Gmail Vaibhav Kapase <vaibhavkapase132@gmail.com>

🍿 TechOps Examples #18 - 26 Aug 2024

1 message

TechOps Examples <govardhana.mk@techopsexamples.com> Mon, Aug 26, 2024 at 7:15 PM
Reply-To: TechOps Examples <govardhana.mk@techopsexamples.com>
To: "vaibhavkapase132@gmail.com" <vaibhavkapase132@gmail.com>

August 26, 2024 | Read Online

How to Reduce Docker Image Size

In partnership with

oneleet

TechOps Examples

Sign Up | Advertise

Today's edition is brought to you by ONELEET – For companies who care (and whose partners/customers care) about actual real-world security.

Improve your security controls now!

Good day. It's Monday, Aug. 26, and in this issue, we're covering:

- How to Reduce Docker Image Size
- GitHub Enterprise Server vulnerable to critical auth bypass flaw
- Kubernetes 1.31 Brings More Stability to Cloud-Native Deployments
- Docker Best Practices: Choosing Between RUN, CMD, and ENTRYPOINT
- EKS Secret Management - with Golang, AWS ParameterStore and Terraform
- A curated list for awesome Kubernetes sources

You share. We listen. As always, send us feedback at **hello@techopsexamples.com**

Happy to share we have partnered with **ONELEET**, to bring you this offering.

Want SOC 2 compliance without the Security Theater?

Question 🤔 does your SOC 2 program feel like Security Theater? Just checking pointless boxes, not actually building security?

In an industry filled with security theater vendors, Oneleet is the only security-first compliance platform that provides an “all in one” solution for SOC 2.

We’ll build you a real-world Security Program, perform the Penetration Test, integrate with a 3rd Party Auditor, and provide the Compliance Software … all within one platform.

Schedule a demo to get a quote! 🛡️

Use Case

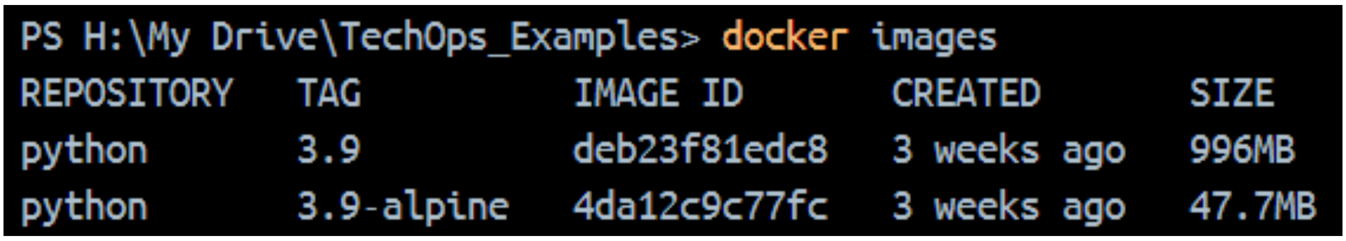
How to Reduce Docker Image Size

Minimizing Docker image sizes accelerates container deployment, and for large-scale operations, this can lead to substantial savings in storage space.

1. Use Official Minimal Base Images:

When building Docker images, always start with an official base image. Instead of using a full-sized OS image, opt for lightweight versions like `python:3.9-slim` or `python:3.9-alpine`. These minimal images contain **only the essentials**, significantly reducing the image size.

Taking an example for a Python image, here are the image sizes for `python:3.9` vs `python:3.9-alpine`:



```
PS H:\My Drive\TechOps_Examples> docker images
REPOSITORY   TAG          IMAGE ID       CREATED        SIZE
python       3.9          deb23f81edc8   3 weeks ago    996MB
python       3.9-alpine   4da12c9c77fc   3 weeks ago    47.7MB
```

Python 3.9-alpine is a whomping 95.2% smaller than Python 3.9.

2. Minimize Layers:

Every command in your Dockerfile (like `RUN`, `COPY`, etc.) generates a separate layer in the final image. Grouping similar commands together into one step decreases the total number of layers, leading to a smaller overall image size.

