

# Docker anti-patterns

Make IT

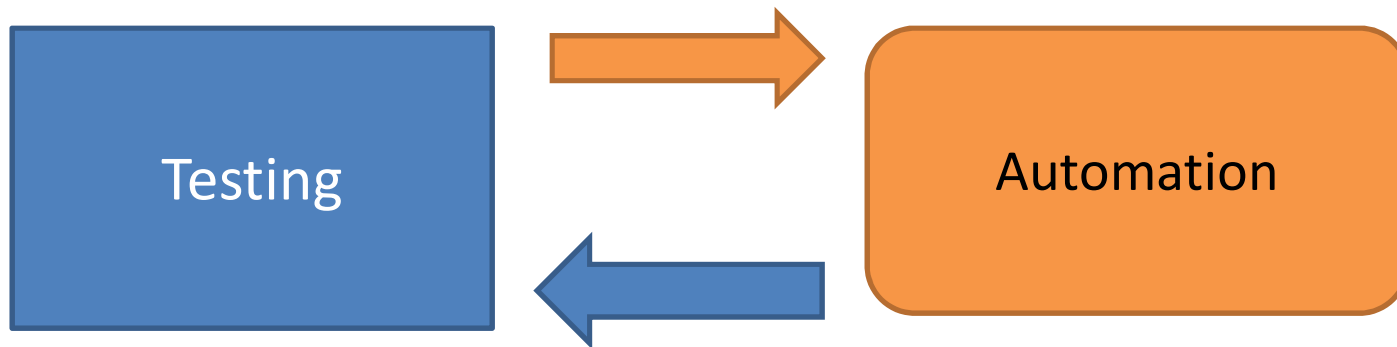
October 2019

Kostis Kapelonis

# Antipattern – common mistake



# Things I love



# Current Work



Docker based CI/CD  
solution for Helm/  
Kubernetes  
deployments

# Current Work

**codefresh**

Pipeline Name  
Release a new update to prod. Must be apdafadsf asdsd...

Documentation Support [TRIGGER PIPELINE](#)

**COMPLETED** STEPS 12 [VIEW YAML](#) START TIME 1/8/2018 22:22 DURATION 10m TRIGGER COMMIT on Idan's Gitlab - codefresh-io/sf-secrets by Idan Arbel [DOWNLOAD LOG](#)

**Initialization** 2.43s

**BUILD →** **BUILD →** **UNIT →**

**PHASE** **DEPENDENCY**

**INITIALIZATION**

- ✓ GIT CLONE Clonning main repository 2.43s
- ✓ GIT CLONE Clonning main repository 2.43s
- ✓ `</>` GIT CLONE Clonning main repository 2.43s
- GIT CLONE Clonning main repository 2.43s

**BUILD**

- ✓ GIT CLONE Clonning main repository 2.43s
- Restart from Step GIT CLONE Clonning main repository 2.43s
- `</>` GIT CLONE Clonning main repository 2.43s
- GIT CLONE Clonning main repository 2.43s

**UNIT**

- GIT CLONE Clonning main repository
- GIT CLONE Clonning main repository





Select pipeline step to view details



# Current Work

HELM Releases Help ADD REPOSITORY

prod@GoogleCloud a few seconds ago

 codefresh	<b>demochat-helm-value-ref</b> Install complete	CLUSTER cluster-1@FirstKubernetes	REVISION 1	MODIFIED 3 months ago	CHART demochat-0.1.0	DEPLOYED
A Helm chart for Kubernetes						
<span>▶ RUN TEST</span> <span>✕ DELETE</span> <span>&lt;/&gt; BADGE</span>						
 Let's Chat	<b>demochat-master</b> Rollback to 8	CLUSTER cluster-1@FirstKubernetes	REVISION 10	MODIFIED 2 months ago	CHART demochat-0.2.0	DEPLOYED
A Helm chart for Kubernetes						
<span>▶ RUN TEST</span> <span>✕ DELETE</span> <span>&lt;/&gt; BADGE</span>						
	<b>wordpress</b> Install complete	CLUSTER cluster-1@FirstKubernetes	REVISION 1	MODIFIED 3 months ago	CHART wordpress-0.7.8	DEPLOYED
Web publishing platform for building blogs and websites.						
<a href="https://github.com/bitnami/bitnami-docker-wordpress">https://github.com/bitnami/bitnami-docker-wordpress</a>						
<span>▶ RUN TEST</span> <span>✕ DELETE</span> <span>&lt;/&gt; BADGE</span>						
 Let's Chat	<b>demochat-prod</b> Upgrade complete	CLUSTER cluster-1@FirstKubernetes	REVISION 9	MODIFIED 2 months ago	CHART demochat-0.2.0	



# Current Work

---



Docker Tutorial | June 20, 2018

Using Docker from Maven and  
Maven from Docker



Kostis Kapelonis



<https://codefresh.io/blog/>

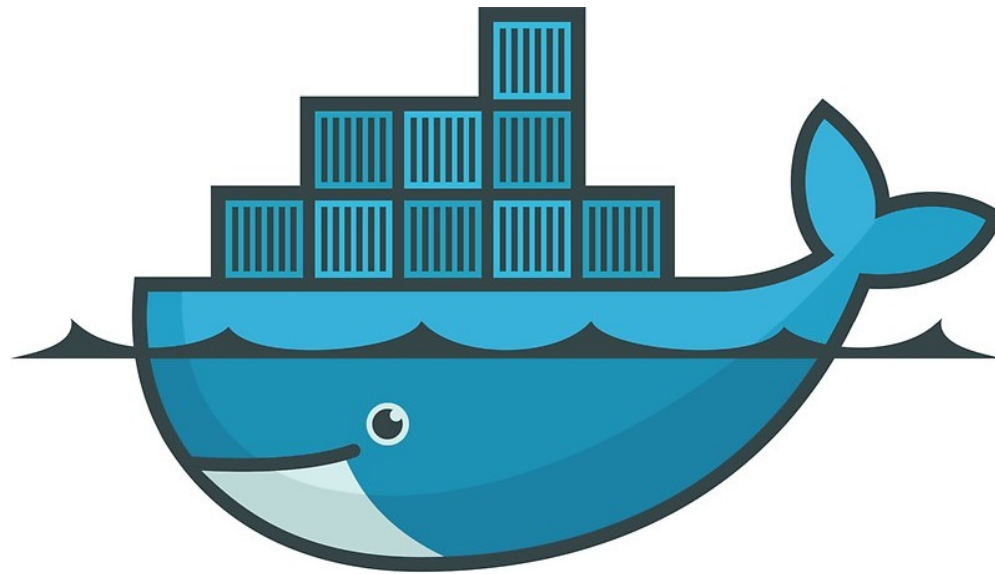
<https://codefresh.io/features/>

# Why

- Container usage is exploding
- Land rush for tools/solutions/companies
- Kubernetes/Nomad/Docker Swarm all use containers
- Containers used for CI/CD
- Reddit /devops and /docker suffer from wrong advice



Docker = Containers



docker

# Docker advice

Too many articles talk about  
Dockerfiles. Too few talk about  
Docker images

# Dockerfile anti-patterns

- [https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
- <https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/>
- Do not use `latest` in your Dockerfile or deployments

# Antipattern 1 – VMs and containers



# Docker advice

Docker images are NOT virtual machines.

# VM questions

## How to handle security updates within Docker containers?

Asked 5 years, 3 months ago   Active 5 months ago   Viewed 31k times



117



When deploying applications onto servers, there is typically a separation between what the application bundles with itself and what it expects from the platform (operating system and installed packages) to provide. One point of this is that the platform can be updated independently of the application. This is useful for example when security updates need to be applied urgently to packages provided by the platform without rebuilding the entire application.



33

Traditionally security updates have been applied simply by executing a package manager command to install updated versions of packages on the operating system (for example "yum update" on RHEL). But with the advent of container technology such as Docker where container images essentially bundle both the application *and* the platform, what is the canonical way of keeping a system with containers up to date? Both the host and containers have their own, independent, sets of packages that need updating and updating on the host will not update any packages inside the containers. With the release of RHEL 7 where Docker containers are especially featured, it would be interesting to hear what Redhat's recommended way to handle security updates of containers is.



# VM questions

## Can I run multiple programs in a Docker container?

Asked 5 years, 11 months ago   Active 2 months ago   Viewed 83k times



131

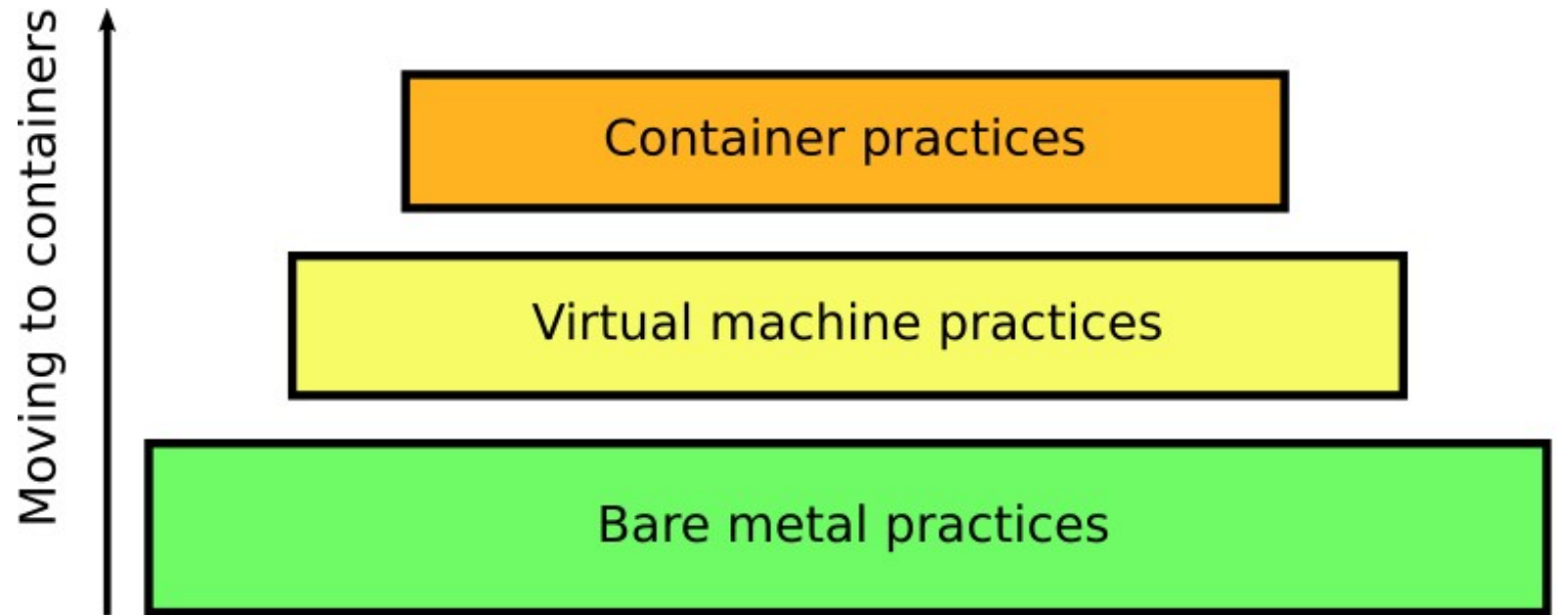


41

I'm trying to wrap my head around Docker from the point of deploying an application which is intended to run on the users on desktop. My application is simply a flask web application and mongo database. Normally I would install both in a VM and, forward a host port to the guest web app. I'd like to give Docker a try but I'm not sure how I'm meant to use more than one program. The documentations says there can only be only ENTRYPOINT so how can I have Mongo and my flask application. Or do they need to be in separate contains, in which case how do they talk to each other and how does this make distributing the app easy?

