
Fast Lane

Advanced Docker Refresher

www.flane.co.uk

Ivan Pranjic | Fast Lane | 14-11-2023

Day 1: Building and Running Advanced Workloads

Topic	Description
Optimize image building process	Introduction to Multi-Stage Builds Use cases for multi-stage builds Creating optimized Docker images for a Java Spring / Dotnet application **Hands-on exercise:** Creating Dockerfiles for building multi-stage Docker images for a Java Spring / Dotnet application
Data persistence and volume management	Understanding data persistence in Docker Working with volumes and bind mounts Docker data management best practices **Hands-on exercise:** Managing volumes and data in Docker containers
Advanced Docker Networking	Host network use-cases and scenarios Creating and managing multiple bridge networks Inter-container communication using custom networks **Hands-on exercise:** Configuring advanced Docker networking scenarios
Docker Compose	Introduction to Docker Compose Writing Docker Compose files for multi-container applications Orchestrating multi-container apps with Compose **Hands-on exercise:** Creating and running multi-container apps using Docker Compose

Multi Stage builds in Docker

1. What are Multi-Stage Builds?
2. Anatomy of a Multi-Stage Dockerfile
3. Benefits of Multi-Stage Builds
4. Practical Example
5. Building Efficient Images
6. Use Cases
7. Security Considerations
8. Tips and Tricks

What are Multi-Stage Builds?

Multiple build stages

```
Dockerfile

# Build a single image

FROM base
COPY --from=development /app /app
RUN build_one_image
```

Image

What are Multi-Stage Builds?

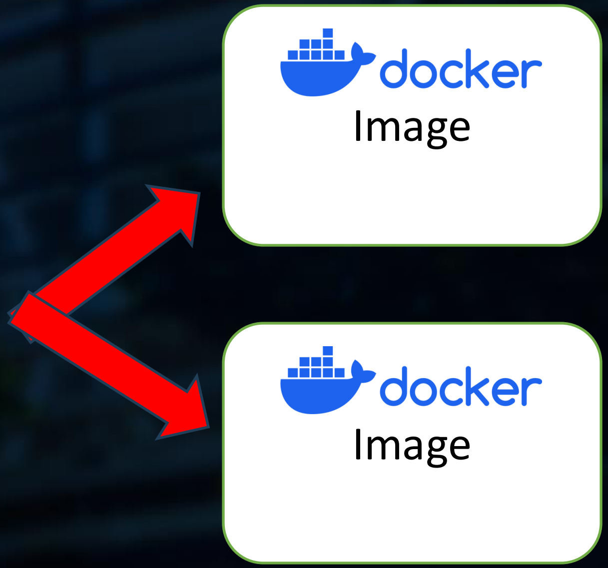
Multiple build stages

```
Dockerfile

# Stage 1: Build for development

FROM base as development
RUN build_for_development

# Stage 2: Final stage for production
FROM base as production
COPY --from=development /app /app
RUN build_for_production
```



Benefits of multi-stage builds

Smaller images

Faster Build images

Enhanced Security

Easier maintenance

Image

Benefits of multi-stage builds

Smaller images

Faster Build images

Enhanced Security

Easier maintenance

Common use-case scenarios

- Building applications that require complex dependencies, like compilers, libraries, or transpilers.
- Compiling code written in one language and running it in a different runtime environment.
- Creating production-ready images from source code and build artifacts.

```
# Stage 1: Build the Go application
FROM golang AS build
WORKDIR /app
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp app.go

# Stage 2: Run the compiled Go binary in a Ruby environment
FROM ruby:2.7
WORKDIR /app
COPY --from=build /app/myapp .
CMD ["ruby", "-e", "system('./myapp')"]
```

Common use-case scenarios - Basic

```
# Stage 1: Build the applications
FROM golang:1.17 AS go-build
WORKDIR /go-app
COPY main.go .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp-go main.go

FROM openjdk:11 AS java-build
WORKDIR /java-app
COPY Main.java .
RUN javac Main.java

FROM gcc:11 AS cpp-build
WORKDIR /cpp-app
COPY main.cpp .
RUN g++ -o myapp-cpp main.cpp

# Stage 2: Create a smaller final stage
FROM alpine:3.14
WORKDIR /app

# Copy the compiled applications from the previous stages
COPY --from=go-build /go-app/myapp-go .
COPY --from=java-build /java-app/Main.class .
COPY --from=cpp-build /cpp-app/myapp-cpp .

# Set the entry point based on the application type
CMD ["/bin/sh", "-c", "./$(ls | grep myapp)"]
```

Compile and Package Applications

Common use-case scenarios - Basic

```
# Stage 1: Build the frontend assets
FROM node:14 AS builder
WORKDIR /app

# Copy the package.json and package-lock.json to install dependencies
COPY package*.json ./
RUN npm install

# Copy the application source code
COPY . .

# Build the frontend assets using a build script (e.g., webpack)
RUN npm run build

# Stage 2: Create a smaller final stage
FROM nginx:alpine
WORKDIR /usr/share/nginx/html

# Copy the built frontend assets from the previous stage
COPY --from=builder /app/build/ .

# Nginx will serve the static assets
EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

```
my-frontend-app/
  └── src/
  └── package.json
  └── package-lock.json
  └── webpack.config.js
```

Building Frontend Assets

Common use-case scenarios - Basic

```
# Stage 1: Download and cache dependencies
FROM node:14 AS dependencies
WORKDIR /app

# Copy the package.json and package-lock.json to install dependencies
COPY package*.json ./
RUN npm install

# Stage 2: Create a smaller final stage
FROM node:14
WORKDIR /app

# Copy only the required files from the dependencies stage
COPY --from=dependencies /app/node_modules ./node_modules
COPY ./src ./src

# Set the command to run the application (adjust as needed)
CMD ["node", "src/index.js"]
```

```
my-nodejs-app/
  └── package.json
  └── package-lock.json
  └── src/
  └── other_files/
```

Dependency Management

Common use-case scenarios - Basic

```
# Stage 1: Transpile TypeScript to JavaScript
FROM node:14 AS builder
WORKDIR /app

# Copy the package.json and package-lock.json to install dependencies
COPY package*.json ./
RUN npm install

# Copy the TypeScript source code
COPY . .

# Build the TypeScript code to JavaScript
RUN npm run build

# Stage 2: Create a smaller final stage
FROM node:14
WORKDIR /app

# Copy the transpiled JavaScript files from the previous stage
COPY --from=builder /app/dist/ .

# Set the command to run the application (adjust as needed)
CMD ["node", "index.js"]
```

```
my-typescript-app/
  └── src/
  └── package.json
  └── tsconfig.json
```

Language Transpilation

Common use-case scenarios - Basic

```
# Stage 1: Build Service 1
FROM node:14 AS service1-builder
WORKDIR /app

# Copy the package.json and package-lock.json to install dependencies
COPY package*.json ./
RUN npm install

# Copy the Service 1 source code
COPY src/ .

# Build Service 1 (e.g., transpile TypeScript, compile binaries, etc.)
RUN npm run build

# Stage 2: Create a smaller final stage for Service 1
FROM node:14
WORKDIR /app

# Copy only the required files for Service 1 from the builder stage
COPY --from=service1-builder /app/node_modules ./node_modules
COPY --from=service1-builder /app/dist ./dist

# Set the command to run Service 1 (adjust as needed)
CMD ["node", "dist/index.js"]
```

```
my-microservices-app/
  ├── service1/
  |    ├── Dockerfile
  |    ├── package.json
  |    ├── src/
  ├── service2/
  |    ├── Dockerfile
  |    ├── package.json
  |    ├── src/
```

```
# Stage 1: Build Service 2
FROM python:3.8 AS service2-builder
WORKDIR /app

# Copy the Service 2 source code
COPY . .

# Build Service 2 (e.g., compile, generate assets, etc.)
RUN python build.py

# Stage 2: Create a smaller final stage for Service 2
FROM python:3.8
WORKDIR /app

# Copy only the required files for Service 2 from the builder stage
COPY --from=service2-builder /app/build ./build

# Set the command to run Service 2 (adjust as needed)
CMD ["python", "build/main.py"]
```

Multi-component Applications

Common use-case scenarios - Advanced

Cross-Compiling

```
# Install Buildx
docker buildx create --use
docker buildx inspect --bootstrap

# Build for multiple platforms
docker buildx build --platform linux/amd64,linux/arm64 -t your-image-name:tag .
```

Common use-case scenarios - Advanced

Complex Dependency Management

```
FROM golang:1.16 AS builder

WORKDIR /app
COPY go.mod .
COPY go.sum .
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp .

