codefresh

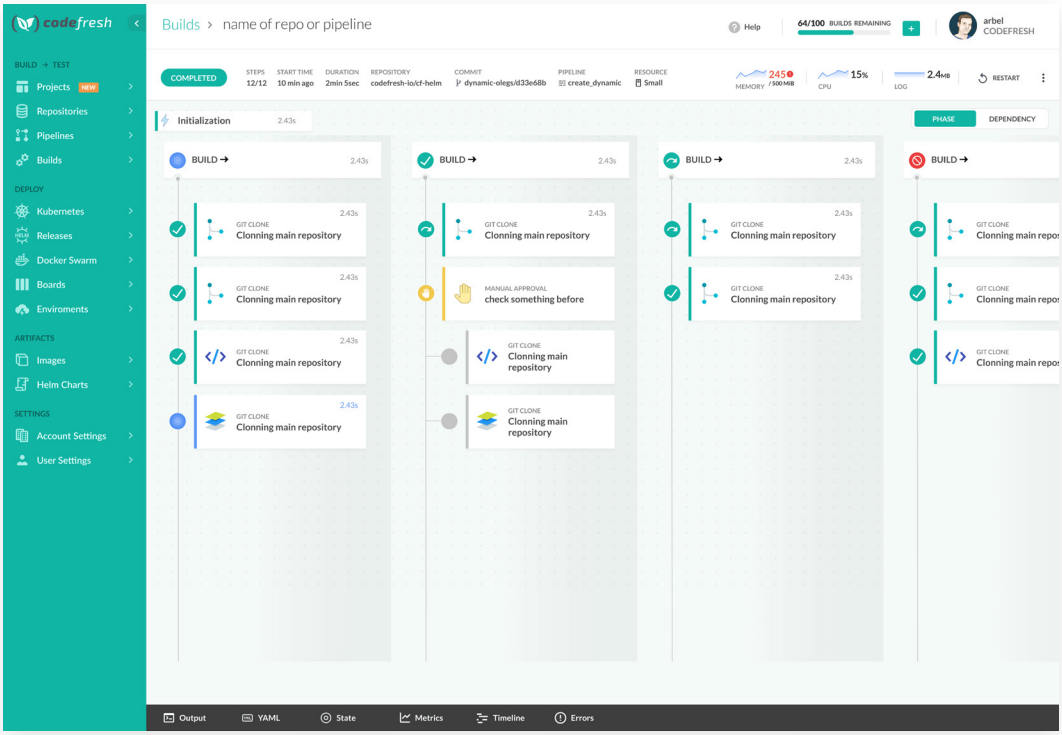
CONTINUOUS DEPLOYMENT / DELIVERY

Docker Anti-Patterns

Table of Contents

codefresh

Codefresh is the only Continuous Integration/Delivery platform designed specifically for microservices and containers running on Kubernetes.



Codefresh includes comprehensive built-in support for Helm charts and deployments and even ofers a private free Helm repository with each account. Combined with the private Docker registry and dedicated Kubernetes dashboards, Codefresh is an one-stop-shop for microservice development.

Container usage is exploding. Even if you are not yet convinced that Kubernetes is the way forward, it is very easy to add value just by using Docker on its own. Containers can now simplify both deployments and CI/CD pipelines.

The official Docker best practices page is highly technical and focuses more on the structure of the Dockerfile instead of generic information on how to use containers in general.
Every Docker newcomer will at some point understand the usage of Docker layers, how they are cached, and how to create small Docker images. Multi-stage builds are not rocket science either. The syntax of Dockerfiles is fairly easy to understand.

However, the main problem of container usage is the inability of companies to look at the bigger picture and especially the immutable character of containers/images. Several companies in particular attempt to convert their existing VM-based processes to containers with dubious results. There is a wealth of information on low-level details of containers (how to create them and run them), but very little information on high level best practices.

To close this documentation gap, I present to you a list of high-level Docker best-practices. Since it is impossible to cover the internal processes of every company out there I will instead explain bad practices (i.e. what you should not do). Hopefully, this will give you some insights on how you should use containers.