docker

# Trying to reuse VM knowledge



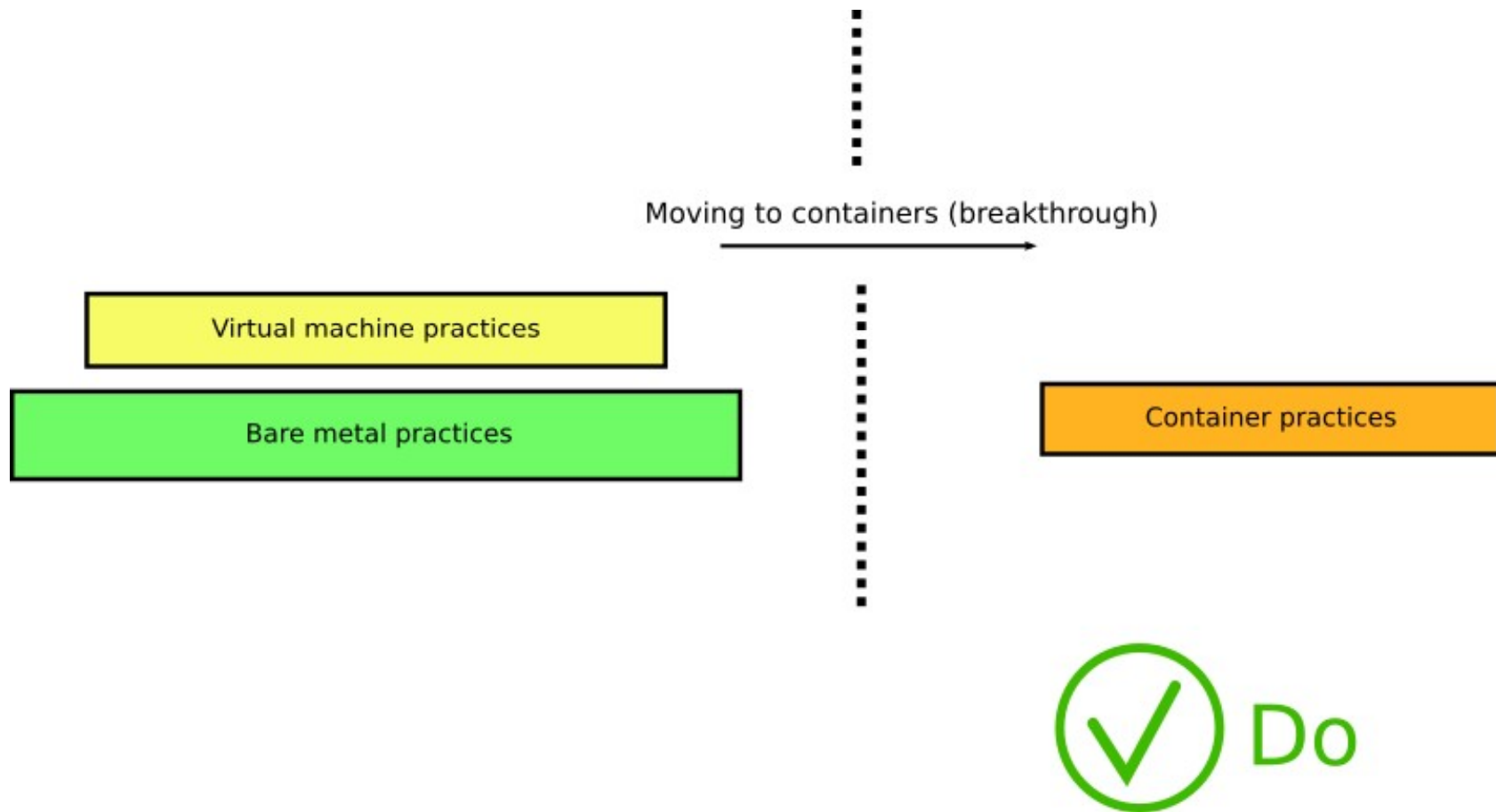
⊗ Don't



# People abuse VM tools

- Configuration management is not the answer
- Ansible/Puppet/Chef/Salt are CM tools
- Most solutions tried to shoe-horn container support
- System administrators were baptized Service Reliability Engineers

# Learn containers from scratch



# Common issues

- Docker tags do not work like Git tags
- Docker has different security model than VMs
- New caching model for layers
- Containers need orchestration
- Auto-scaling, monitoring, logging, errors need a different approach
- Transition from VMs to containers is harder

# Common questions

- How do I update files in a container?
- How do I ssh to a container?
- How do I get logs outside of a container?
- How do I apply security fixes to a container?
- How do I run multiple programs to a container
- How do I backup/restore a container?

# Answer

You don't. Forget your  
VM knowledge

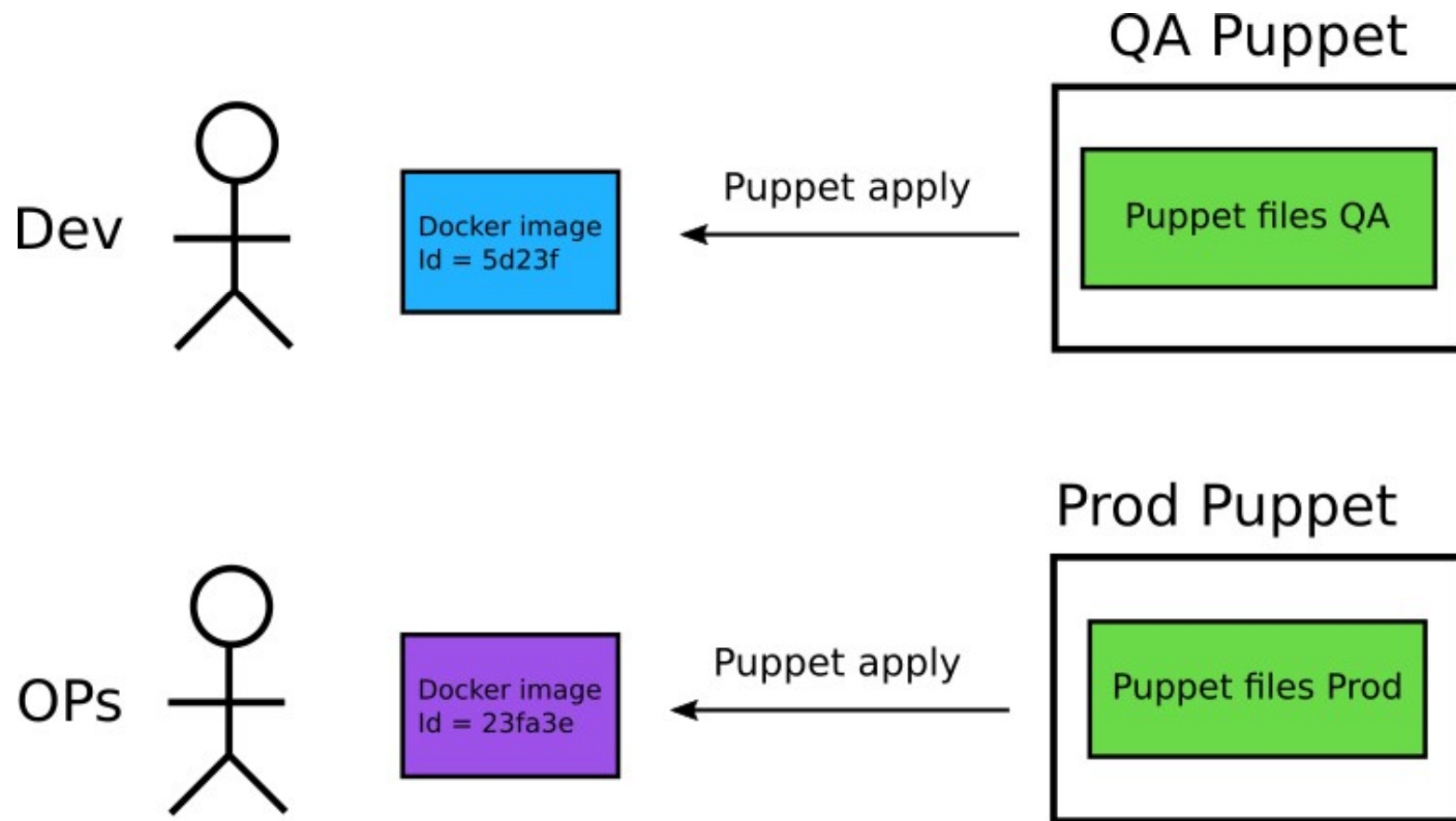
# Antipattern 2 – Opaque Dockerfiles



# What does this dockerfile do?

```
1 FROM alpine:3.4
2
3 RUN apk add --no-cache \
4     ca-certificates \
5     pciutils \
6     ruby \
7     ruby-irb \
8     ruby-rdoc \
9     && \
10    echo http://dl-4.alpinelinux.org/alpine/edge/community/ >> /etc/apk/repositories && \
11    apk add --no-cache shadow && \
12    gem install puppet:"5.5.1" facter:"2.5.1" && \
13    /usr/bin/puppet module install puppetlabs-apk
14
15 # Install Java application
16 RUN /usr/bin/puppet agent --onetime --no-daemonize
17
18 ENTRYPOINT ["java","-jar","/app/spring-boot-application.jar"]
```

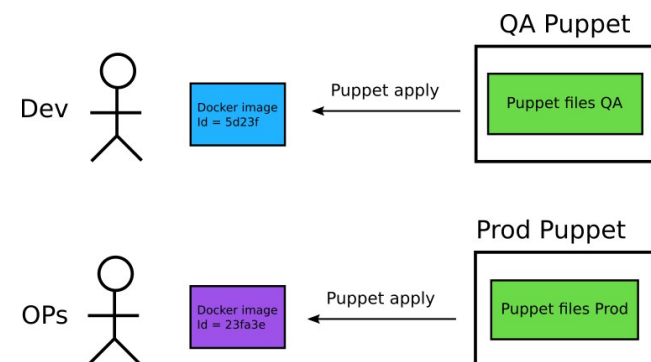
# You need access to Puppet server





# What does this dockerfile do?

- Building the image requires access to Puppet
- Different puppet server -> different image
- Non-repeatable builds
- Docker image requires puppet knowledge
- Application version is undefined
- Team just used the existing VM puppet files (see antipattern 1)



# How to fix this dockerfile

Docker images should be  
transparent and  
repeatable

# What does this dockerfile do?

```
1 FROM openjdk:8-jre-alpine
2
3 ENV MY_APP_VERSION="3.2"
4
5 RUN apk add --no-cache \
6     | | ca-certificates
7
8 WORKDIR /app
9 ADD http://artifactory.mycompany.com/releases/${MY_APP_VERSION}/spring-boot-application.jar .
10
11 ENTRYPOINT ["java", "-jar", "/app/spring-boot-application.jar"]
```

# Transparent Docker images

- Anyone should be able to build the image
- Building multiple times results in the same image
- Application version is visible
- Changing the version of application should be easy
- Other good practices apply (e.g. multi-stage builds)

# Antipattern 3 – Side effects in Docker



# Repeatable Docker images

Building an image should  
have ZERO side effects

# What does this dockerfile do?

```
1 FROM node:9
2 WORKDIR /app
3
4 COPY package.json ./package.json
5 COPY package-lock.json ./package-lock.json
6 RUN npm install
7 COPY . .
8
9 RUN npm test
10
11 ARG npm_token
12
13 RUN echo "//registry.npmjs.org/:_authToken=${npm_token}" > .npmrc
14 RUN npm publish --access public
15
16 EXPOSE 8080
17 CMD [ "npm", "start" ]
```

# This Dockerfile is not idempotent

- Copies source code (OK)
- Downloads Dependencies (OK)
- Runs Unit tests (OK)
- Publishes NPM module (NOT OK)
- Binds to port 8080 (OK)

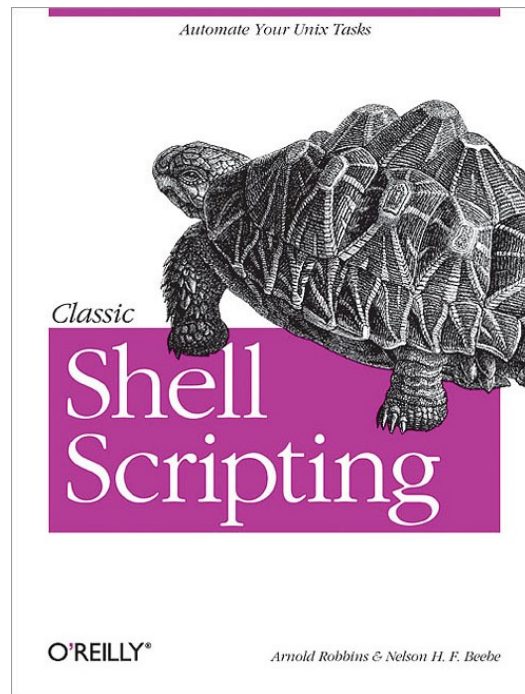




# Repeatable Docker images

Is it safe to build this  
dockerfile twice?