FROM scratch
COPY --from=builder /app/myapp /myapp
CMD ["/myapp"]
```

Common use-case scenarios - Advanced

Artifact Testing

```yaml
version: '3'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.test
    depends_on:
      - main
  main:
    build:
      context: .
      dockerfile: Dockerfile
```

Common use-case scenarios - Advanced

Optimized Image Layers

```
FROM node:14 as builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

Common use-case scenarios - Advanced

Custom Build Environments

```
# Dockerfile.build
FROM node:14 as builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

# Main Dockerfile
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

Common use-case scenarios - Advanced

Multi-Service Integration

```yaml
version: '3'
services:
  web:
    image: nginx:alpine
  api:
    image: your-api-image:tag
    depends_on:
      - db
  db:
    image: postgres:latest
```

Common use-case scenarios - Advanced

Security Hardening

```
FROM node:14 as builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html

# Additional security measures
RUN adduser -D nonrootuser
USER nonrootuser
```

Anatomy of a multi-stage Dockefile

```
# Stage 1: Build
FROM base_image AS build

# Set working directory
WORKDIR /app

# Copy source code to the container
COPY src/ .

# Run build commands
RUN some_build_command

# Stage 2: Test
FROM base_image AS test

# Copy artifacts from the build stage
COPY --from=build /app/build /app/build

# Run tests
RUN some_test_command

# Stage 3: Final
FROM base_image AS final

# Copy artifacts from the build and test stages
COPY --from=build /app/build /app/build
COPY --from=test /app/test_results
/app/test_results

# Configure the runtime environment
ENV ENV_VARIABLE=value

# Define the entry point
CMD ["app_executable"]
```

Benefits of a Multi-stage Dockerfile

- Smaller image size
- Faster Build Times
- Improved Security
- Simplified Build Process
- Enhanced Dependency Management

Practical Example on multi stage Dockerfiles

Single-Stage Dockerfile:

```
# Use a Java base image
FROM openjdk:11
# Set the working directory
WORKDIR /app
# Copy the Java source code to the container
COPY my-java-app/ .
# Compile the Java application
RUN javac HelloWorld.java
# Run the application CMD ["java", "HelloWorld"]
```

Multi-Stage Dockerfile:

```
# Stage 1: Build
FROM openjdk:11 AS build
# Set the working directory
WORKDIR /app
# Copy the Java source code to the container
 COPY my-java-app/ .
# Compile the Java application
RUN javac HelloWorld.java

# Stage 2: Final
 FROM openjdk:11
# Set the working directory
WORKDIR /app
# Copy the compiled application from the build stage
COPY --from=build /app/HelloWorld.class .
 # Run the application
 CMD ["java", "HelloWorld"]
```

Building efficient images

- Organizing and Ordering Stages
- Copy Only Necessary Files
- Use Build Arguments and Build Context
- Clean Up in Each Stage
- Security Considerations

When to use multi-stage builds

- Building Python Applications
- Web Servers
- Microservices
- Golang Applications
- .NET Applications
- Cross-Compiling and Multi-Architecture Support
- Dependency-Heavy Applications

When NOT to use multi-stage builds

- Simple and Lightweight Applications
- Monolithic Applications
- Rapid Prototyping and Development

Security Considerations

- Reduced Attack Surface
- Isolation of dependencies
- Temporary Build Containers
- Implement Access Control
- Scan for Vulnerabilities
- Keep Dockerfiles Clean
- Avoid Secrets in Dockerfiles
- Secure Docker Hosts
- Implement Network Security

Tips and tricks

1. Use Build Arguments
2. Conditional Stages
3. Leverage .dockerignore
4. Caching Optimization
5. Use Multi-architecture Support
6. Combine Multiple COPY Commands
7. Use .dockerignore to Optimize Copying
8. Minimize Commands and Layer Sizes

Tips and tricks

Use Build Arguments

```
# Dockerfile
ARG BASE_IMAGE=alpine:latest

FROM $BASE_IMAGE as builder
# Build stage commands

FROM another-image:latest
# Final stage commands
```

```
docker build --build-arg BASE_IMAGE=nginx:latest -t my-custom-image:tag .
```

Conditional Stages

```
# Dockerfile
FROM base-image as builder
# Build stage commands

# Only include the test stage if the TEST_BUILD argument is set
FROM tester-image as tester
ARG TEST_BUILD
RUN [ "$TEST_BUILD" = "true" ] && run_tests_command || echo "Tests skipped"

FROM production-image:latest
# Final stage commands
```

```
docker build --build-arg TEST_BUILD=true -t my-image-with-tests:tag .
```

Tips and tricks

Caching Optimization

```
# Dockerfile
FROM base-image as builder
WORKDIR /app
COPY . .
RUN npm install

FROM base-image
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY . .
CMD ["npm", "start"]
```

Tips and tricks

Use Multi-architecture Support

```
docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t multi-arch-image:tag .
```

Combine Multiple COPY Commands

```
# Dockerfile
FROM base-image as builder
WORKDIR /app
COPY file1 .
COPY file2 .
COPY file3 .
# Combine multiple COPY commands into one
COPY file4 file5 file6 .

FROM final-image
COPY --from=builder /app /app
```

Tips and tricks

Use .dockerignore to Optimize Copying

```
# .dockerignore
node_modules
*.log
secret-key
```

Minimize Commands and Layer Sizes

```
# Dockerfile
FROM base-image as builder
WORKDIR /app
COPY . .
RUN npm install && npm run build