Instead of doing this:

```
RUN apk update
RUN apk add --no-cache git
RUN rm -rf /var/cache/apk/*
```

Do this:

```
RUN apk update && apk add --no-cache git && rm -rf /var/cache/apk/*
```

3. Use .dockerignore File:

When creating Docker images, Docker transfers all the files from your project directory into the image by default. To avoid including unneeded files, use a `.dockerignore` file to exclude them.

Sample `.dockerignore`

```
__pycache__
*.pyc
*.pyo
*.pyd
venv/
```

4. Multi-Stage Builds (Mandatory atleast for me 😀):

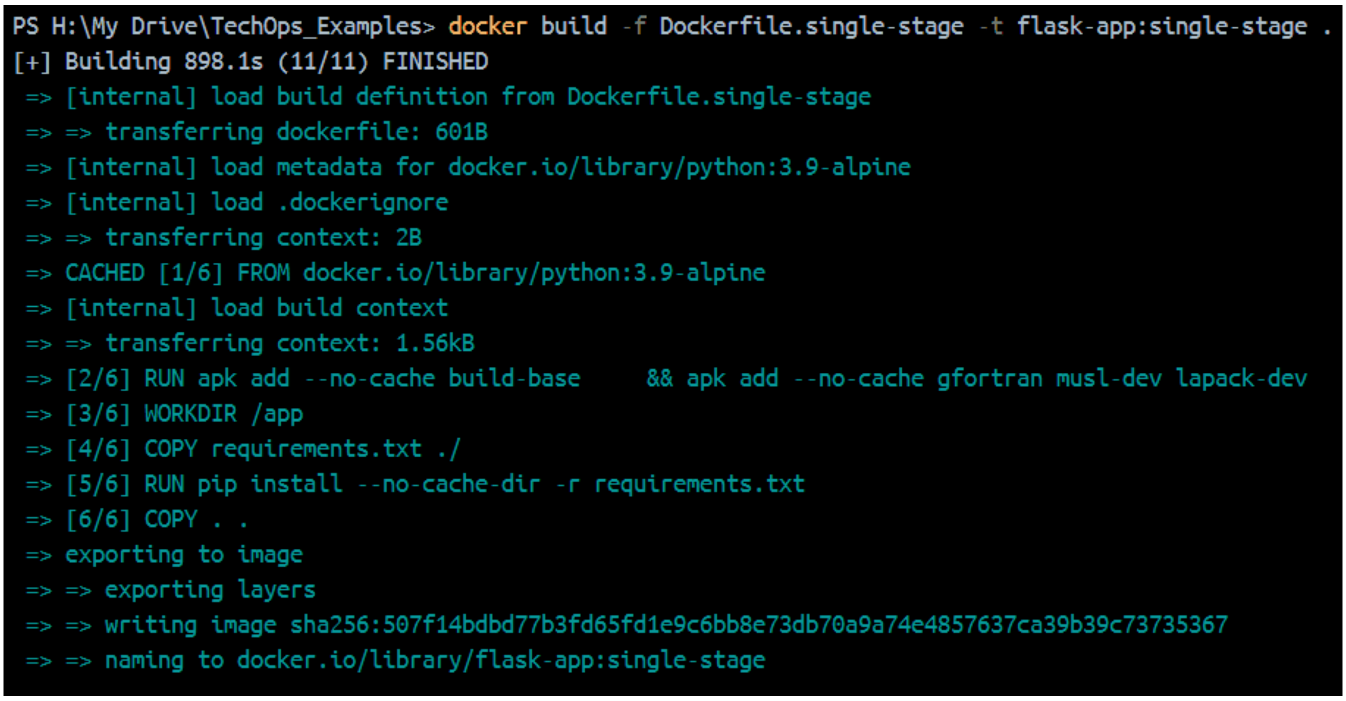
Multi-stage builds enable you to divide the build process from the final runtime environment. This approach is particularly beneficial when your application needs certain tools for compiling that are not necessary in the final image.