# The golden rule for Dockerfiles



Dockerfiles are NOT  
glorified bash scripts

# Inside a Dockerfile don't:

- Perform Git commits or create Git tags
- Upload files or call POST requests
- Cleanup or tamper with Database Data
- Send notifications to external systems
- Perform destructive actions
- Tamper with shared network files



# Inside a Dockerfile do:

- Clone source code
- Download dependencies
- Compile/package code
- Fetch extra resources (GET calls)
- Process/minify local resources and assets
- Tamper only with local files



# Learn how Docker caching works

Unsafe actions in  
Dockerfiles break the  
caching

# What does this dockerfile do?

```
1 FROM node:10.15-jessie
2
3 RUN apt-get update && apt-get install -y mysql-client && rm -rf /var/lib/apt
4
5 RUN mysql -u root --password="" < test/prepare-db-for-tests.sql
6
7 WORKDIR /app
8
9 COPY package.json ./package.json
10 COPY package-lock.json ./package-lock.json
11 RUN npm install
12 COPY . .
13
14 RUN npm integration-test
15
16 EXPOSE 8080
17 CMD [ "npm", "start" ]
```

# Tampering with the database

1. You run integration tests and they break
2. You fix your source code
3. Docker will NOT rerun the mysql commands
4. Your tests will now use old data



# No side effects in Dockerfiles

- You should be able to build N times
- Building an image should not affect anything external
- Each build must result in the same image
- Remove unsafe actions from Dockerfiles (and place them in CI/CD)
- Only idempotent actions in Dockerfiles



# Antipattern 4 – Types of Images



Don't confuse types of images

Each organization must  
have at least two types  
of Docker images

# Two types of images

- Deployment images
- Sent to production servers
- They contain the application code and NOTHING else
- Tooling images
- Used by developers, ops, CI/CD system
- Have source code, tools, compilers, frameworks
- Cloud tools, linters, testing facilities etc.
- Never deployed anywhere

# Production images: lean and fast



# Production images

- Small (use multi-stage builds)
- Only ship application code in the final state
- Minimal to avoid security issues
- No dev tools, no test tools (git, linters)
- No curl, wget, nc, etc
- Battle hardened, security tested





# CI/CD images: kitchen sink

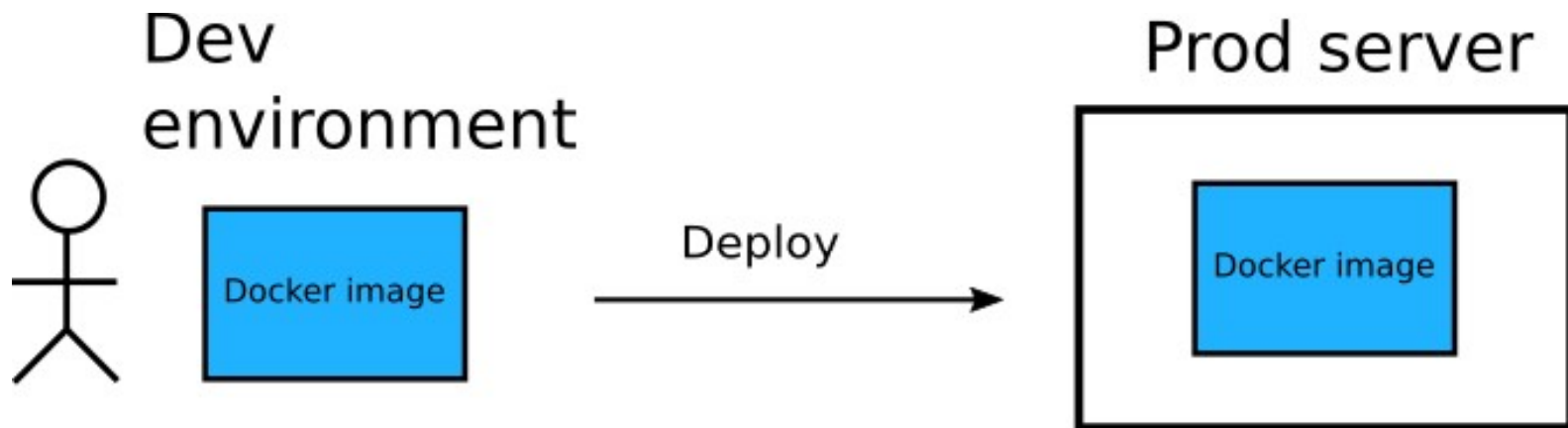


# CI/CD images: kitchen sink

- Used by developers and CI/CD system
- Size of image can be big
- Contain pure source code, dev tools
- Test utilities, cloud utilities included
- Used to setup dev environment
- Have support role, NEVER deployed



# Do not ship dev environments



**REJECTED**



# Multi-stage builds



Why Docker?

Products

Solutions

Customers

Resources

All

Products

Community

Insights

Company



## Multi-Stage Builds

By Sophia Parafina

July 05 2017



Automated builds, docker cloud, Multi-stage Builds

This is part of a series of articles describing how the AtSea Shop application was built using enterprise development with Docker. In the [previous post](#), I introduced the AtSea application and how I developed a REST application with Docker. [Multi-stage builds](#), a Docker feature introduced in Docker 17.06 CE, let you orchestrate a complex build with a single Dockerfile. Before multi-stage build, Docker users would use a script to compile the applications on the host and then use Dockerfiles to build the images. The [AtSea application](#) is the perfect use case for a multi-stage build because:

# Single stage image

```
1 FROM golang:1.7.1
2 COPY src /go/src
3 RUN go build -o bin/sample src/sample/trivial-web-server.go
4 EXPOSE 8080
5 CMD ["/go/bin/sample"]
6 |
```

---



# Single stage image

- This image is 700MB
- It is a full Debian/Ubuntu environment
- Contains Git, Mercurial, SSH client, subversion, curl
- It has a full blown GO dev environment
- This is an attacker's dream
- Very slow to deploy



# Multi-stage builds

```
1 FROM golang:1.7.1 AS build-env
2 COPY src /go/src
3 RUN CGO_ENABLED=0 GOOS=linux go build -o bin/sample src/sample/trivial-web-server.go
4
5 FROM scratch
6 COPY --from=build-env /go/bin/sample /app/sample
7
8 EXPOSE 8080
9 CMD ["/app/sample"]
10
```

---



# Multi-stage builds

- This image is 6 MB
- It contains just the executable
- Minimal attack vector(not even a shell)
- Harder to exploit
- Very fast to deploy



# Check your Docker images



30MB for Go app, 150MB  
for Java, other  
languages in-between

# Check your Docker images



Do not ship GIT, gcc, curl, test frameworks etc. in production



# Antipattern 5 – Different env images

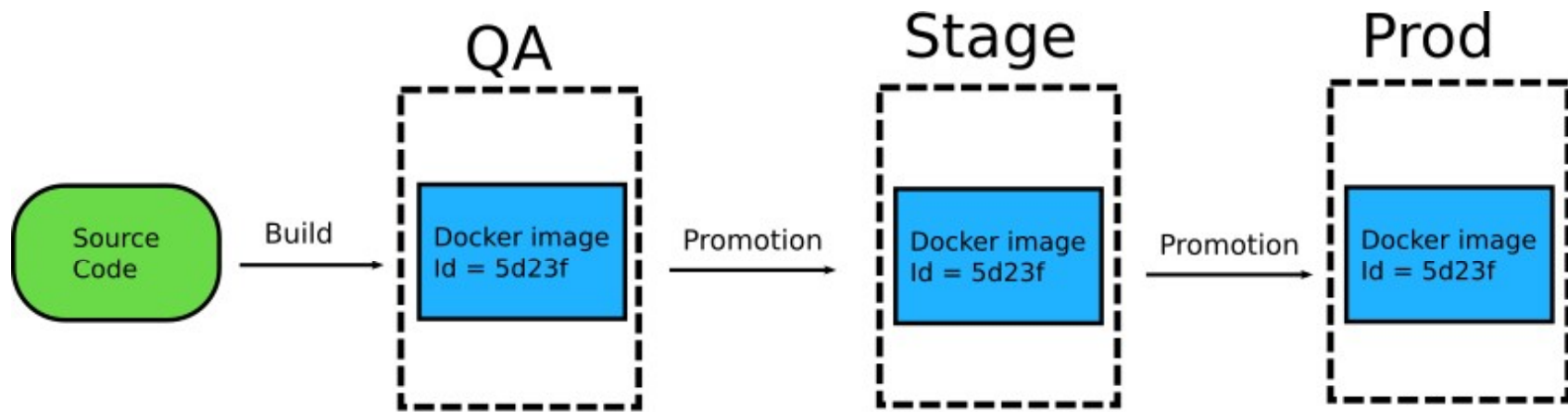




# Why immutable containers

- Containers (unlike VMs) are immutable
- Build once and promote to environments
- You deploy what you tested
- Multiple promotion stages
- Each release is a new build (brand new)
- Need help from CI/CD

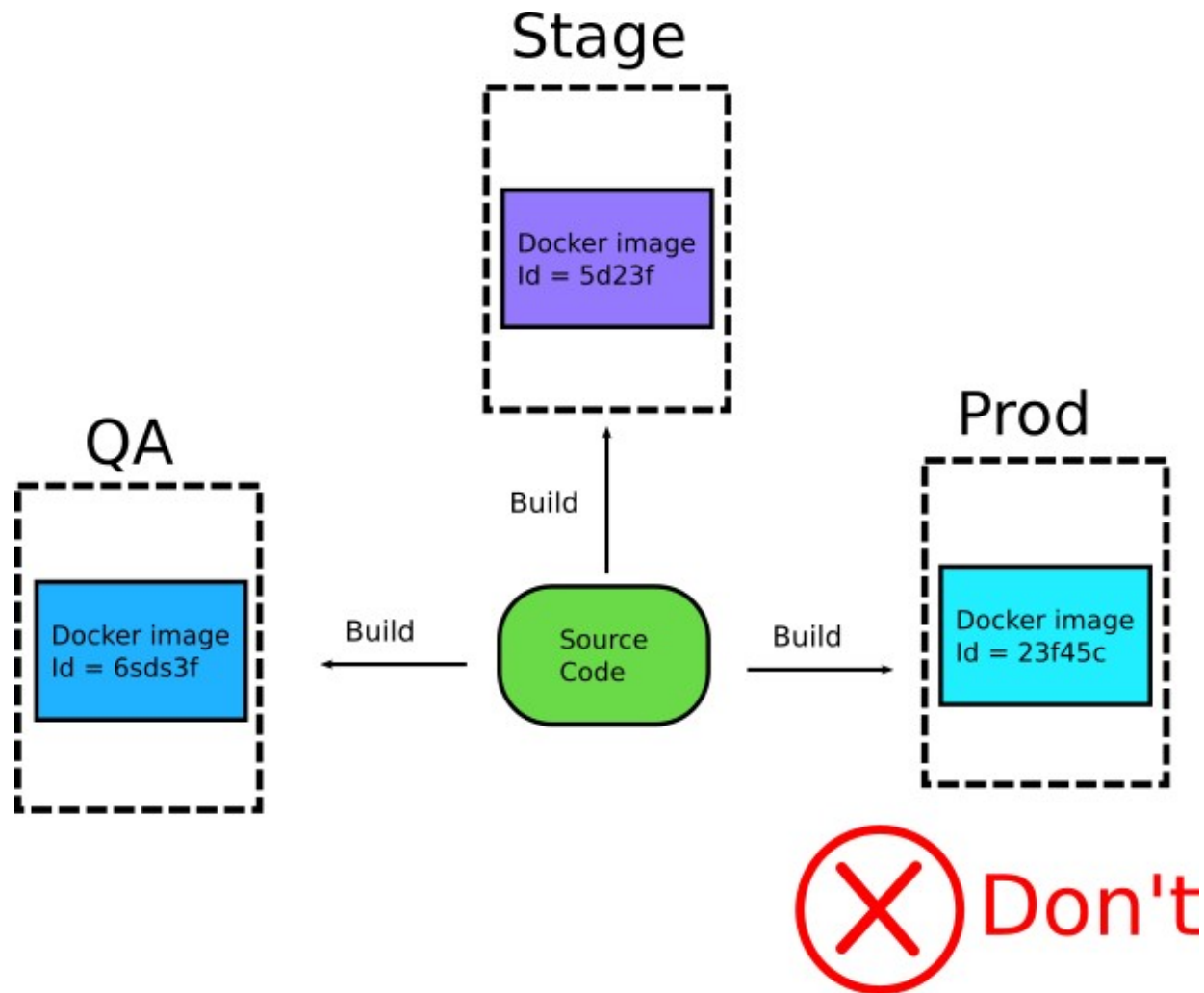
# Promote the same image



# Golden rule for deployments

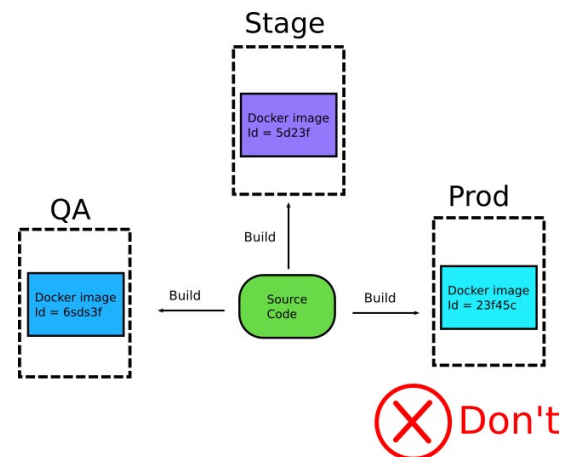
A Docker image is built  
once for ALL  
environments

# Different images per environment



# Different images per environment

- No guarantee that what you tested is the same
- Developers abuse QA image to insert debug tools
- Configuration drift is possible (also a big problem with VMs)



# Fix your CI/CD system

- Do not build multiple times the source code
- Promote image between pipelines
- Promote image between different registries
- All configuration should be external
- The same Docker image ID is used in the whole lifecycle