FROM final-image
WORKDIR /app
COPY --from=builder /app/dist ./dist
CMD ["npm", "start"]
```

Data Persistance

Data Integrity → Continuous Availability → Data Durability → Data Recovery

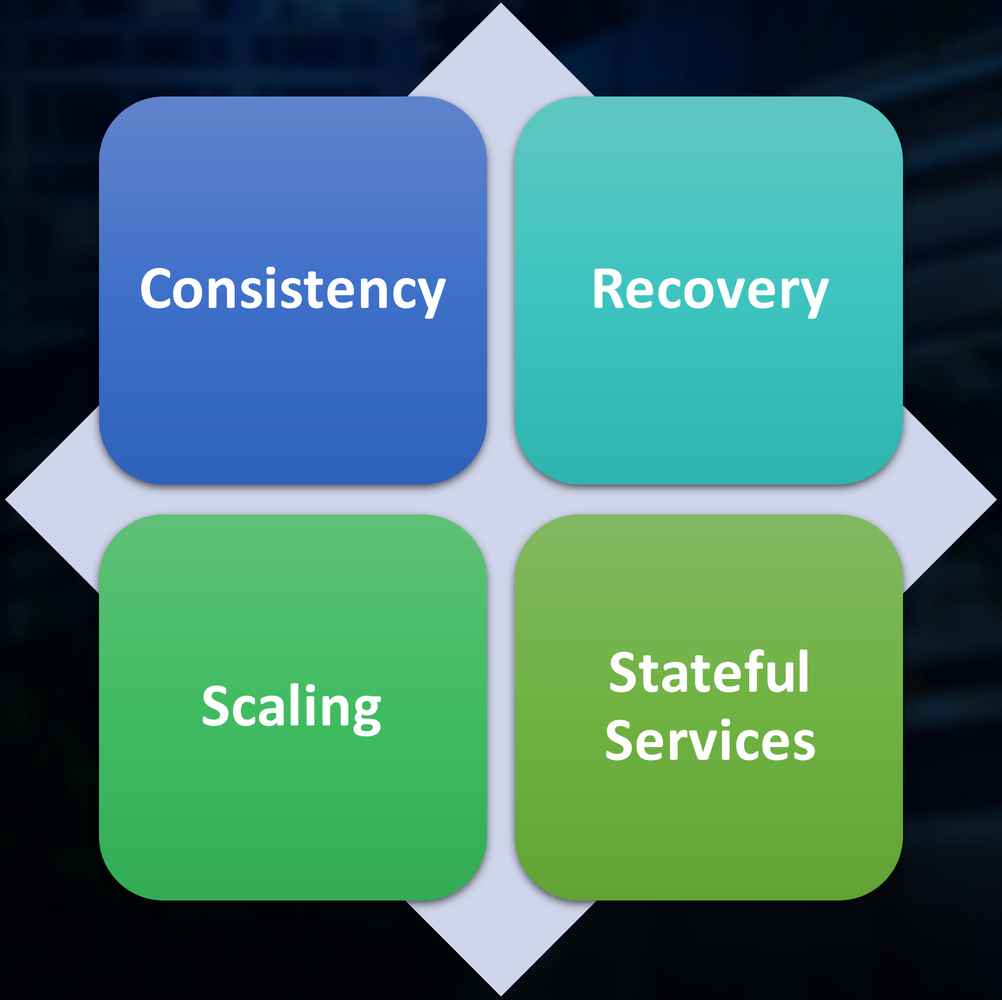
Challenges of Data Persistance

Ephemeral Containers

File System Isolation

Data Lifecycle

Need for data durability



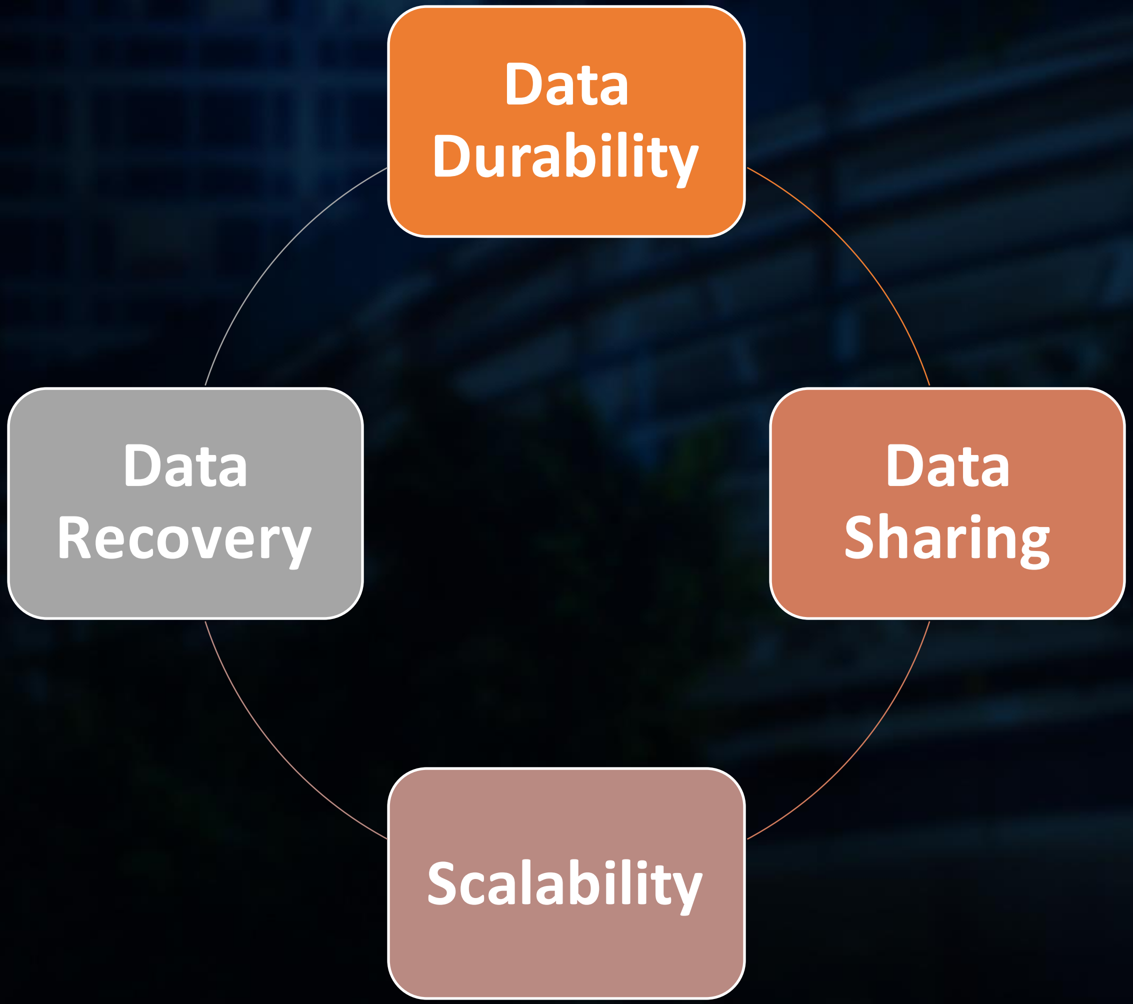
Data Handling in Containers

Ephemeral Nature of Container File Systems

Data Inside Containers

Data Handling in Containers

Storing Important Data in Containers Is Generally Discouraged



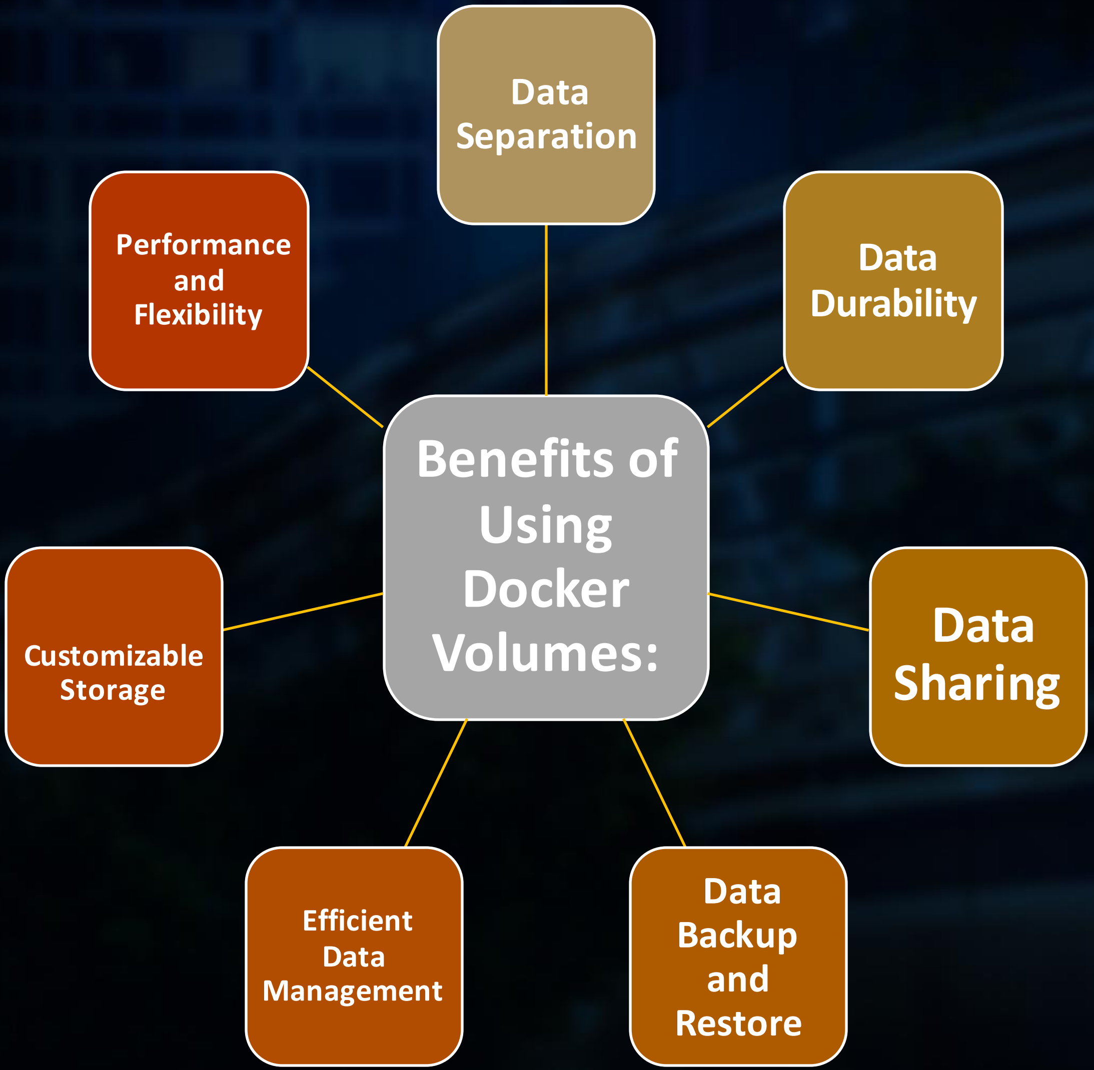
Working with Volumes and Bind Mounts

Docker Volumes

Types of Docker Volumes

- Named Volumes
- Anyonymous volumes

Docker Volumes



Bind mounts

Binding Host Directory