Here is the complete list of bad practices that we will examine:

1. Attempting to use VM practices on containers.
2. Creating Docker files that are not transparent.
3. Creating Dockerfiles that have side effects.
4. Confusing images used for deployment with those used for development.
5. Building different images per environment
6. Pulling code from git into production servers and building images on the fly.
7. Promoting git hashes between teams.
8. Hardcoding secrets into container images.
9. Using Docker as poor man's CI/CD.
10. Assuming that containers are a dumb packaging method.

ANTI-PATTERN 1

Treating Docker containers as Virtual Machines

Before going into some more practical examples, let's get the basic theory out of the way first. Containers are not Virtual Machines. At first glance, they might look like they behave like VMs but the truth is completely different. Stackoverflow and related forums are filled with questions like:

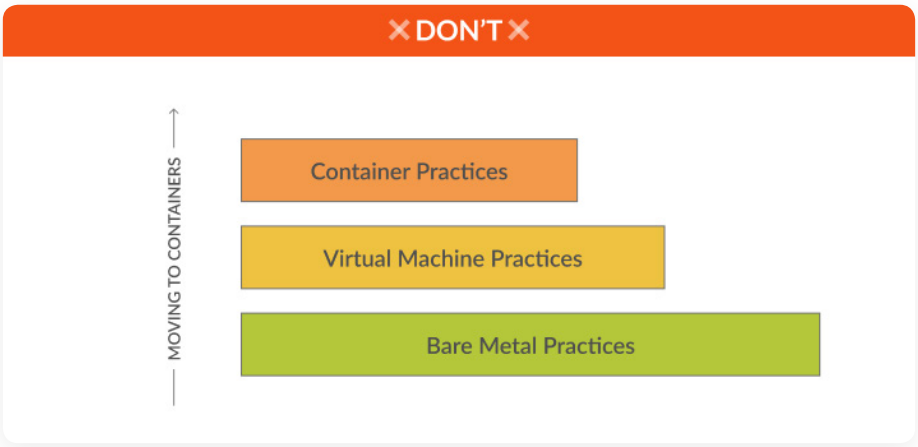
1. How to I update applications running inside containers?
2. How do I ssh in a Docker container?
3. How do I get logs/files from a container?
4. How do I apply security fixes inside a container?
5. How do I run multiple programs in a container?

All these questions are technically correct, and the people that have answered them have also given technically correct answers. However, all these questions are the canonical example of the XY problem. The real question behind these questions is:

"How can I unlearn all my VM practices and processes and change my workflow to work with immutable, short-lived, stateless containers instead of mutable, long-running, stateful VMs?"

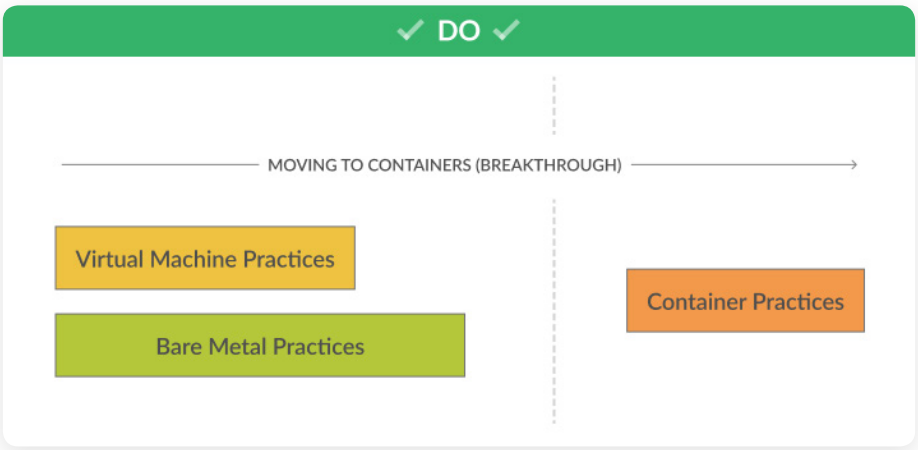
Many companies out there are trying to reuse the same practices/tools/knowledge from VMs in the container world. Some companies were even caught completely off-guard as they had not even finished their bare-metal-to-vm migration when containers appeared.