Single Stage Vs Multi-Stage Builds Comparison:

Take an example of a Flask app built using the `python:3.9-alpine` image with a **single-stage Dockerfile** like:

```
# Use an official Python runtime as a parent image
FROM python:3.9-alpine

# Install necessary build dependencies
RUN apk add --no-cache build-base \
    && apk add --no-cache gfortran musl-dev lapack-dev
```

```
# Set the working directory
WORKDIR /app

# Copy the requirements file and install dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code to the working directory
COPY . .

# Expose the port the app will run on
EXPOSE 5000

# Run the Flask app
CMD ["python", "app.py"]
```

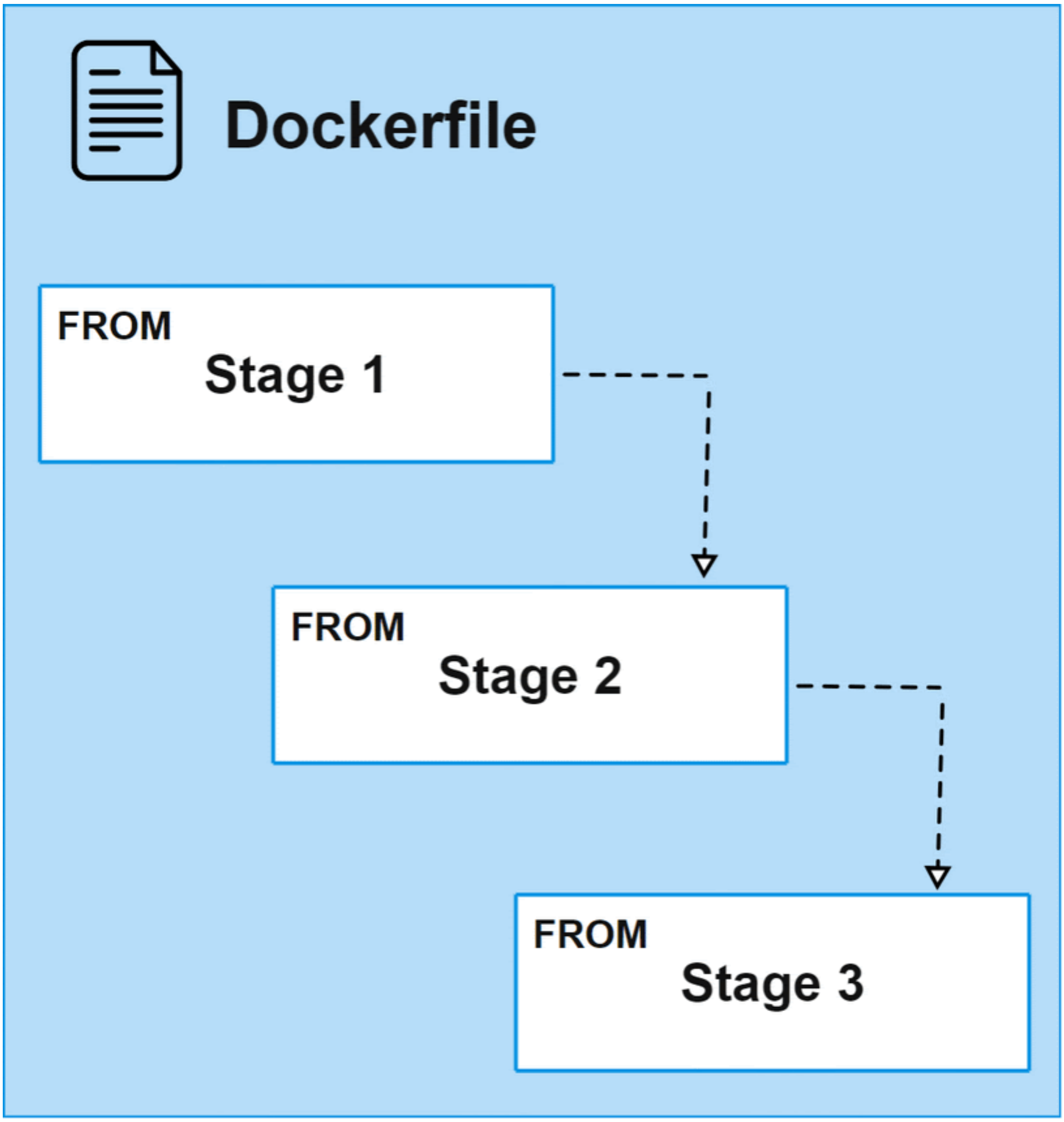


```
PS H:\My Drive\TechOps_Examples> docker build -f Dockerfile.single-stage -t flask-app:single-stage .
[+] Building 898.1s (11/11) FINISHED
 => [internal] load build definition from Dockerfile.single-stage
 => => transferring dockerfile: 601B
 => [internal] load metadata for docker.io/library/python:3.9-alpine
 => [internal] load .dockerignore
 => => transferring context: 2B
 => CACHED [1/6] FROM docker.io/library/python:3.9-alpine
 => [internal] load build context
 => => transferring context: 1.56kB
 => [2/6] RUN apk add --no-cache build-base     && apk add --no-cache gfortran musl-dev lapack-dev
 => [3/6] WORKDIR /app
 => [4/6] COPY requirements.txt ./
 => [5/6] RUN pip install --no-cache-dir -r requirements.txt
 => [6/6] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:507f14bdbd77b3fd65fd1e9c6bb8e73db70a9a74e4857637ca39b39c73735367
 => => naming to docker.io/library/flask-app:single-stage
```