```
docker run -v /host/directory:/container/directory -d my-container
```

Data Accessibility

```
# Host side
echo "Host Data" > /host/directory/data.txt

# Container side
cat /container/directory/data.txt
```

No Volume Creation

```
docker run -v /host/data:/container/data -d my-container
```

When are bind mounts preffered?

- Development and Testing
- Local File Access
- Application Code Hot Reloading
- Configuration Overrides
- Sharing Data with Host
- Database Data Persistence
- Legacy Applications

Using tmpfs Mounts in Docker:

In-Memory Storage

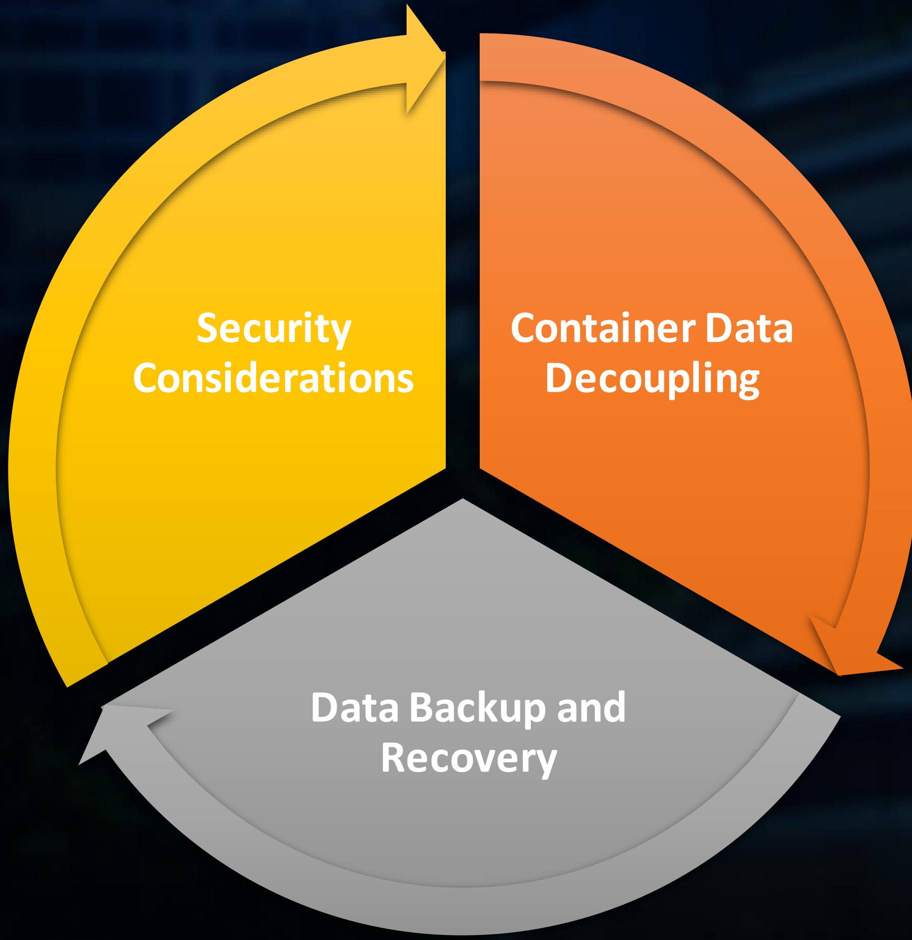
Speed and Efficiency

Limited Capacity

Scenarios Where tmpfs Mounts Are Useful

- Cache Storage
- High-Speed Data Processing
- Temporary Workspaces
- Optimizing I/O
- Performance Benchmarks
- Security and Isolation

Docker Data Management best practices



Docker Data management best practices

Data Sharing Between Containers

Data Migration and Portability

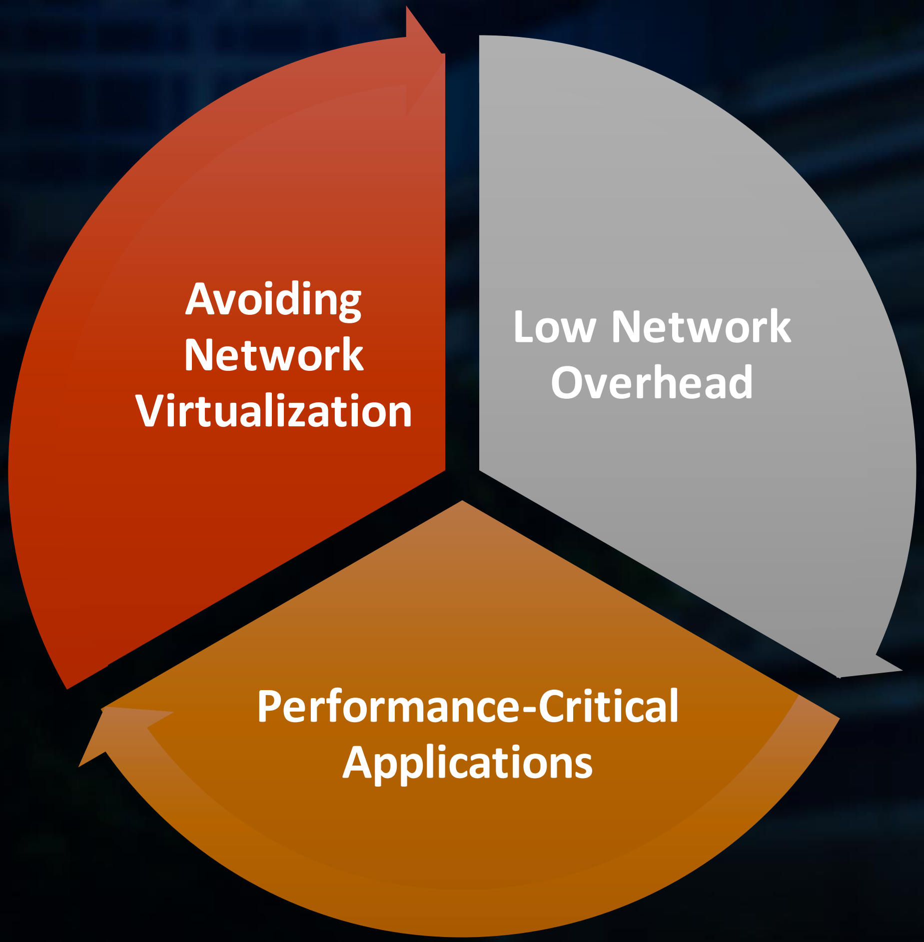
Volume Plugins and Advanced Storage Options

Monitoring and Maintenance

How to create a preloaded database?

- Database Initialization
- docker-entrypoint-initdb.d Directory
- Initialization Scripts
- Connection to Data Persistence
- Benefits of docker-entrypoint-initdb.d

Host Network Use Cases and Scenarios



Benefits and Drawbacks

- **Benefits of Host Networking**
  - Reduced Network Overhead
  - Low Latency
  - Direct Access to Network Hardware
- **Drawbacks of Host Networking**
  - Security and Isolation
  - Port Conflicts
  - Limited Portability

Creating and managing multiple bridge networks in Docker Networking

Understanding Bridge Networks

Docker default bridge network

Custom bridge network

Creating and managingi multiple bridge networks in Docker Networking

Creating Custom Bridge Networks