# Antipattern 6 – Building in production



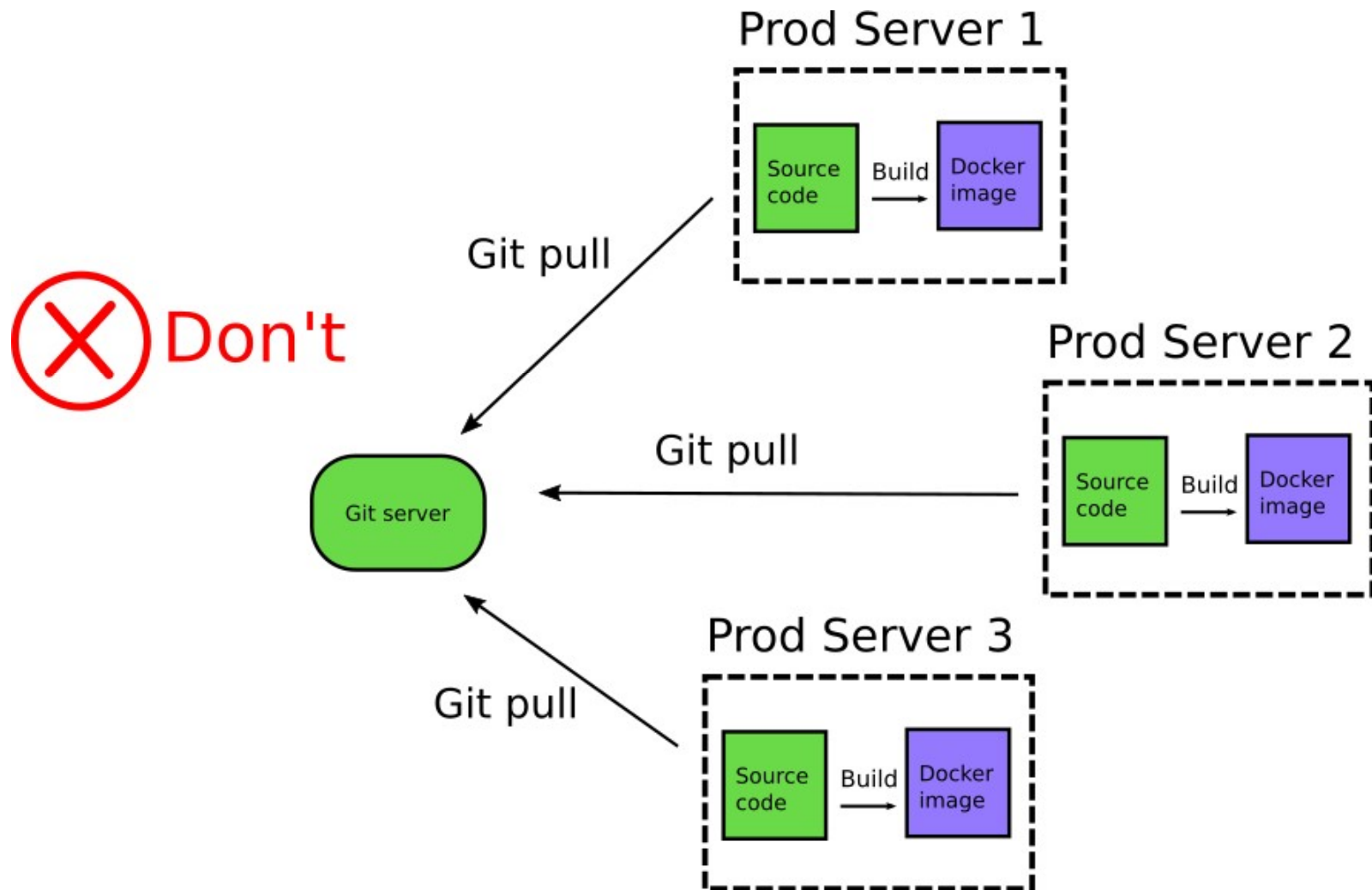
# Very common anti-pattern

- Building Docker images in production servers
- Abuse GIT/Ansible as deployment tool
- Each server is responsible for its own image
- No central image repository
- People are trying to reuse VM knowledge (see anti-pattern 1)



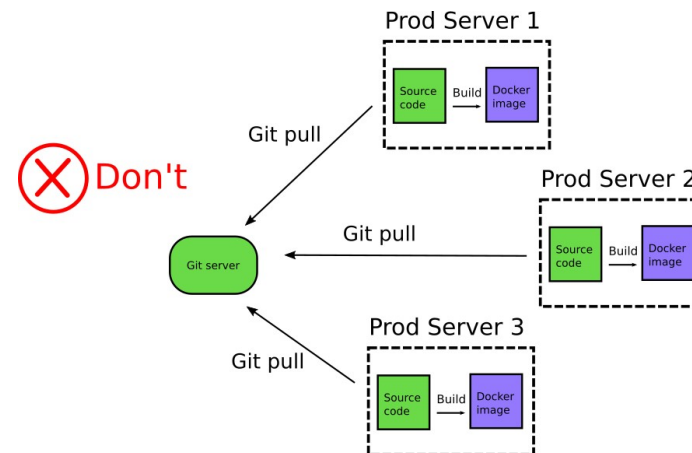


# Building in production servers



# Building in production servers

- Git is not a deployment tool
- Inbound git access has security issues
- This is a VM technique (anti-pattern 1)
- Different image per server (anti-pattern 5)
- You don't know what docker image is deployed

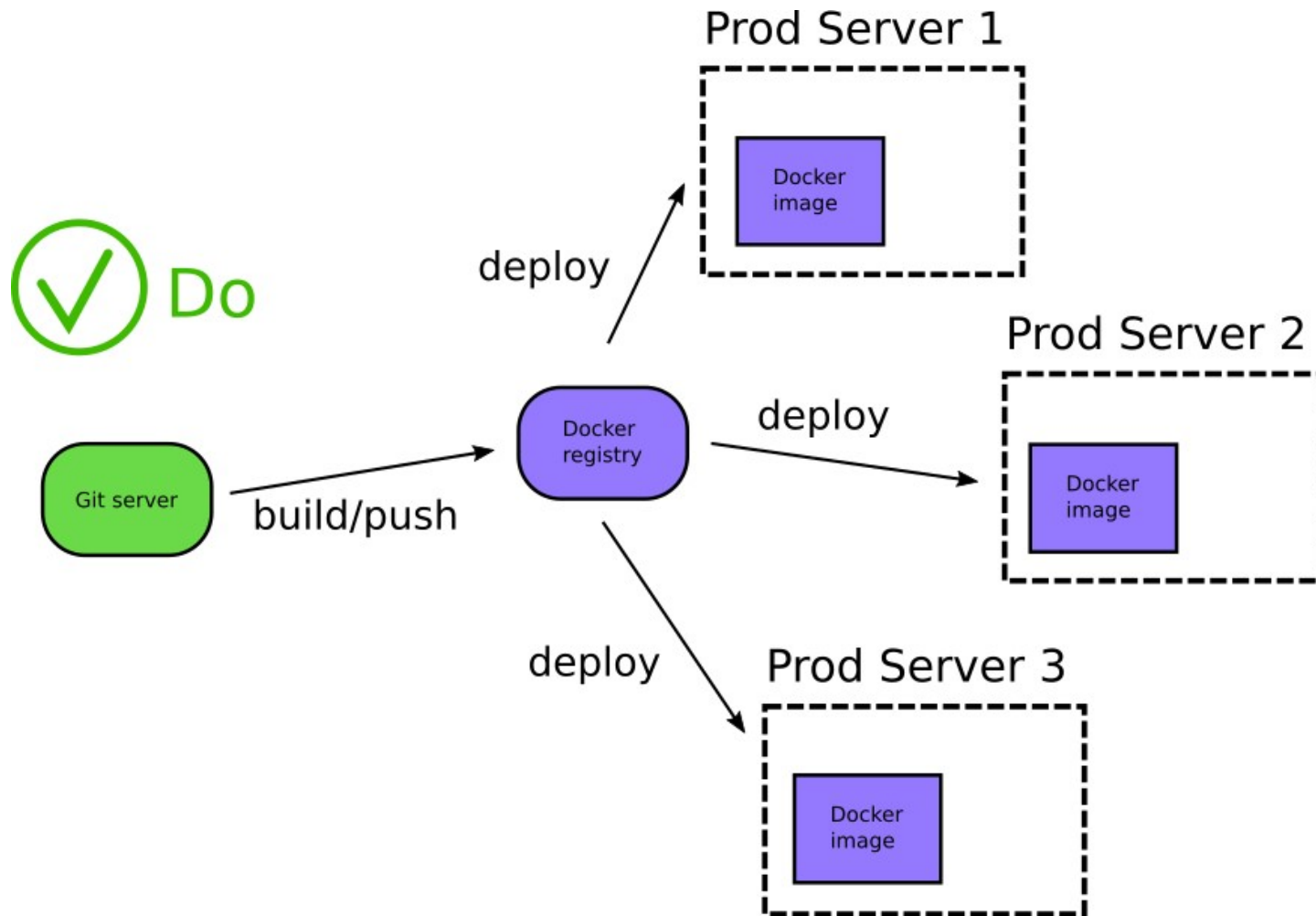


# Central Docker storage



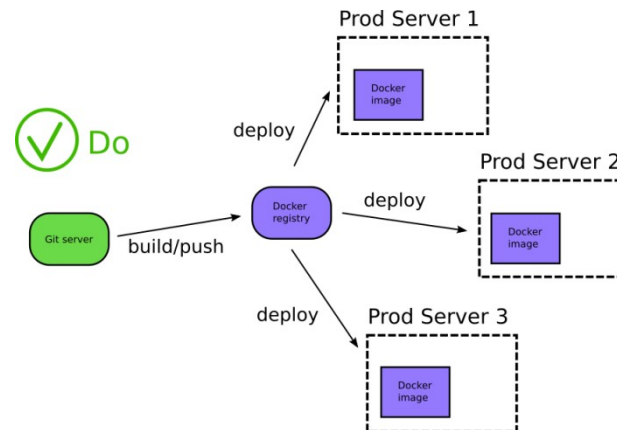
Learn/Use Docker Registries

# Use Docker registries



# Use Docker registries

- Central repository of past releases
- All servers run same image
- Rollback is trivial
- Secure outbound access + DMZ
- Source code never leaves the premises



# Use Multiple Docker registries

1. Dev Docker registry (auto-cleaned)
2. QA Registry
  - Holds Release candidates
  - Promoted from Dev
3. Prod Registry
  - Deployed in production
  - Security scans
  - Audit trail
  - Promoted from QA



# Use Docker registries

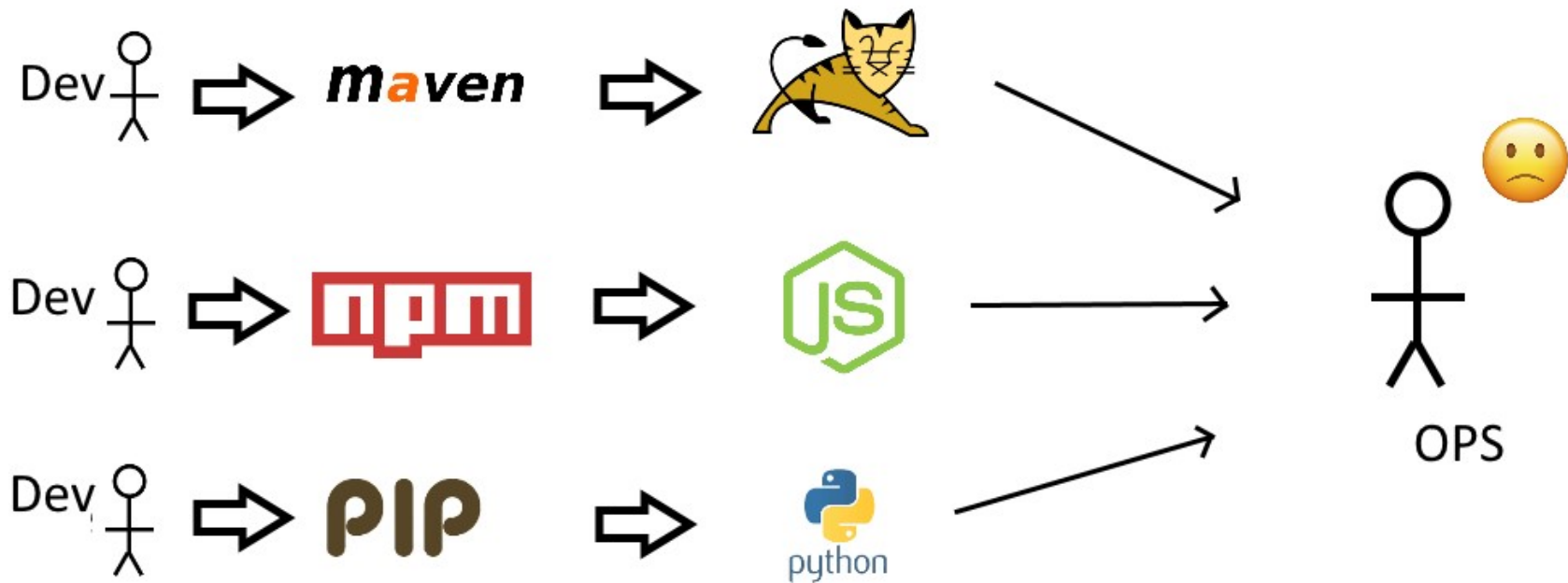
- Many open source solutions
- Many proprietary solutions
- Well defined API
- Looking to the future - OCI
- Docker registries for Helm charts as well