Docker - Single Stage Build Output

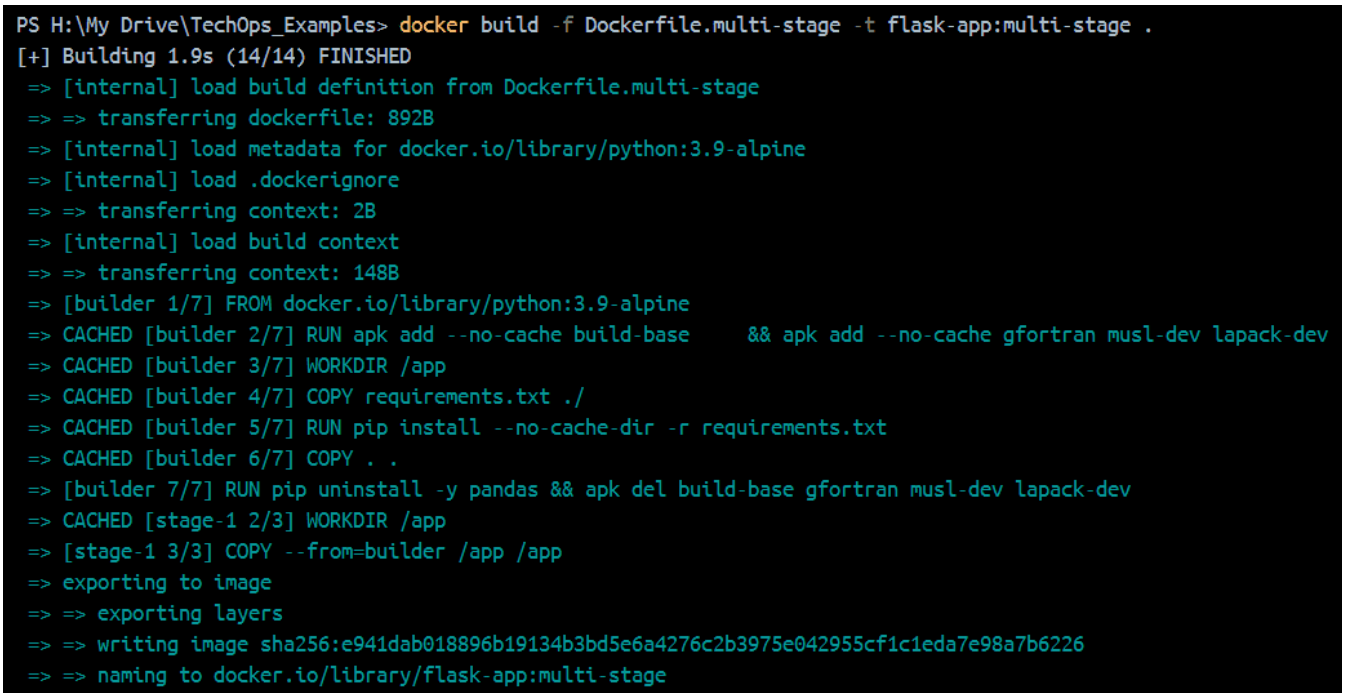
The image built was of size: **588 MB**

Redesigned **Multi Stage Dockerfile** looks like:



```
# Dockerfile.multi-stage

# Stage 1: Build
FROM python:3.9-alpine AS builder

# Install necessary build dependencies
RUN apk add --no-cache build-base \
    && apk add --no-cache gfortran musl-dev lapack-dev

# Set the working directory
```

```
WORKDIR /app

# Copy the requirements file and install dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code to the working directory
COPY . .

# Uninstall unnecessary dependencies
RUN pip uninstall -y pandas && apk del build-base gfortran musl-dev lapack-
dev

# Stage 2: Production
FROM python:3.9-alpine

# Set the working directory
WORKDIR /app

# Copy only the necessary files from the build stage
COPY --from=builder /app /app

# Expose the port the app will run on
EXPOSE 5000

# Run the Flask app
CMD ["python", "app.py"]
```



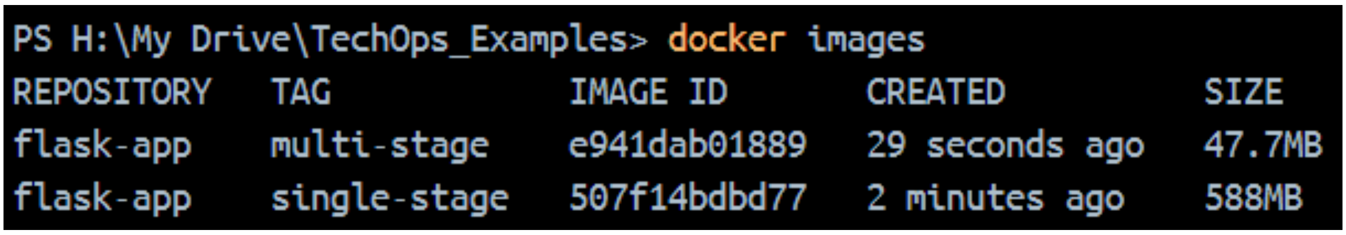
```
PS H:\My Drive\TechOps_Examples> docker build -f Dockerfile.multi-stage -t flask-app:multi-stage .
[+] Building 1.9s (14/14) FINISHED
 => [internal] load build definition from Dockerfile.multi-stage
 => => transferring dockerfile: 892B
 => [internal] load metadata for docker.io/library/python:3.9-alpine
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build context
 => => transferring context: 148B
 => [builder 1/7] FROM docker.io/library/python:3.9-alpine
 => CACHED [builder 2/7] RUN apk add --no-cache build-base     && apk add --no-cache gfortran musl-dev lapack-dev
 => CACHED [builder 3/7] WORKDIR /app
 => CACHED [builder 4/7] COPY requirements.txt ./
 => CACHED [builder 5/7] RUN pip install --no-cache-dir -r requirements.txt
 => CACHED [builder 6/7] COPY . .
 => [builder 7/7] RUN pip uninstall -y pandas && apk del build-base gfortran musl-dev lapack-dev
 => CACHED [stage-1 2/3] WORKDIR /app
 => [stage-1 3/3] COPY --from=builder /app /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:e941dab018896b19134b3bd5e6a4276c2b3975e042955cf1c1eda7e98a7b6226
 => => naming to docker.io/library/flask-app:multi-stage
```