```
docker network create --driver bridge my_custom_network
```

```
docker network create --driver bridge --subnet 172.18.0.0/16 --gateway 172.18.0.1 my_custom_network
```

```
docker run --name my_container --network my_custom_network -d nginx
```

```
docker network create --driver bridge --label environment=production --label team=backend my_custom_network
```

Inter-Container Communication using Custom Networks

Network Isolation and Security

Microservices communication
- Database access
- Internal APIs

Strategies
- Custom bridge networks
- Use of network policies
- Encrypted communication

Best practices
- Least privilege principle
- Monitor and auditing

Inter-Container Communication using Custom Networks

Connecting containers to custom bridge networks

```
docker run -d --name container1 --network mynetwork myapp
docker run -d --name container2 --network mynetwork myapp
```

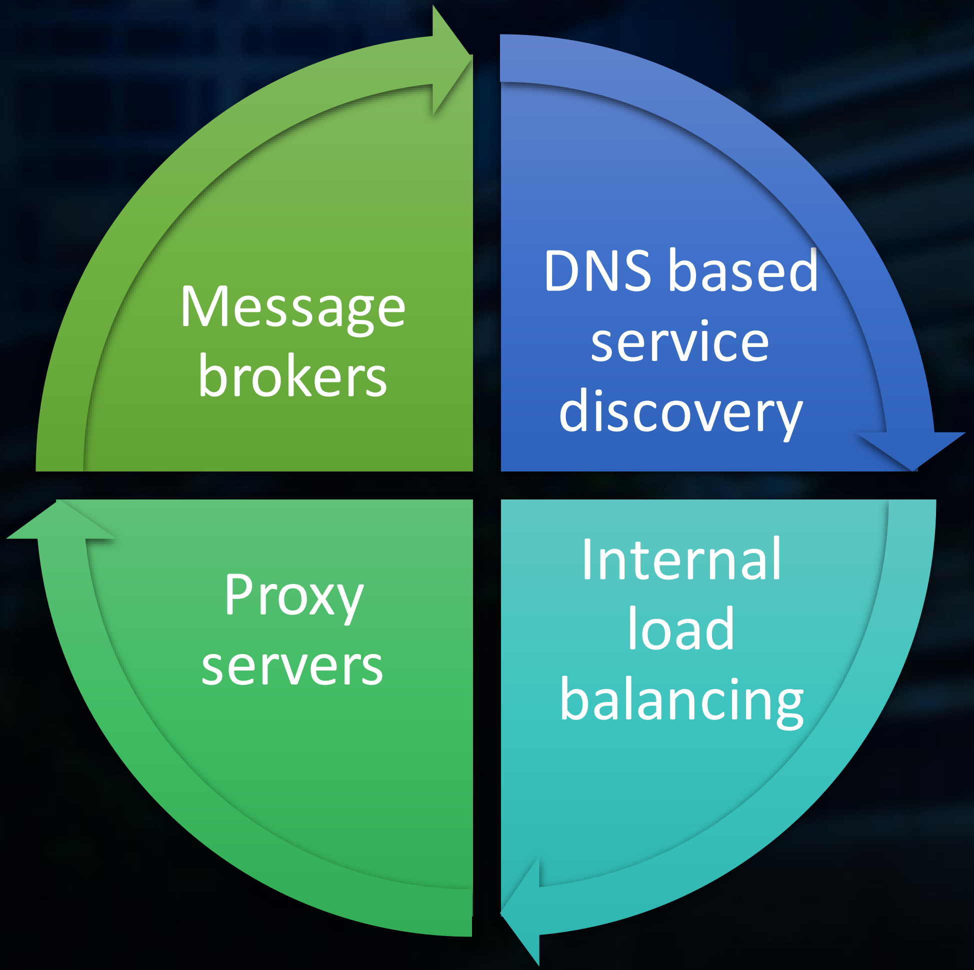
Port mapping and exposing services

```
docker run -d --name webapp --network mynetwork -p 8080:80 mywebapp
```

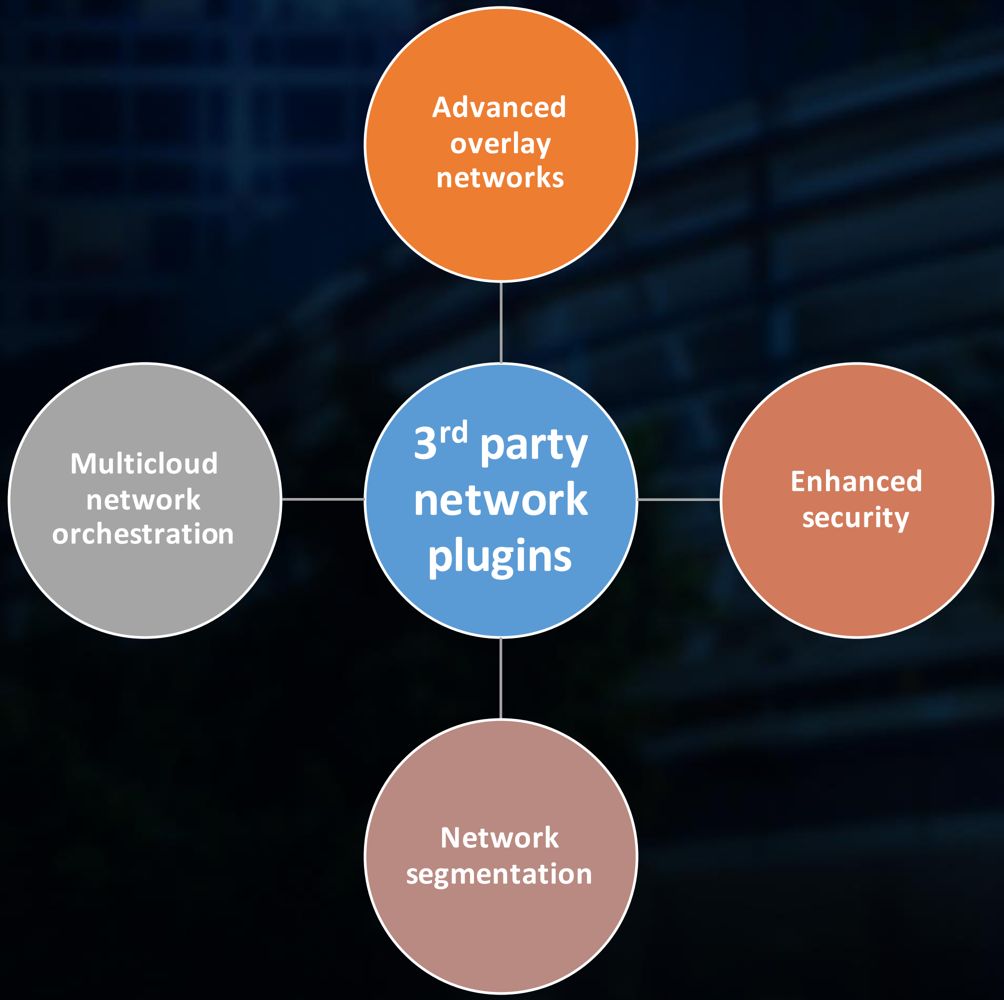
Considerations

- Container naming
- Dynamic IP assignment
- Port collision prevention
- Network scoping

Inter-Container Communication using Custom Networks



Advanced Networking Tools and Plugins in Docker Networking



Advanced Networking Tools and Plugins in Docker Networking

Introduction to Docker Compose

- Key features Declarative definition
  - Orchestration
  - Simplicity
  - Simplifying multi-container application
- Declarative definition
  - Orchestration and automation
  - Rapid stack setup
  - Use of YAML file
- Benefits
  - Dev&Test
  - Local environments
  - Declarative Configuration
  - Collaboration

Introduction to Docker Compose