Unlearning something is very difficult. Most people that start using containers see them initially as an extra abstraction layer on top of their existing practices:



Containers are not VMs

In reality, containers require a completely different view and change of existing processes. You need to rethink **all** your CI/CD processes when adopting containers.



Containers require a new way of thinking

There is no easy fix for this anti-pattern other than reading about the nature of containers, their building blocks, and their history (going all the way back to the venerable chroot).

If you regularly find yourself wanting to open ssh sessions to running containers in order to "upgrade" them or manually get logs/files out of them you are definitely using Docker in the wrong way and you need to do some extra reading on how containers work.

ANTI-PATTERN 2

Creating Docker images that are not transparent

A Dockerfile should be transparent and self-contained. It should describe all the components of an application in plain sight. Anybody should be able to get the same Dockerfile and recreate the same image. It is ok if the Dockerfile downloads extra libraries (in a versioned and well-controlled manner) but creating Dockerfiles that perform “magic” steps should be avoided.

Here is a particularly bad example:

```
FROM alpine:3.4

RUN apk add --no-cache \
      ca-certificates \
      pciutils \
      ruby \
      ruby-irb \
      ruby-rdoc \
      && \
    echo http://dl-4.alpinelinux.org/alpine/edge/
community/ >> /etc/apk/repositories && \
    apk add --no-cache shadow && \
    gem install puppet:"5.5.1" facter:"2.5.1" && \
    /usr/bin/puppet module install puppetlabs-apk

# Install Java application
RUN /usr/bin/puppet agent --onetime --no-daemonize

ENTRYPOINT ["java","-jar","/app/spring-boot-application.
jar"]
```

Now don’t get me wrong. I love Puppet as it is a great tool (or Ansible, or Chef for that matter). Misusing it for application deployments might have been easy with VMs, but with containers it is disastrous.

First of all, it makes this Dockerfile location-dependent. You have to build it on a computer that has access to the production Puppet server. Does your workstation have access to the production puppet server? If yes, should your workstation really have access to the production puppet server?

But the biggest problem is that this Docker image cannot be easily recreated. Its contents depend on what the puppet server had at the time of the initial build. If you build the same Dockerfile today you might get a completely different image. And if you don't have access to the puppet server or the puppet server is down you cannot even build the image in the first place. You don't even know what is the version of the application if you don't have access to the puppet scripts.

The team that created this Dockerfile was just lazy. There was already a puppet script for installing the application in a VM. The Dockerfile was just retrofitted to do the same thing (see the previous anti-pattern).

The fix here is to have minimal Dockerfiles that describe explicitly what they do. Here is the same application with the "proper" Dockerfile.

```
FROM alpine:3.4

ENV MY_APP_VERSION="3.2"

RUN apk add --no-cache \
      ca-certificates

WORKDIR /app
ADD  http://artifactory.mycompany.com/releases/${MY_APP_
VERSION}/spring-boot-application.jar .

ENTRYPOINT ["java","-jar","/app/spring-boot-application.
jar"]
```

Notice that:

1. There is no dependency on puppet infrastructure. The Dockerfile can be built on any developer machine that has access to the binary repository
2. Versions of the software are explicitly defined.
3. It is very easy to change the version of the application by editing only the Dockerfile (instead of puppet scripts).

This was just a very simple (and contrived) example. I have seen many Dockerfiles in the wild that depend on "magic" recipes with special requirements for the time and place they can be built. Please don't write your Dockerfiles in this manner, as developers (and other people who don't have access to all systems) will have great difficulties creating Docker images locally.

An even better alternative would be if the Dockerfile compiled the source Java code on its own (using multi-stage builds). That would give you even greater visibility on what is happening in the Docker image.

ANTI-PATTERN 3

Creating Dockerfiles that have side effects

Let's imagine that you are an operator/SRE working at very big company where multiple programming languages are used. It would be very difficult to become an expert in all the programming languages and build systems out there.

This is one of the major advantages of adopting containers in the first place. You should be able to download any Dockerfile from any development team and build it without really caring about side effects (because there shouldn't be any).

Building a Docker image should be an idempotent operation. It shouldn't matter if you build the same Dockerfile one time or a thousand times. Or if you build it on a CI server first and then on your workstation.