Docker - Multi Stage Build Output

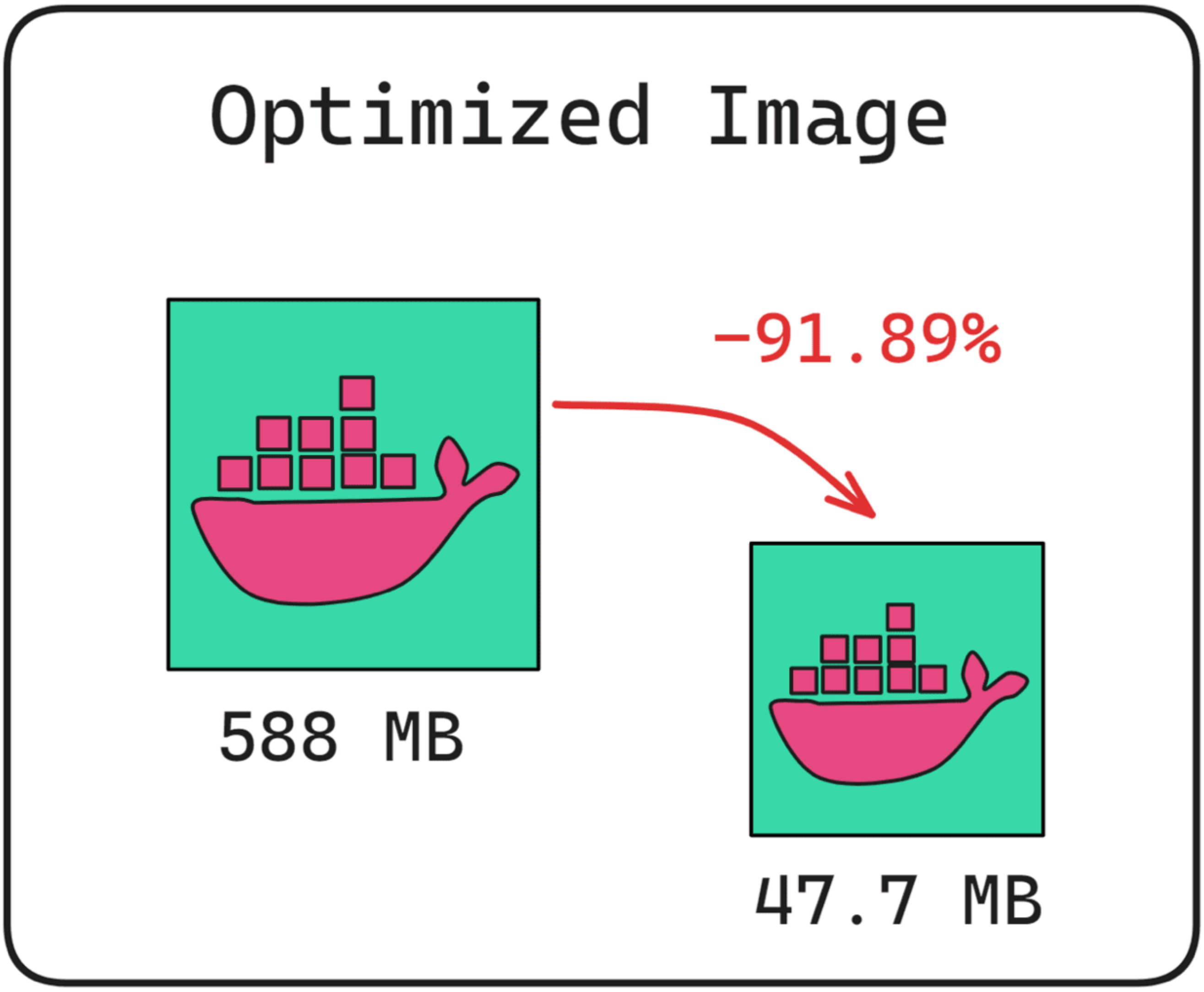
The new image size was: ***Only 47.7 MB***

The application works exactly the same, but it spins up much faster in this version.

This is an illustration of the drastic effect of Multi-Stage Builds.



```
PS H:\My Drive\TechOps_Examples> docker images
REPOSITORY   TAG            IMAGE ID       CREATED          SIZE
flask-app    multi-stage    e941dab01889   29 seconds ago   47.7MB
flask-app    single-stage   507f14bdbd77   2 minutes ago    588MB
```



5. Use Static Binaries and the 'scratch' Base Image:

If your application can be compiled into a static binary, you can use the `scratch` base image, which is essentially an empty image. This leads to extremely small final images.

Example:

```
FROM scratch
COPY myapp /
CMD ["/myapp"]
```

Works well for applications that **don't need operating system-level dependencies**.

Security Considerations

- Use **Trusted and Official** Base Images
- Run Containers as **Non-Root Users**
- Regularly **scan your Docker images** for known vulnerabilities
- Limit the network exposure of your container by **restricting the ports and IP addresses**

```
docker run -p 127.0.0.1:8080:8080 myimage
```

- Avoid hardcoding sensitive information like API keys or passwords directly into your Dockerfile or environment variables.

Final reminder,

Less the image size = Faster deployments + Quicker scaling + Lean infrastructure

p.s. if you think someone else you know may like this newsletter, share with them to ***join here***

Tool Of The Day

Intel Owl

IntelOwl - Get threat intelligence data about a malware, an IP address or a domain from multiple sources at