```yaml
version: '3'  # Docker Compose file version

services:    # List of services in the application
  web:       # Service named 'web'
    image: nginx:latest  # Docker image to use for the 'web' service
    ports:
      - "8080:80"  # Map port 8080 on the host to port 80 on the container

  db:        # Service named 'db'
    image: postgres:latest  # Docker image to use for the 'db' service
    environment:
      POSTGRES_DB: mydatabase  # Environment variable for database name
      POSTGRES_USER: user      # Environment variable for database user
      POSTGRES_PASSWORD: password  # Environment variable for database password

networks:    # Definition of networks for inter-service communication
  mynetwork: {}  # Network named 'mynetwork'

volumes:     # Definition of volumes for data persistence
  myvolume: {}  # Volume named 'myvolume'
```

Introduction to Docker Compose

```
docker-compose up

docker-compose up -d

docker-compose down

docker-compose build

docker-compose up -d --scale service_name=3

docker-compose logs

docker-compose exec service_name command
```

Writing Docker Compose Files for Multi-Container Applications

```yaml
version: '3'  # Docker Compose file version

services:    # List of services in the application
  web:       # Service named 'web'
    image: nginx:latest  # Docker image to use for the 'web' service
    ports:
      - "8080:80"  # Map port 8080 on the host to port 80 on the container

  db:        # Service named 'db'
    image: postgres:latest  # Docker image to use for the 'db' service
    environment:
      POSTGRES_DB: mydatabase  # Environment variable for database name
      POSTGRES_USER: user      # Environment variable for database user
      POSTGRES_PASSWORD: password  # Environment variable for database password

networks:    # Definition of networks for inter-service communication
  mynetwork: {}  # Network named 'mynetwork'

volumes:     # Definition of volumes for data persistence
  myvolume: {}  # Volume named 'myvolume'
```

Writing Docker Compose Files for Multi-Container Applications

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
```

Defining and Configuring Services

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  db:
    image: postgres:latest
    ports:
      - "5432:5432"
```

Service Dependencies

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    scale: 3
```

Scaling Services

Writing Docker Compose Files for Multi-Container Applications

```yaml
version: '3'

networks:
  mynetwork:
    driver: bridge

services:
  web:
    image: nginx:latest
    networks:
      - mynetwork
```

Defining custom networks

```yaml
version: '3'

volumes:
  mydata:

services:
  db:
    image: postgres:latest
    volumes:
      - mydata:/var/lib/postgresql/data
```

Defining volumes

Writing Docker Compose Files for Multi-Container Applications

```
version: '3'

services:
  web:
    image: nginx:latest
    environment:
      - NGINX_PORT=80
```

Setting environment variables

```
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "${NGINX_PORT:-80}:80"
```

Variable substitution

```
version: '3.1'

secrets:
  db_password:
    file: ./secrets/db_password.txt

services:
  db:
    image: postgres:latest
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/db_password
```

Managing secrets

Orchestrating Multi-Container Apps with Compose

```yaml
services:
  web:
    image: nginx:latest
    depends_on:
      - db
```

Service Dependencies

- Depends on
- Healthcheck

```yaml
services:
  web:
    image: nginx:latest
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 1s
      timeout: 3s
      retries: 10
```