# Antipattern 7 – Using Git hashes

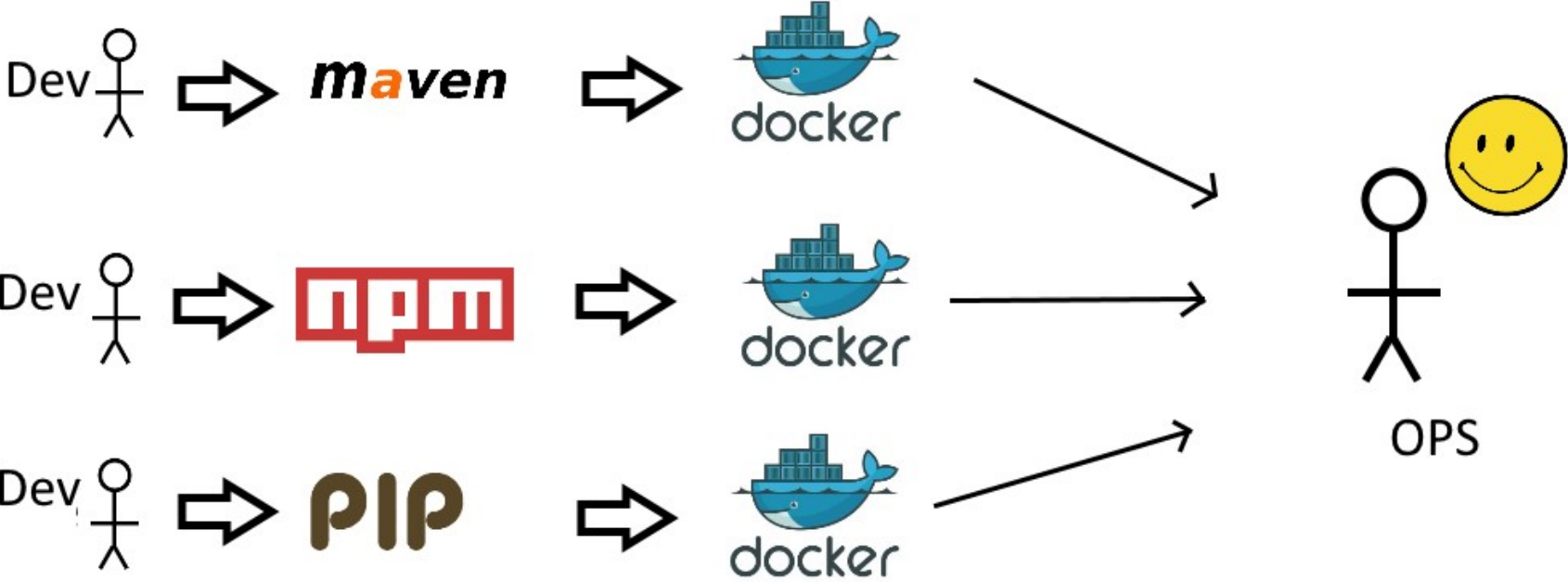




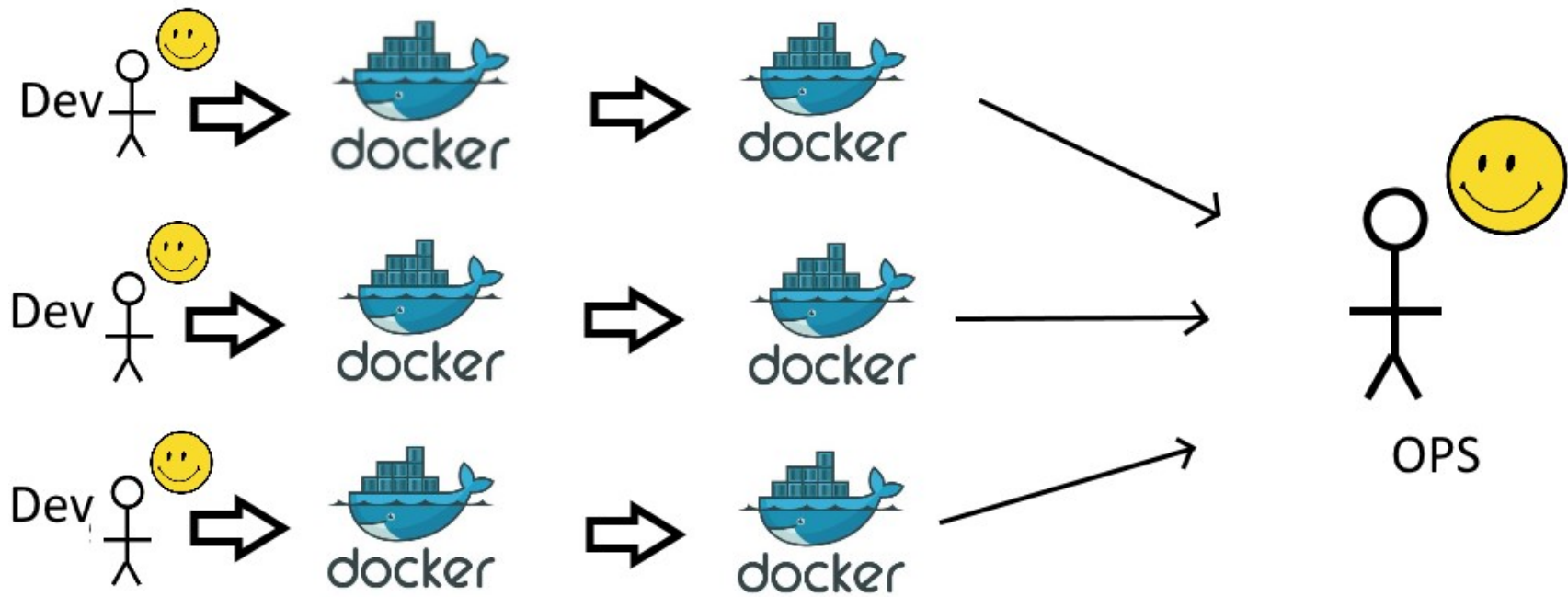
# Before containers



# Containers as deployment artifact



# Containers in CI/CD



# Corollary from anti-patterns 5 - 6

- Build Docker images once
- Promoted between Docker registries

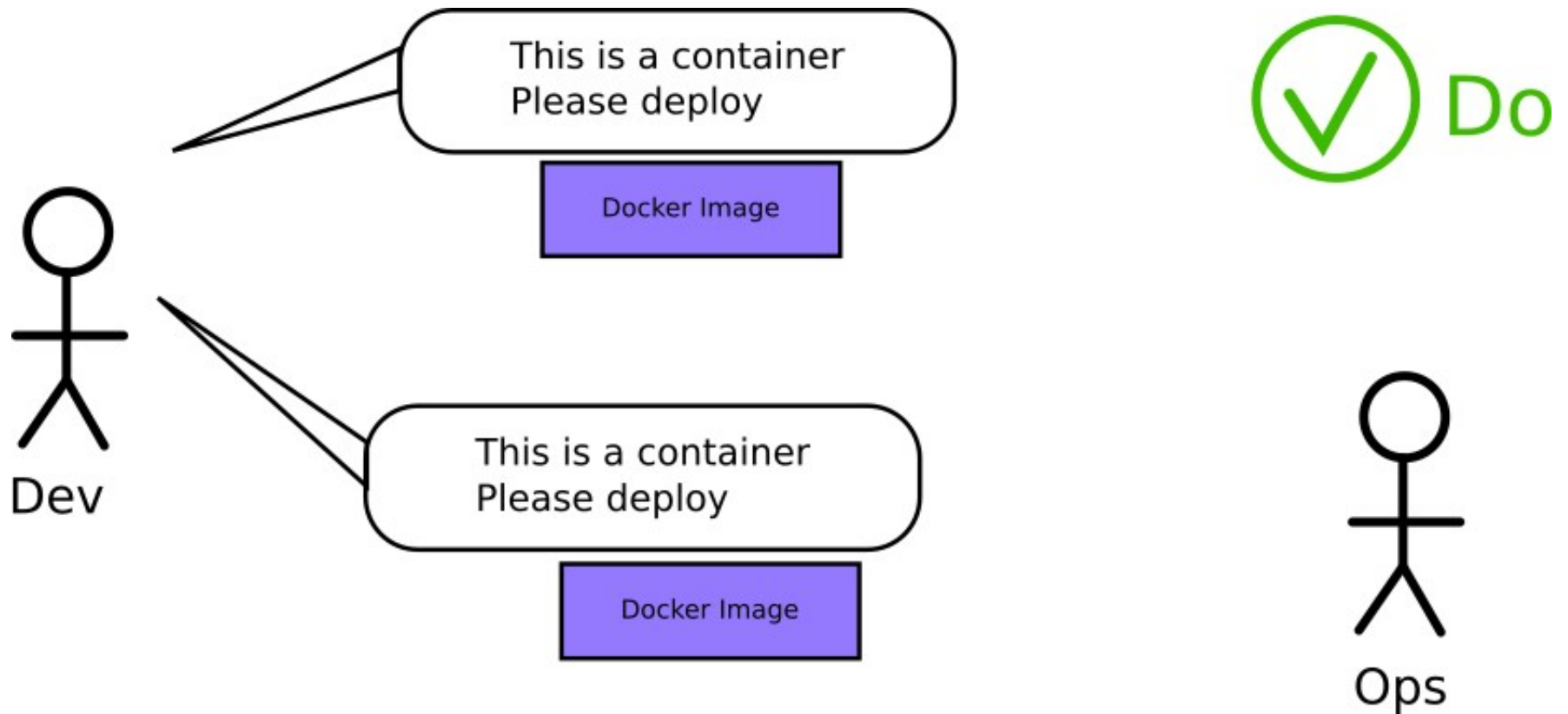
Docker images are the central entity

Talk about Docker tags instead  
of Git hashes

# Git hash as hand-off



# Docker tag as hand-off

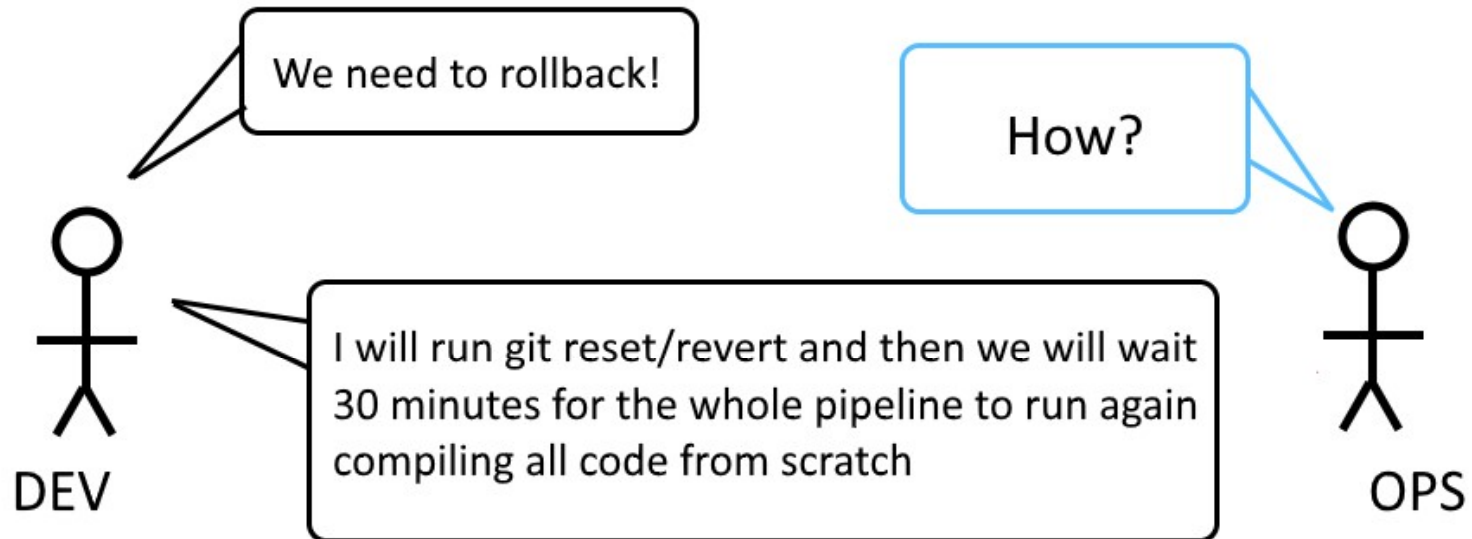


# Don't talk about Git hashes

- Everybody should talk about Docker tags
- Docker registry is the central place to pick releases for QA/Ops
- CI/CD pipelines should work with Docker images
- Build image once and promote image in the software pipeline