Yet, there are several Dockerfiles out there that during the build phase…

1. perform git commits or other git actions,
2. clean up or tamper with database data,
3. or call other external services with POST/PUT operations.

Containers offer isolation as far as the host filesystem is concerned but there is nothing protecting you from a Dockerfile that contains a RUN directive with curl POSTING an HTTP payload to your intranet.

Here is a simple example where a Dockerfile both packages (a safe action) and publishes (an unsafe action) an npm application in a single run.

```
FROM node:9
WORKDIR /app

COPY package.json ./package.json
COPY package-lock.json ./package-lock.json
RUN npm install
COPY . .

RUN npm test

ARG npm_token

RUN echo "//registry.npmjs.org/:_authToken=${npm_token}" >
.npmrc
RUN npm publish --access public

EXPOSE 8080
CMD [ "npm", "start" ]
```

This Docker file confuses two unrelated concerns, releasing a version of the application, and creating a Docker image for it. Maybe sometimes these two actions happen indeed together at the same time, but this is no excuse for polluting a Dockerfile with side effects.

Docker is **NOT** a generic CI system and it was never meant to be one. Don't abuse Dockerfiles as glorified bash scripts that have unlimited power. Having side effects while containers are running is ok. Having side effects during container build time is not.

The solution is to simplify your Dockerfiles and make sure that they only contain idempotent operations such as:

- Cloning source code
- Downloading dependencies
- Compiling/packaging code
- Processing/Minifying/Transforming local resources
- Running scripts and editing files on the container filesystem only

Also, keep in mind the ways Docker caches filesystem layers. Docker assumes that if a layer and the ones before it have not "changed" they can be reused from cache. If your Dockerfile directives have side effects you essentially break the Docker caching mechanism.

```
FROM node:10.15-jessie

RUN apt-get update && apt-get install -y mysql-client && rm
-rf /var/lib/apt

RUN mysql -u root --password="" < test/prepare-db-for-
tests.sql

WORKDIR /app

COPY package.json ./package.json
COPY package-lock.json ./package-lock.json
RUN npm install
COPY . .

RUN npm integration-test

EXPOSE 8080
CMD [ "npm", "start" ]
```

Let's say that you try to build this Dockerfile and your unit tests fail. You make a change to the source code and you try to rebuild again. Docker will assume that the layer that clears the DB is already "run" and it will reuse the cache. So your unit tests will now run in a DB that isn't cleaned and contains data from the previous run.

In this contrived example, the Dockerfile is very small and it is very easy to locate the statement that has side effects (the mysql command) and move it to the correct place in order to fix layer caching. But in a real Dockerfile with many commands, trying to hunt down the correct order of RUN statements is very difficult if you don't know which have side effects and which do not.

Your Dockerfiles will be much simpler if all actions they perform are read-only and with local scope.

ANTI-PATTERN 4

Confusing images that are used for development with those that are used for deployment

In any company that has adopted containers, there are usually two separate categories of Docker images. First, there are the images that are used as the actual deployment artifact sent to production servers.

The deployment images should contain:

1. The application code in minified/compiled form plus its runtime dependencies.
2. Nothing else. Really nothing else

The second category is the images used for the CI/CD systems or developers and might contain:

1. The source code in its original form (i.e. unminified)
2. Compilers/minifiers/transpilers
3. Testing frameworks/reporting tools
4. Security scanning, quality scanning, static analyzers
5. Cloud integration tools
6. Other utilities needed for the CI/CD pipeline

It should be obvious that these categories of container images should be handled separately as they have different purposes and goals. Images that get deployed to servers should be minimal, secure and battle-hardened. Images that get used in the CI/CD process are never actually deployed anywhere so they have much less strict requirements (for size and security).

Yet for some reason, people do not always understand this distinction. I have seen several companies who try to use the same Docker image both for development

and for deployment. Almost always what happens is that unrelated utilities and frameworks end up in the production Docker image.

There are exactly 0 reasons on why a production Docker image should contain git, test frameworks, or compilers/minifiers.