Scaling services

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    scale: 3
```

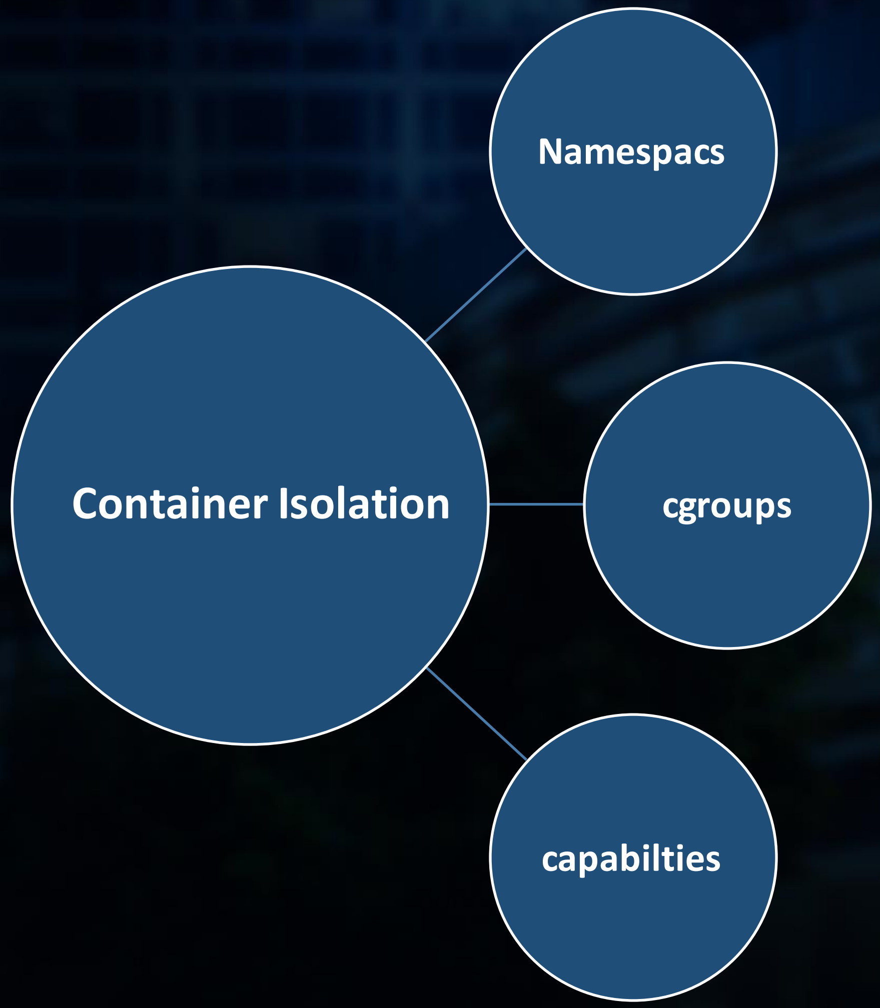
Orchestrating Multi-Container Apps with Compose

- Deployment Startegies
  - Rolling updates
  - Blue-green deployments
  - Zero downtime deployments
- Extending Compose
  - Docker Swarm integration
  - Kubernetes Integration
  - Multi-environment support

Docker Security best practices

1. Understanding container security challenges
2. Running containers with rootless access
3. Securing the container runtime environment

Understanding container security challenges



Understanding container security challenges

Image Security:

Understanding container security challenges

- Security Vulnerabilities:

Understanding container security challenges

- **Resource Management:**

Running containers with rootless access

Root User Risk

- Priviledge escalation
- File-system manipulation
- Resource Exhaustion

Running containers with rootless access

Rootles containers

- non-privileged users
- Reduced attack surface
- Improved isolation

Running containers with rootless access

User Namespaces:

- UID mapping
- Privilege separation
- Rootless containers w/ user namespaces

Securing the container runtime environment

Container runtimes

- Selecting secure runtimes
- Security features

Securing the container runtime environment

Seccomp and AppArmor

- Security profiles
- Configuration and enforcement

Securing the container runtime environment

Least privilege principle

- Limiting capabilities
- Access to Sensitive Host directories

Securing the container runtime environment

Container Networking

- Network Security
- Isolation

Securing the container runtime environment

Runtime security scanning

Runtime scanning tools

Realtime insights

Securing the container runtime environment

Security Auditing and Monitoring

- Continuos monitoring
- Security event tracking

Using minimal base images for enhanced security

Minimal Base Images

Advantages of Minimal Base Images

Using minimal base images for enhanced security

Best Practices for Selecting Minimal Base Images

Creating Minimal Custom Images

Alpine Linux

Official images

Scratch image

Implementing image scanning for vulnerabilities

Image Scanning Defined

Importance of Image Scanning

Image Scanning Tools

Implementing image scanning for vulnerabilities

Continuous Integration/Continuous Deployment (CI/CD) Integration

Actionable Outcomes

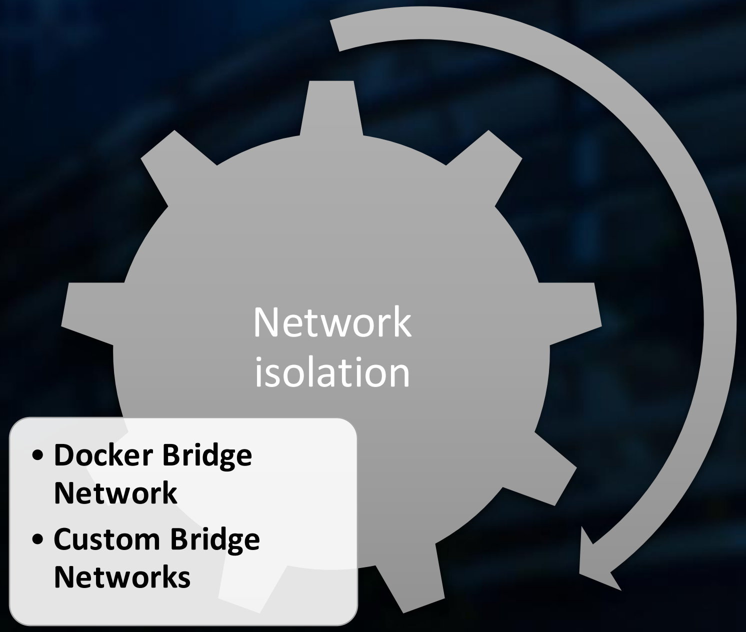
Registries and Repository Scanning

Inter-Container Communication and Secrets

Managing secrets and sensitive data in Docker

- The Importance of Securing Secrets:
- Environment Variables:
- Docker Secret Management:
- Swarm and Kubernetes Secret Management:
- External Secret Management Tools:
- Volume Mounting:
- Access Control and Auditing:
- Rotating Secrets:
- Encrypted Communication:

Secure inter-container communication techniques

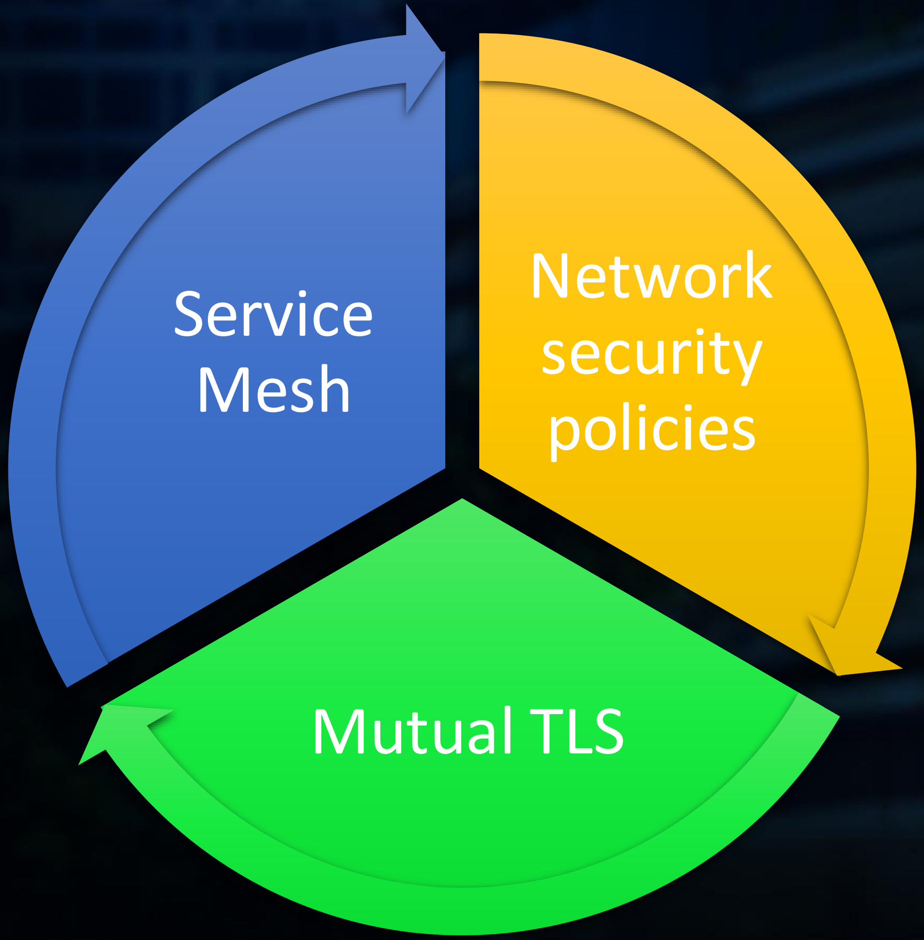


Secure inter-container communication techniques

Use of environment varibles

Container names and DNS resolutions

Secure inter-container communication techniques



Managing secrets and sensitive data in Docker

- Securing secrets
- Envrionment Variables
- Docker Secret Manager

Managing secrets and sensitive data in Docker

Swarm and Kubernetes Secret Management

External Secret Management Tools

Volume Mounting

Managing secrets and sensitive data in Docker

Access Control and Auditing	Rotating Services	Encrypted communication

Overview of Docker security tools

SCOUT

Overview of Docker security tools

sysdig

docker

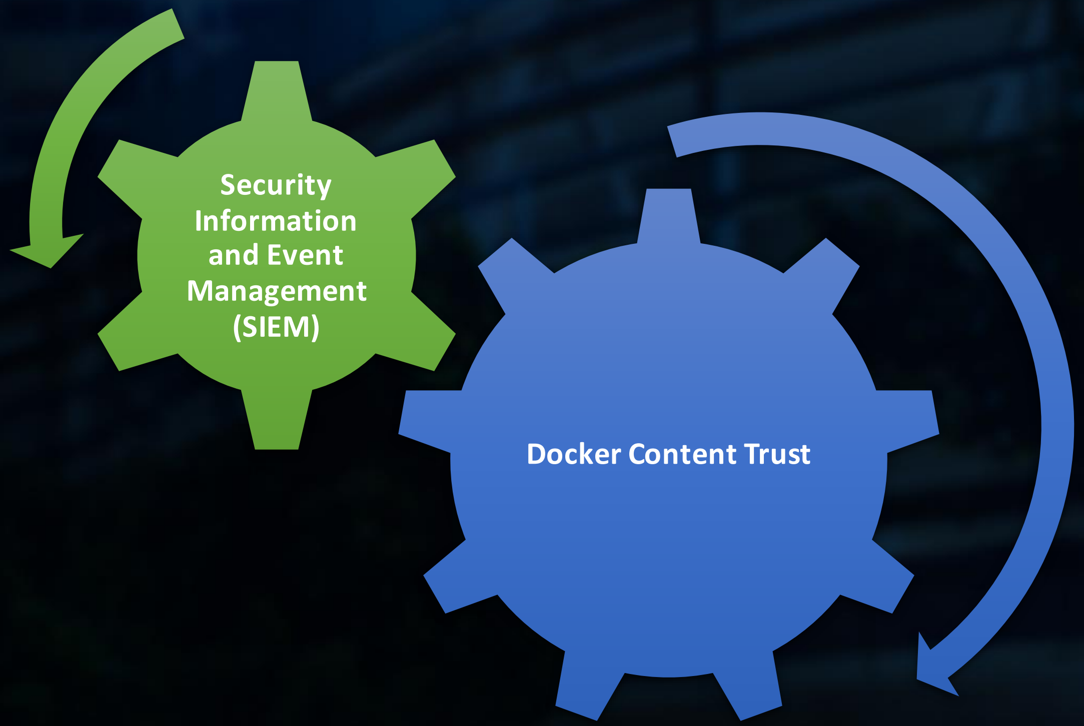
BENCH

Implementing security auditing and monitoring

elasticsearch + logstash + kibana

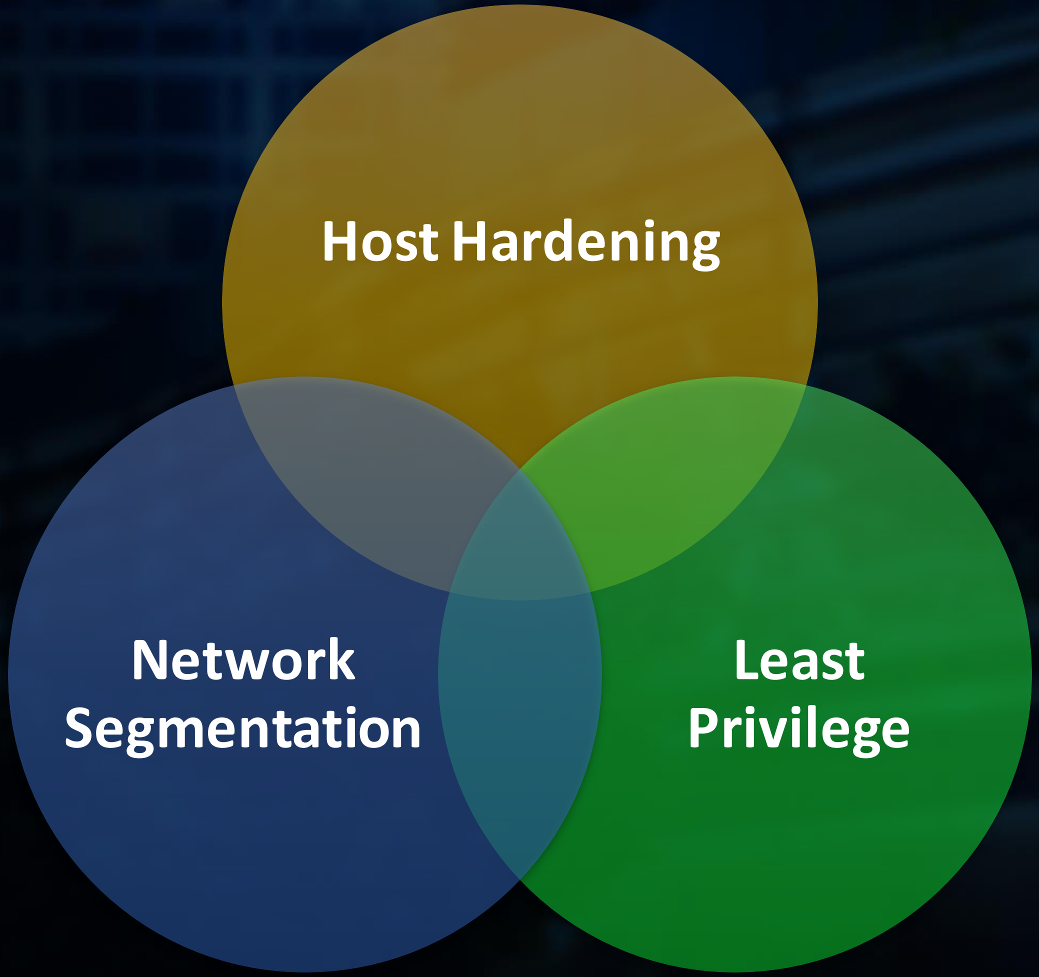
Grafana Prometheus

Implementing security auditing and monitoring

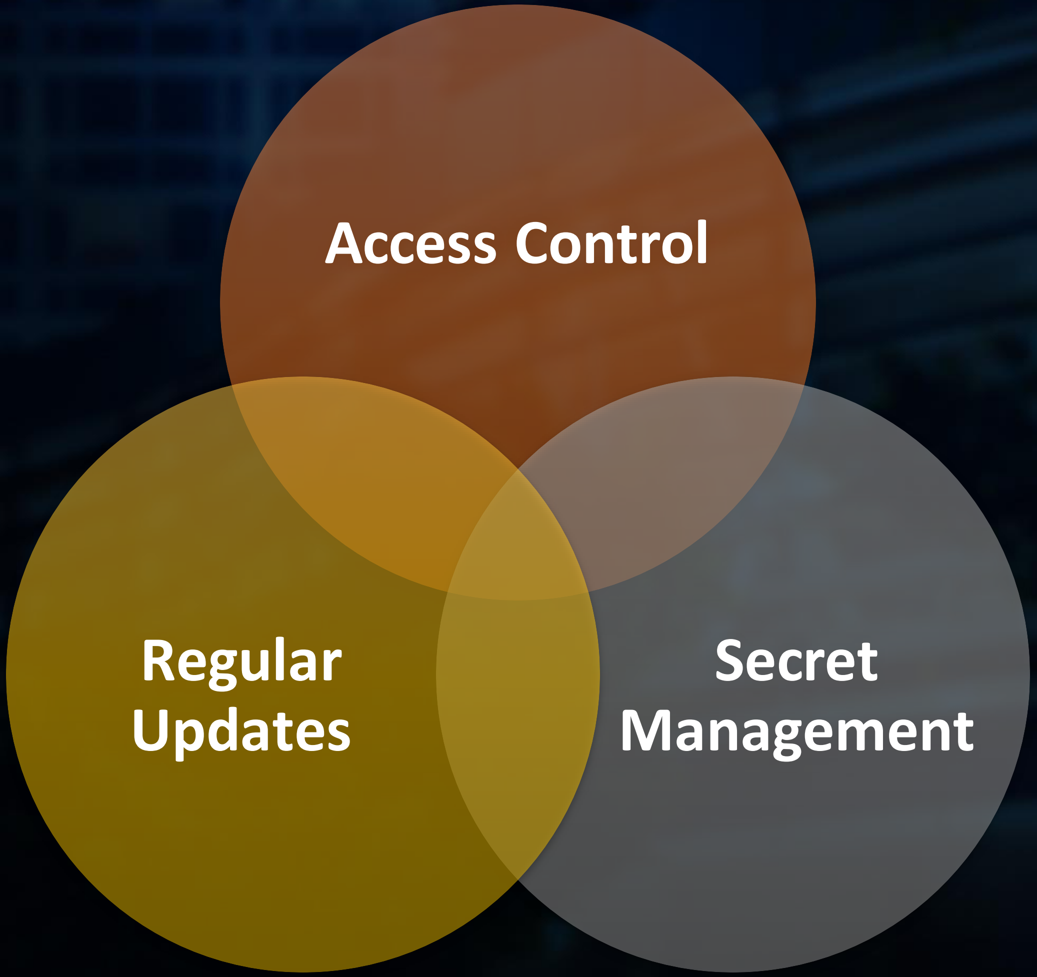


Implementing security auditing and monitoring

Best practices for securing Docker in production



Best practices for securing Docker in production



Best practices for securing Docker in production

Introduction to Docker monitoring

Key Metrics to Monitor

- Resource Usage (CPU and Memory
- Container Health
- Network Activity
- Security Events
- I/O Metrics
- Application-Specific Metrics

Challenges of Container Monitoring

- Ephemeral Nature
- Dynamic Scaling
- Real-Time Visibility
- Container Network Complexity
- Data Collection Overhead

Tools and techniques for monitoring Docker containers

STATS

Tools and techniques for monitoring Docker containers

Grafana

Prometheus

Tools and techniques for monitoring Docker containers

cAdvisor

docker

API

Tools and techniques for monitoring Docker containers

cAdvisor

docker

API

Performance optimization for Dockerized applications

- Understanding Container Performance
  - Resource allocation
  - Resource contention
- Resource Limits and Reservations
  - Resource limits
  - Container Performance Management

Performance optimization for Dockerized applications

Resource Monitoring

Container Orchestration:

Performance optimization for Dockerized applications

Load Balancing

Caching and Acceleration

Performance optimization for Dockerized applications

- Microservices Optimization
- Continuous Performance Testing


Fast Lane