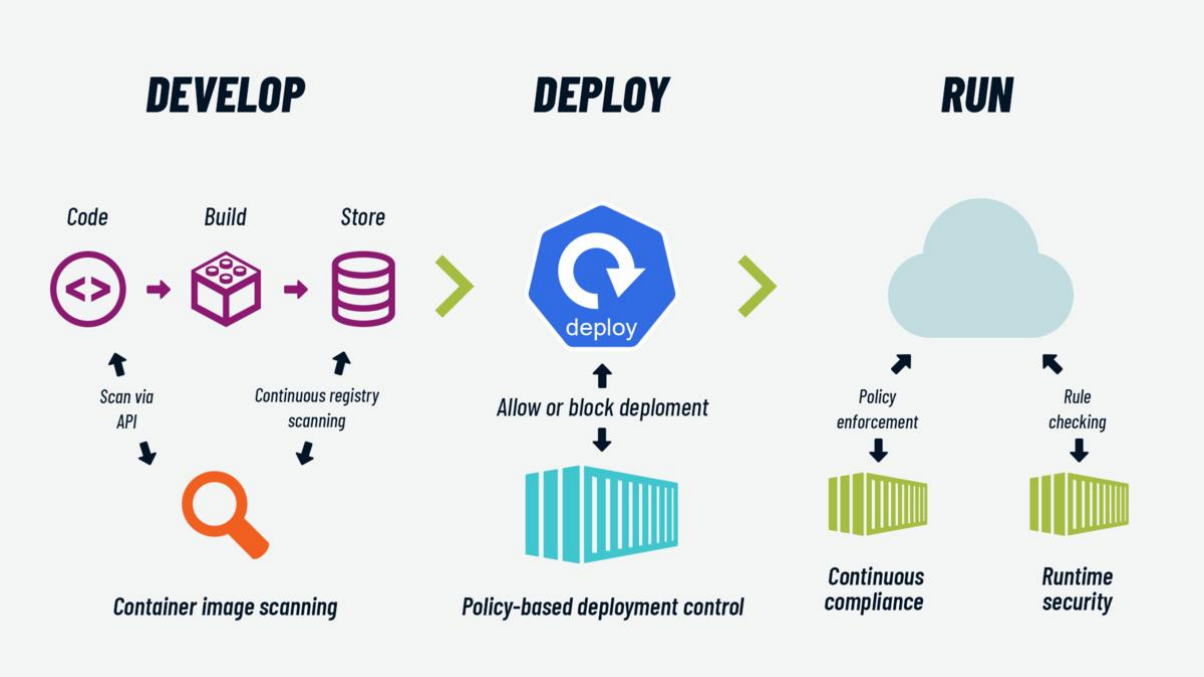
- Rotate secrets regularly.
- Never commit secrets to Git.
- Use .env + secrets manager (AWS KMS, Vault, Docker Swarm).

---

4. Scanning & Securing Docker Images







6

4.1 Why Scan Docker Images?

Containers often include:

- CVEs (Common Vulnerabilities & Exposures)
- Outdated libraries
- Misconfigured permissions
- Hardcoded secrets

4.2 Popular Scanning Tools

Tool	**Purpose**
Trivy	Vulnerabilities + misconfig + secrets
Grype	Fast CVE scanner
Clair	Registry-level scanning

Tool	Purpose
Docker Scout	Built into Docker Hub
Anchore	CI/CD scanning

4.3 Example: Scan with Trivy

Install:

sudo apt install trivy

Scan an image:

trivy image nginx:latest

Scan file system for secrets:

trivy fs --scanners secret .

4.4 Security Best Practices

- Use **official images** whenever possible.
- Always pin versions: python:3.10-slim.
- Never run containers as **root** (use USER 1000).
- Use **read-only root filesystem**:

read_only: true

- Enable **seccomp**, **AppArmor**, **SELinux**.
- Limit container capabilities:

cap_drop:

- ALL

- Use **image signing** with cosign.

4.5 CI/CD Integration

Add scanning steps in pipeline:

- GitHub Actions
- GitLab CI
- Jenkins
- Azure DevOps

Example (Jenkins):

trivy image --exit-code 1 myapp:1.0.0

If vulnerabilities found, build fails.

✓ Module 1 Summary

Topic	Quick Recap
Multi-Stage Builds	Reduce image size, split build/runtime
Docker Compose Production	Scaling, healthchecks, restart policies
Volumes	Persistent storage
Networking	Custom networks, DNS-based discovery
Secrets	Secure storage using Swarm/KMS/Vault
Scanning	Use Trivy/Grype to identify CVEs, misconfigs, secrets