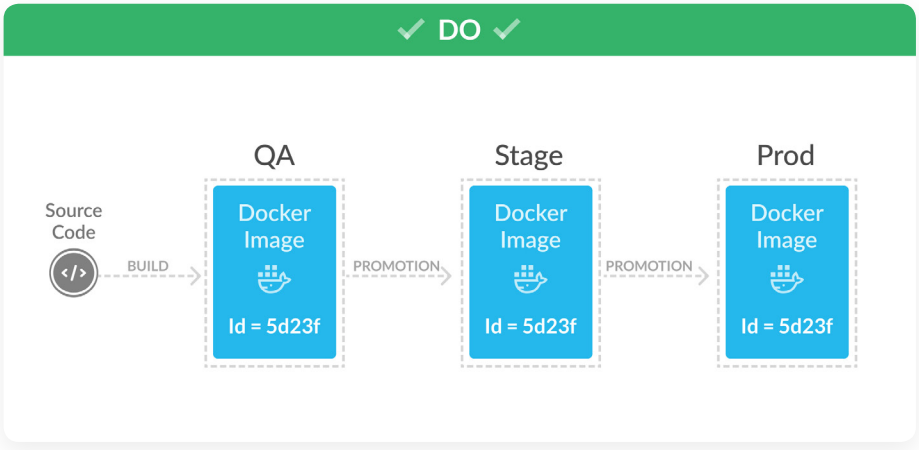
The promise of containers as a universal deployment artifact was always about using the same deployment artifact between different environments and making sure that what you are testing is also what you are deploying (more on this later). But trying to consolidate local development with production deployments is a losing battle.

In summary, try to understand the roles of your Docker images. Each image should have a single role. If you are shipping test frameworks/libraries to production you are doing it wrong. You should also spend some time to learn and use multi-stage builds.

ANTI-PATTERN 5

Using different images for each environment (qa, stage, production)

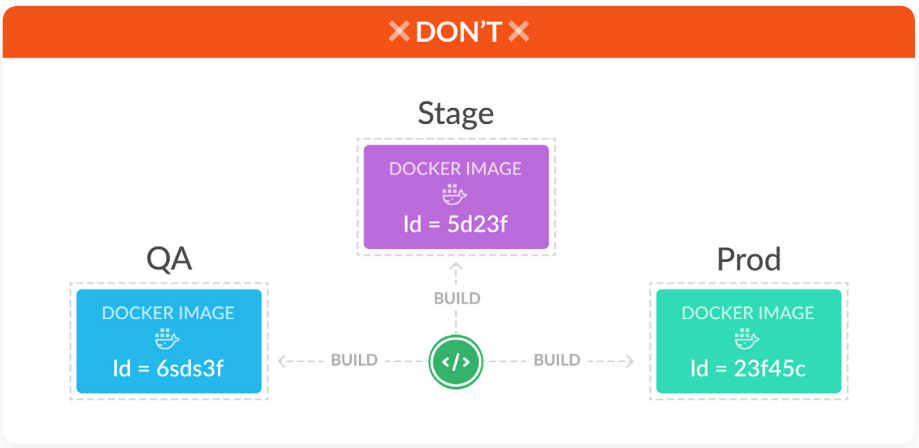
One of the most important advantages of using containers is their immutable attribute. This means that a Docker image should be built only once and then promoted to various environments until it reaches production.



Promoting the same Docker image

Because the exact same image is promoted as a single entity, you get the guarantee that what you are testing in one environment is the same as the other.

I see a lot of companies that build different artifacts for their environments with slightly different versions of code or configuration.



Different image per environment

This is problematic because there is no guarantee that images are "similar enough" to verify that they behave in the same manner. It also opens a lot of possibilities for abuse, where developers/operators are sneaking in extra debugging tools in the non-production images creating an even bigger rift between images for different environments.

Instead of trying to make sure that your different images are the same as possible, it is far easier to use a single image for all software lifecycle phases.

Note that it is perfectly normal if the different environments use different settings (i.e. secrets and configuration variables) as we will see later in this article. Everything else, however, should be exactly the same.