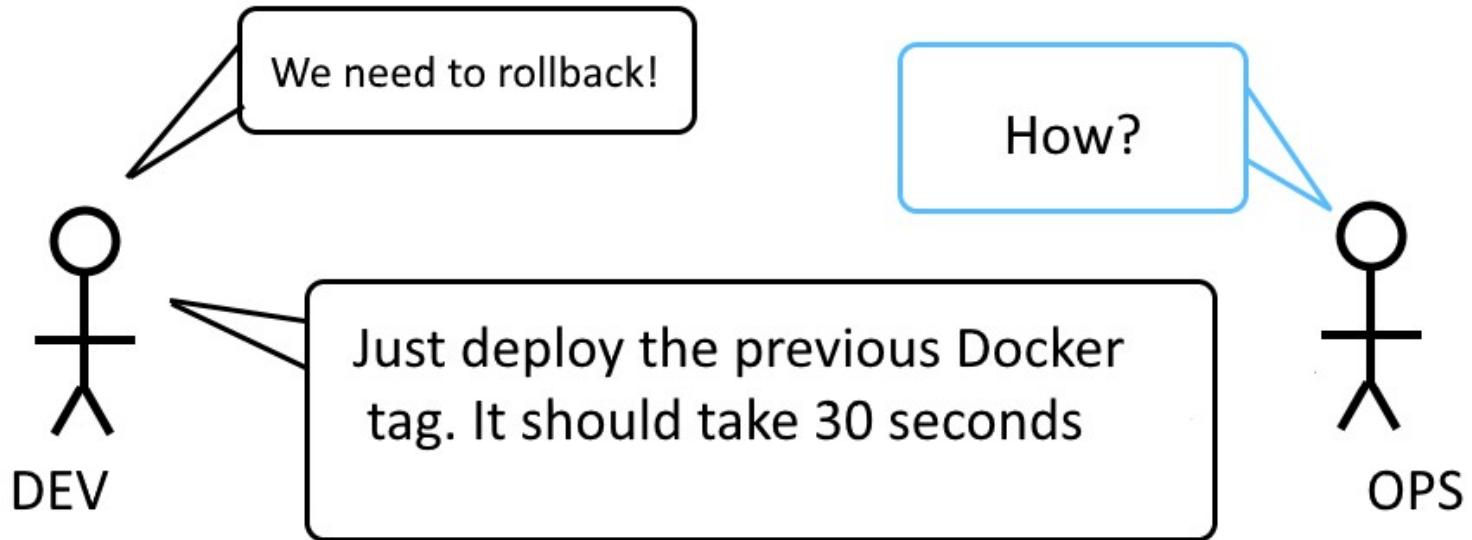


# Rolling back



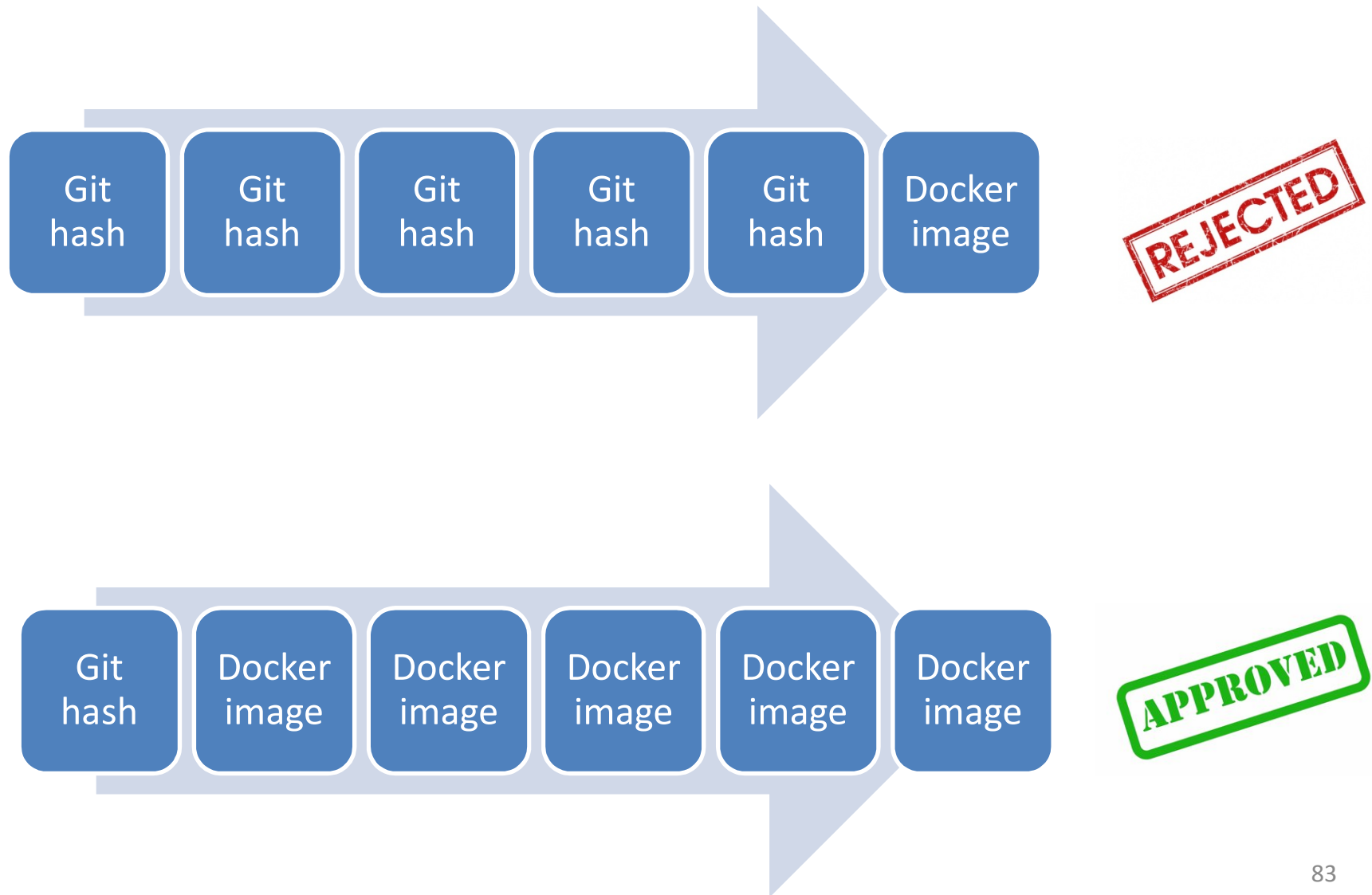
**REJECTED**

# Rolling back



**APPROVED**

# Change your CI/CD pipelines

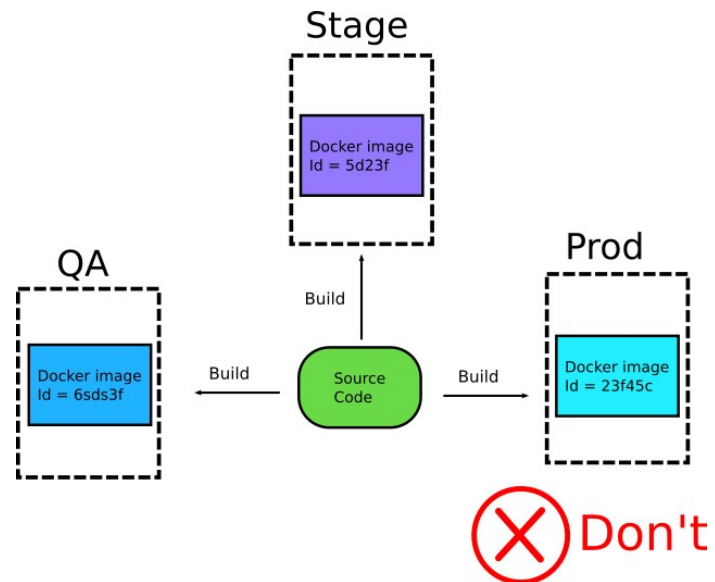


# Antipattern 8 – Hardcoding secrets

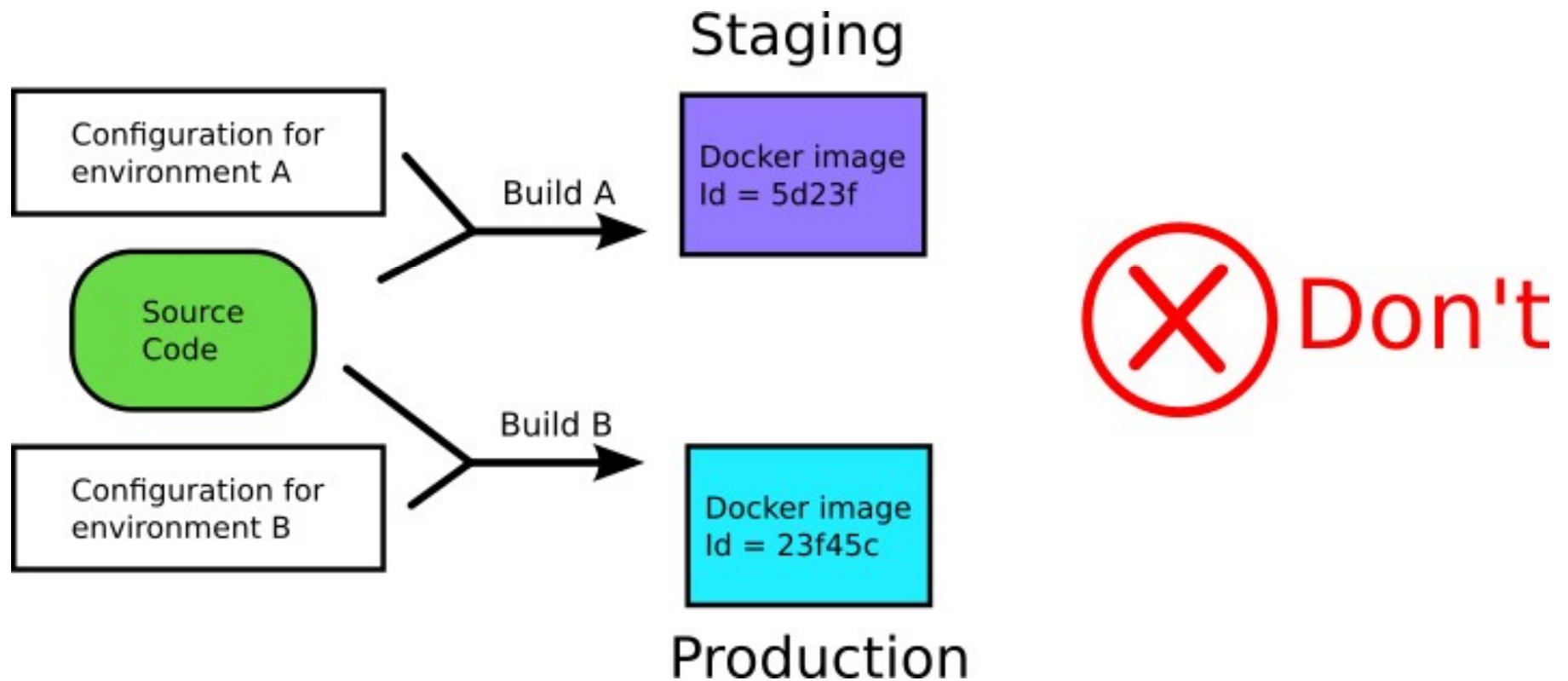


# Do not hardcode secrets and conf

- People still hardcode configuration (anti-pattern 1)
- People still create different images per environment (5)