ANTI-PATTERN 6

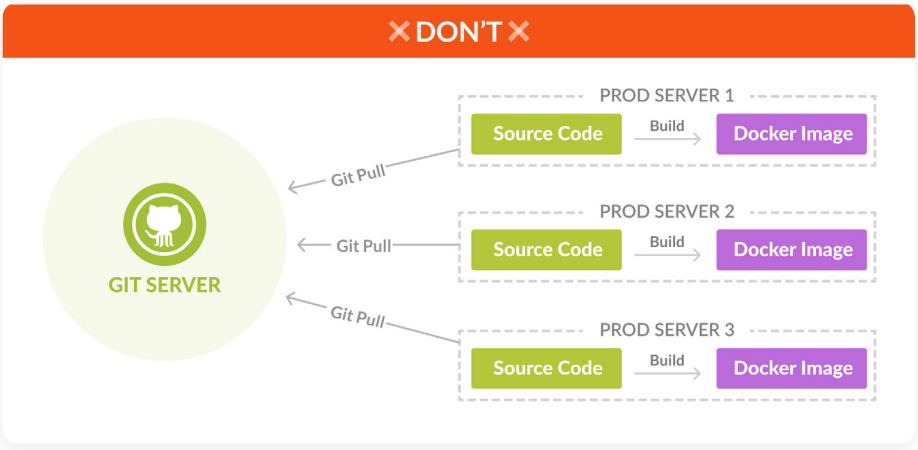
Creating Docker images on production servers

The Docker registry serves as a catalog of existing applications that can be re-deployed at any time to any additional environments. It is also a central location of application assets with extra metadata along with previous historical versions of the same application. It should be very easy to choose a specific tag of a Docker image and deploy it to any environment.

One of the most flexible ways of using Docker registries is by promoting images between them. An organization has at least two registries (the development one and the production one). A Docker image should be built once (see previous anti-pattern) and placed in the development registry. Then, once integration tests, security scans, and other quality gates verify its correct functionality, the image can be promoted to the production Docker registry to be sent to production servers or Kubernetes clusters.

Is also possible to have different organizations for Docker registries per region/location or per department. The main point here is that the canonical way for Docker deployments also includes a Docker registry. Docker registries serve both as an application asset repository as well as intermediate storage before an application is deployed to production.

A very questionable practice is the complete removal of Docker registries from the lifecycle and the pushing of source code directly to production servers.



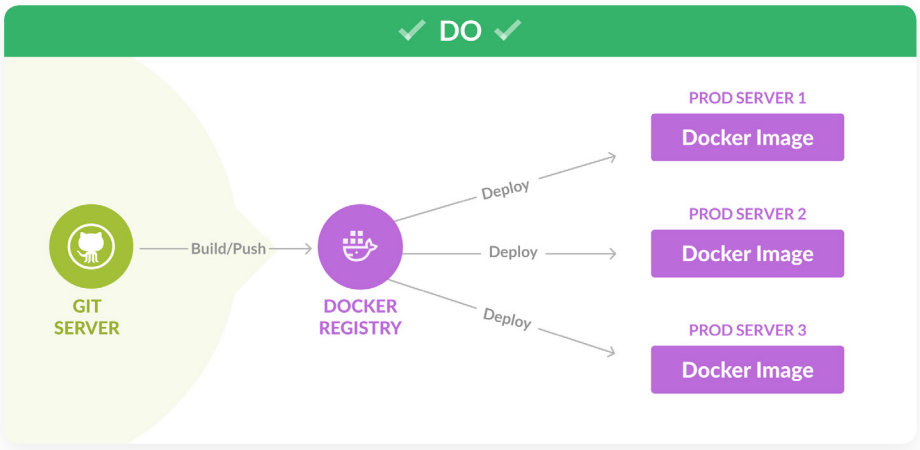
Building images in production servers

Production servers use “git pull” to get the source code and then Docker build to create an image on the fly and run it locally (usually with Docker-compose or other custom orchestration). This “deployment method” essentially employs multiple anti-patterns all at once!

This deployment practice suffers from a lot of issues, starting with security. Production servers should not have inbound access to your git repositories. If a company is serious about security, this pattern will not even fly with the security committee. There is also no reason why production servers should have git installed. Git (or any other version control system) is a tool intended for developer collaboration and not an artifact delivery solution.

But the most critical issue is that with this “deployment method” you bypass completely the scope of Docker registries. You no longer know what Docker image is deployed on your servers as there is no central place that holds Docker images anymore.