# Do not hardcode secrets and conf



# Golden rule for Docker deployments

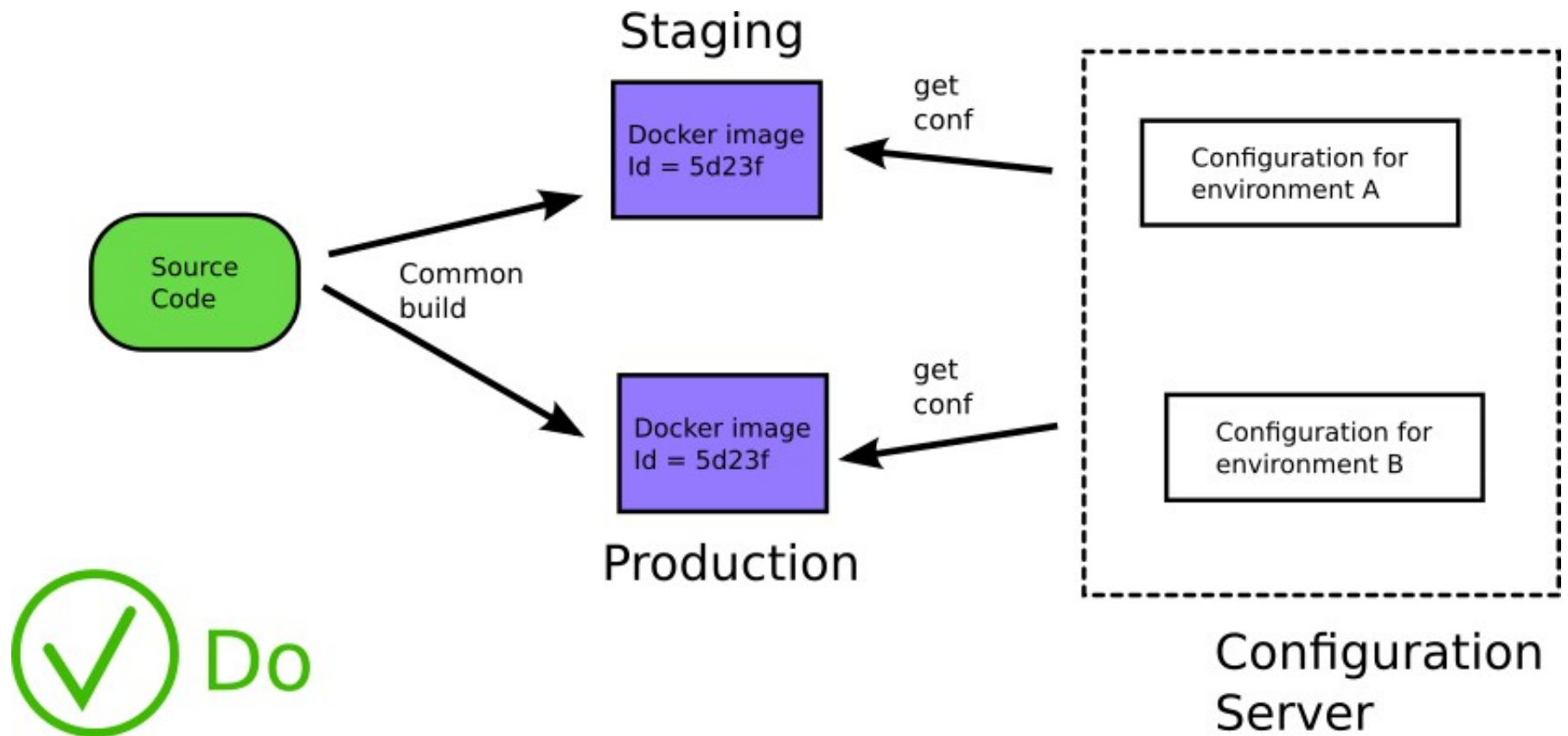
A Docker image should be  
configuration agnostic

# Golden rule for Docker deployments

A Docker image should fetch configuration during runtime (and not buildtime)



# Runtime configuration



# Runtime configuration



## THE TWELVE-FACTOR APP

### III. Config

#### Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of “config” does not include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

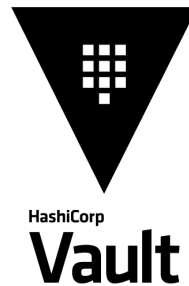
<https://12factor.net/config>

# Golden rule for deployments

A Docker image is built  
once for ALL  
environments

# Multiple solutions for conf/secrets

## Config-maps



Lyft Confidant



**CERBERUS**  
SECURE PROPERTY STORE FOR CLOUD APPLICATIONS

# If your Docker image...

- ...has hardcoded IPs
- ...contains passwords and secrets
- ...mentions specific URLs of other services
- ...has tags like foo-dev or foo-qa or foo-staging



**You are doing it wrong!**

# Antipattern 9 – Poor mans' CI



# “Shift your CI scripts to docker build”

```
1 # Run Sonar analysis
2 FROM newtmitch/sonar-scanner AS sonar
3 COPY src src
4 RUN sonar-scanner
5 # Build application
6 FROM node:11 AS build
7 WORKDIR /usr/src/app
8 COPY . .
9 RUN yarn install \
10 | yarn run lint \
11 | yarn run build \
12 | yarn run generate-docs
13 LABEL stage=build
14 # Run unit test
15 FROM build AS unit-tests
16 RUN yarn run unit-tests
17 LABEL stage=unit-tests
18 # Push docs to S3
19 FROM containerlabs/aws-sdk AS push-docs
20 ARG push-docs=false
21 COPY --from=build docs docs
22 RUN [[ "$push-docs" == true ]] && aws s3 cp -r docs s3://my-docs-bucket/
23 # Build final app
24 FROM node:11-slim
25 EXPOSE 8080
26 WORKDIR /usr/src/app
27 COPY --from=build /usr/src/app/node_modules node_modules
28 COPY --from=build /usr/src/app/dist dist
29 USER node
30 CMD ["node", "./dist/server/index.js"]
```



```
1 # Run Sonar analysis
2 FROM newtmitch/sonar-scanner AS sonar
3 COPY src src
4 RUN sonar-scanner
5 # Build application
6 FROM node:11 AS build
7 WORKDIR /usr/src/app
8 COPY . .
9 RUN yarn install \
10 | yarn run lint \
11 | yarn run build \
12 | yarn run generate-docs
13 LABEL stage=build
14 # Run unit test
15 FROM build AS unit-tests
16 RUN yarn run unit-tests
17 LABEL stage=unit-tests
18 # Push docs to S3
19 FROM containerlabs/aws-sdk AS push-docs
20 ARG push-docs=false
21 COPY --from=build docs docs
22 RUN [[ "$push-docs" == true ]] && aws s3 cp -r docs s3://my-docs-bucket/
23 # Build final app
24 FROM node:11-slim
25 EXPOSE 8080
26 WORKDIR /usr/src/app
27 COPY --from=build /usr/src/app/node_modules node_modules
28 COPY --from=build /usr/src/app/dist dist
29 USER node
30 CMD ["node", "./dist/server/index.js"]
```

Depend on Sonar  
Anti-pattern 1

Upload to S3/Side effect  
Anti-pattern 3

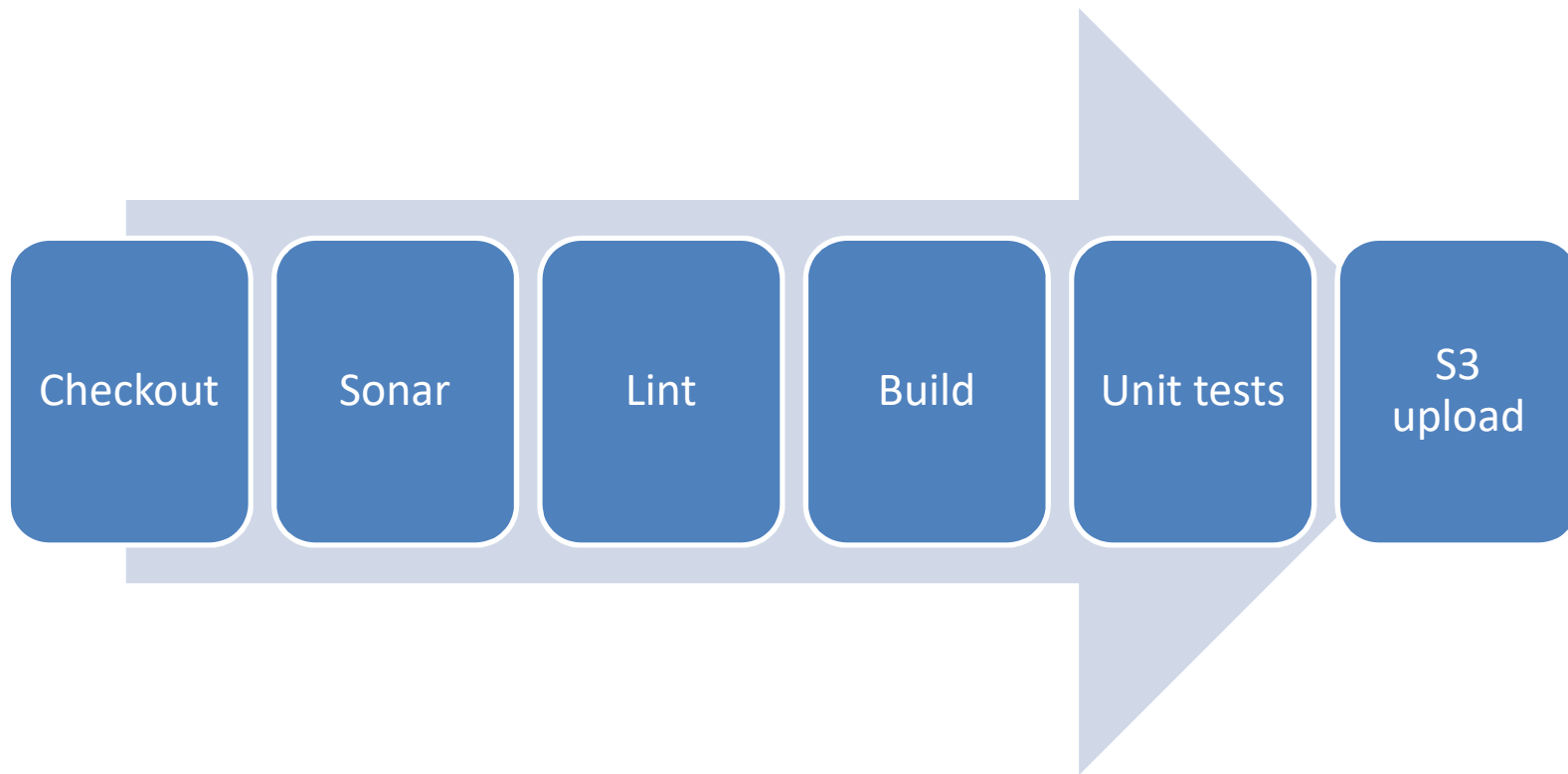
Both Dev/Deployment  
Anti-pattern 4



# Creating Dockerfiles that do too much

Dockerfiles are not  
glorified bash scripts

# Use CI/CD pipeline as intended



**APPROVED**

Don't abuse Docker as CI/CD

Side effects are only allowed in  
the CI/CD pipeline

# Antipattern 10 – Docker afterthought



# Creating Dockerfiles that do too little

## Don't use docker as a simple package format

# What does this dockerfile do?

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 ARG JAR_FILE
4 COPY ${JAR_FILE} app.jar
5 ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

How was the jar created???

What is the app version???



# Example taken from Spring Boot Guide



The screenshot shows a web browser window with the URL <https://spring.io/guides/gs/spring-boot-docker/>. The page features a dark navigation bar with the Spring logo and 'by Pivotal.' on the left, and 'PROJECTS', 'GUIDES', and 'BLOG' on the right. The 'GUIDES' link is highlighted in green. Below the navigation bar, the page content is on a light gray background. It starts with a green horizontal line, followed by the text 'GETTING STARTED'. The main heading is 'Spring Boot with Docker'. Below this, a paragraph states: 'This guide walks you through the process of building a Docker image for running a Spring Boot application.' The next section is titled 'What you'll build' and contains a paragraph: 'Docker is a Linux container management toolkit with a "social" aspect, allowing users to publish container images and consume those published by others. A Docker image is a recipe for running a containerized process, and in this guide we will build one for a simple Spring boot application.'



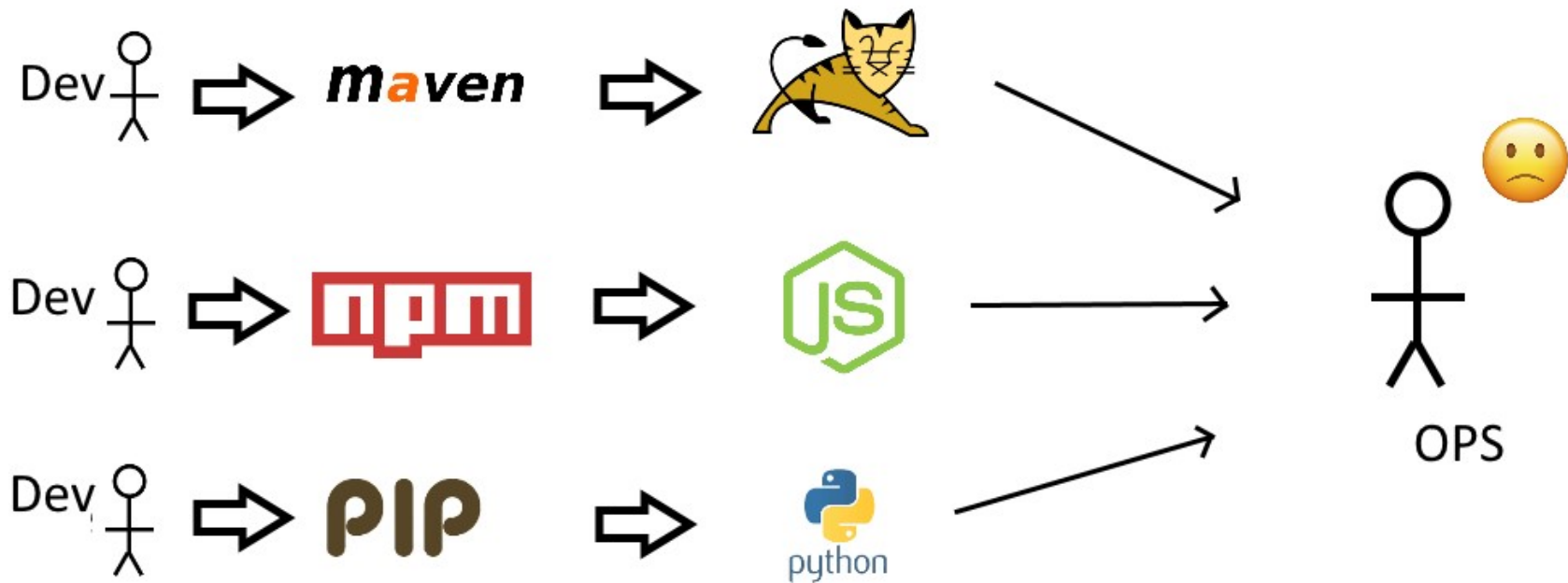
# Creating Dockerfiles that do too little

This dockerfile expects a full  
Java environment

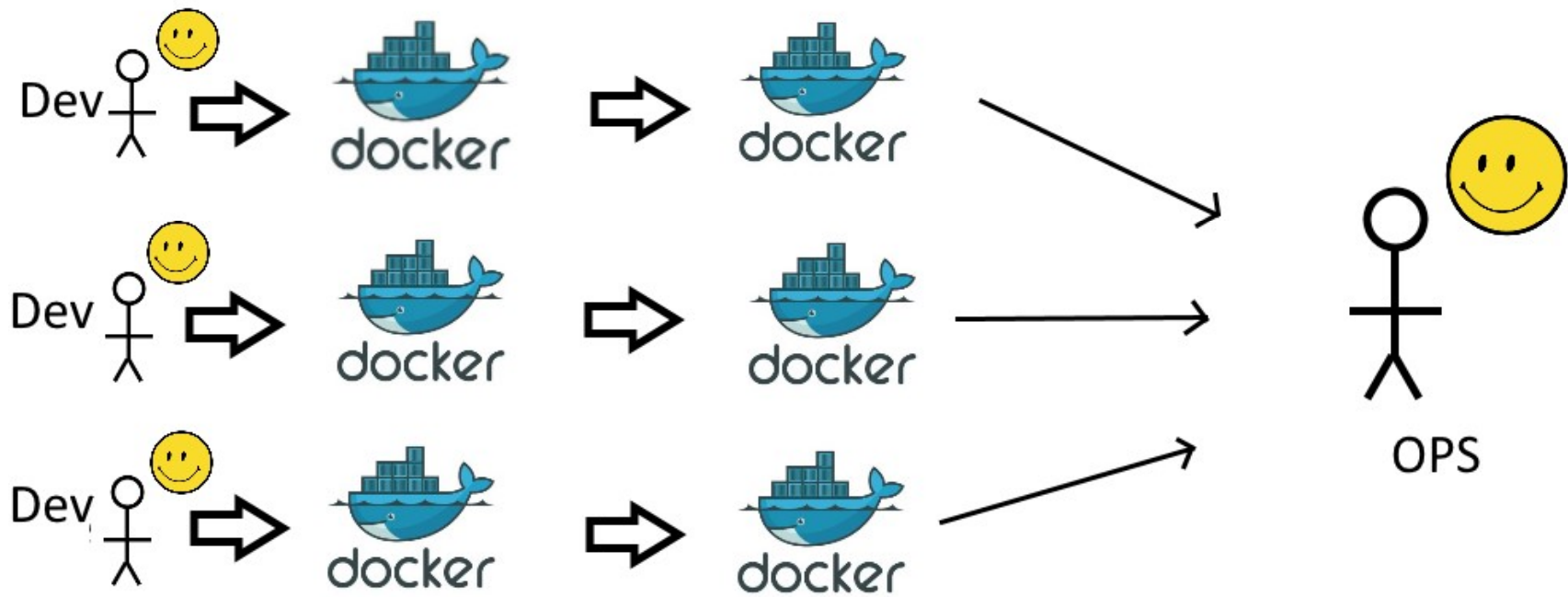




# Before containers



# Containers in CI/CD



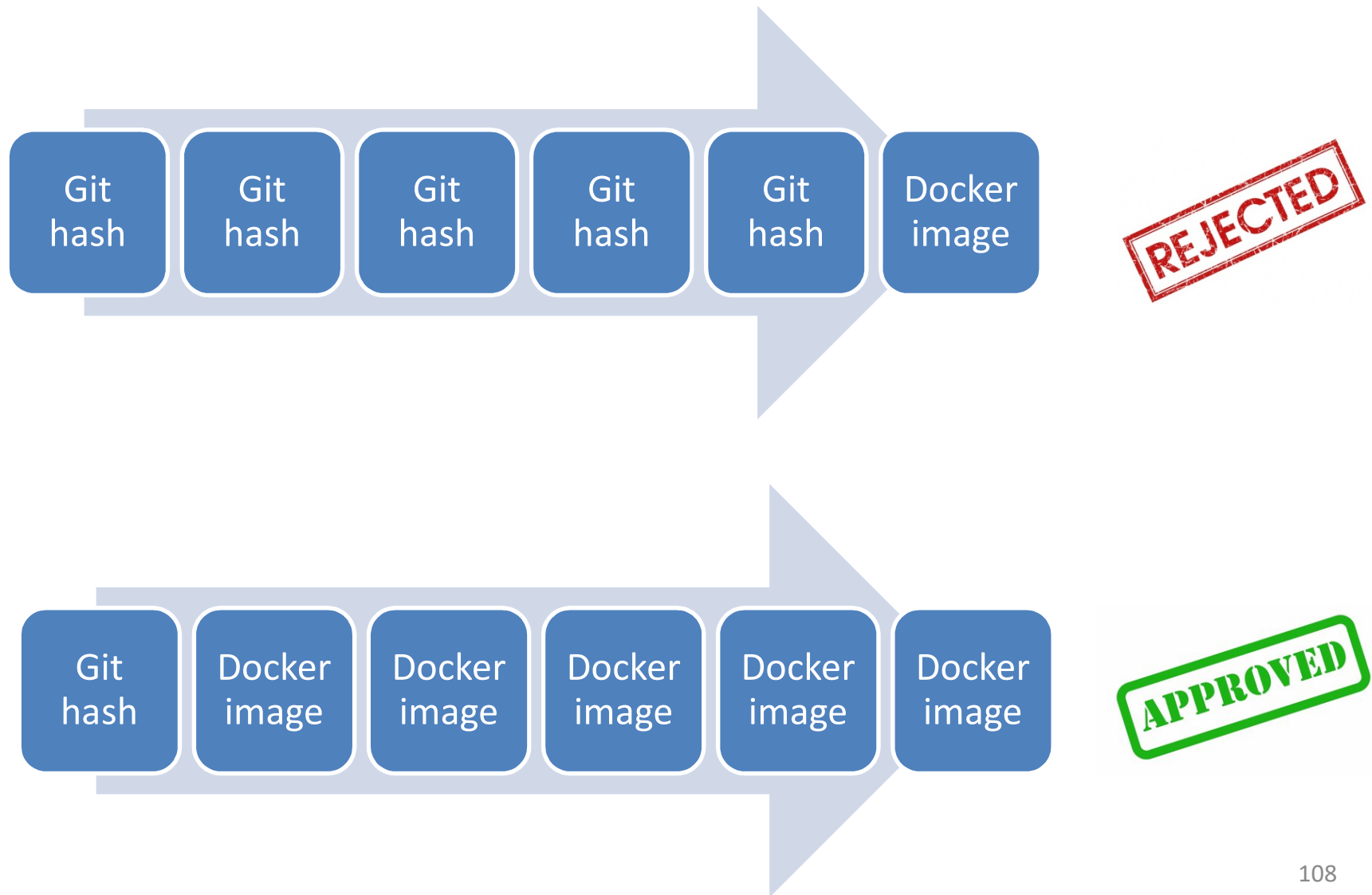
# Fixing the Dockerfile

```
1 FROM openjdk:8-jdk-alpine
2 COPY pom.xml /tmp/
3 COPY src /tmp/src/
4 WORKDIR /tmp/
5 RUN ./gradlew build
6 COPY /tmp/build/app.war /app.jar
7 ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

## Exercise 1- multistage

## Exercise 2 – Layer caching

# Change your CI/CD pipelines



# One command for all

- `docker build . -t my-tag` (developer)
- `docker build . -t my-tag` (operator)
- `docker build . -t my-tag` (ci/cd server)
- `docker build . -t my-tag` (test engineer)
- `docker build . -t my-tag` (release engineer)
- `docker build . -t my-tag` (security analyst)

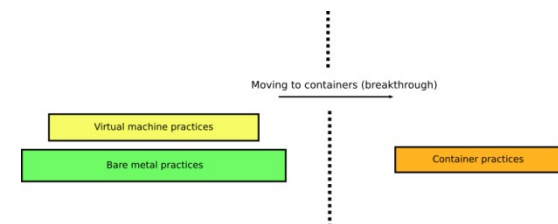
# Conclusion

10 Docker antipatterns for  
MANAGING docker images

# Antipattern 1

Attempting to use VM  
practices with Containers

Solution: Understand what  
containers are

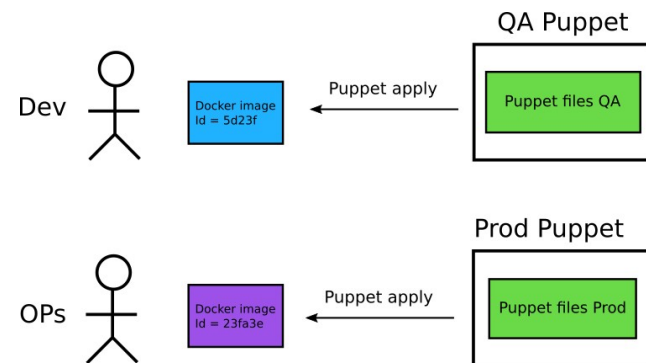


Do

# Antipattern 2

Creating Docker files that are not transparent

Solution: Write Dockerfiles from scratch





# Antipattern 3

Building Docker images with  
side effects

Solution: Move side effects to  
CI/CD server



# Antipattern 4

Confusing images used for  
deployment with development

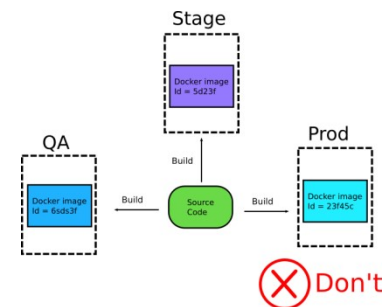
Solution: Don't ship development  
tools to production



# Antipattern 5

Building different images per environment

Solution: Build image only once and promote



# Antipattern 6

Building images in production servers

Solution: Use 3 docker registries



# Antipattern 7

Promoting Git hashes between teams

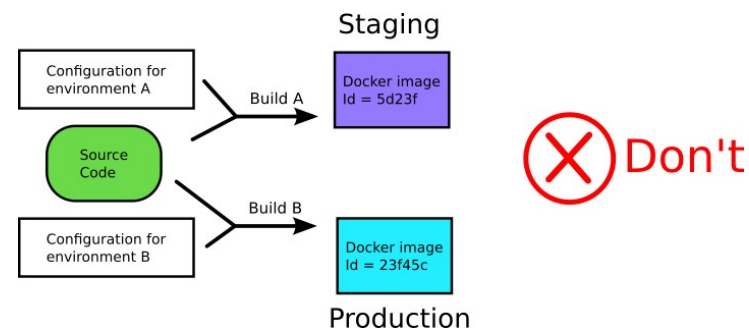
Solution: Promote container images between teams



# Antipattern 8

Hardcoding secrets in  
containers

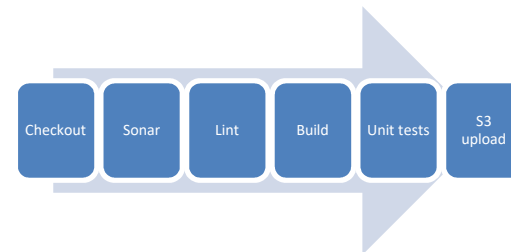
Solution: Build image once and  
fetch configuration



# Antipattern 9

Using Docker as poor man's  
CI/CD

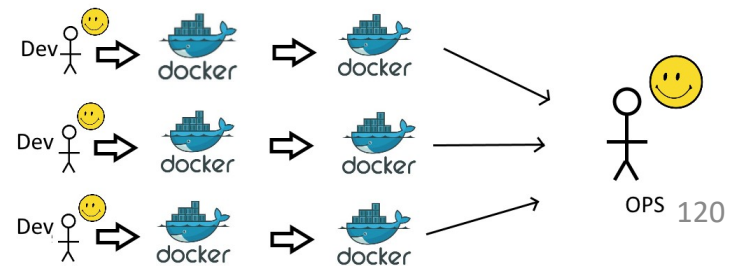
Solution: Use CI/CD as a CI/CD  
solution



# Antipattern 10

Using Docker as dumb packaging method

Solution: Create dockerfiles that package/compile on their own





# The end



<https://codefresh.io/containers/docker-anti-patterns/>