This deployment method might work ok in a startup, but will quickly become inefficient in bigger installations. You need to learn how to use Docker registries and the advantages they bring (also related to security scanning of containers).



Using a Docker registry

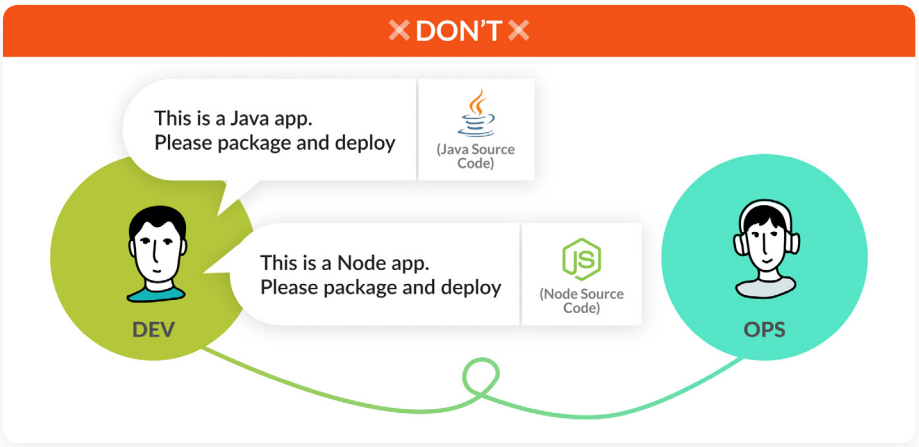
Docker registries have a well-defined API, and there are several open-source and proprietary products that can be used to set-up one within your organization.

Notice also that with Docker registries your source code securely resides behind the firewall and never leaves the premise

ANTI-PATTERN 7

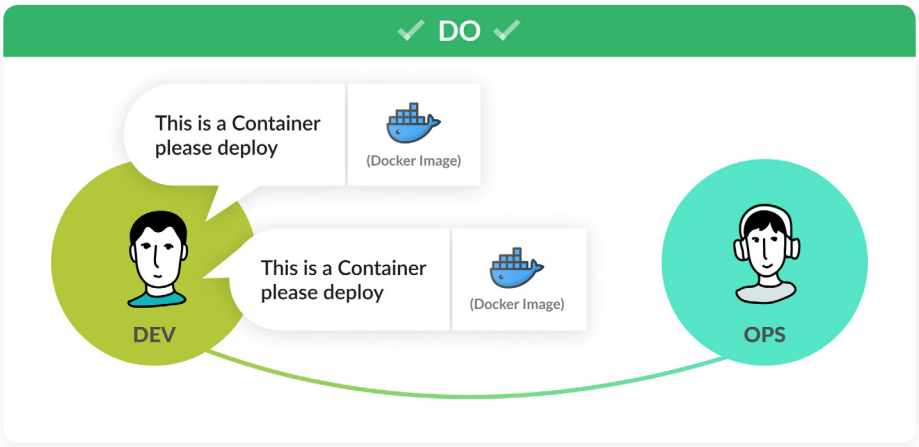
Working with git hashes instead of Docker images

A corollary to the previous two anti-patterns is that once you adopt containers, your Docker registry should become the single point of truth for everything. People should talk in terms of Docker tags and image promotions. Developers and operators should use containers as their common language. The hand-over entity between teams should be a container and not a git hash.



Talking about git hashes

This comes in contrast with the old way of using Git hashes as "promotion" artifacts. The source code is of course important, but re-building the same hash over and over in order to promote it is a waste of resources (see also anti-pattern 5). Several companies think that containers should only be handled by operators, while developers are still working with just the source code. This could not be further from the truth. Containers are the perfect opportunity for developers and operators to work together.



Talking about git hashes

Ideally, operators should not even care about what goes on with the git repo of an application. All they need to know is if the Docker image they have at hand is ready to be pushed to production or not. They should not be forced to rebuild a git hash in order to get the same Docker image that developers were using in pre-production environments.

You can understand if you are the victim of this anti-pattern by asking operators in your organization. If they are forced to become familiar with application internals such as build systems or test frameworks that normally are not related to the actual runtime of the application, they have a heavy cognitive load which is otherwise not needed for daily operations.

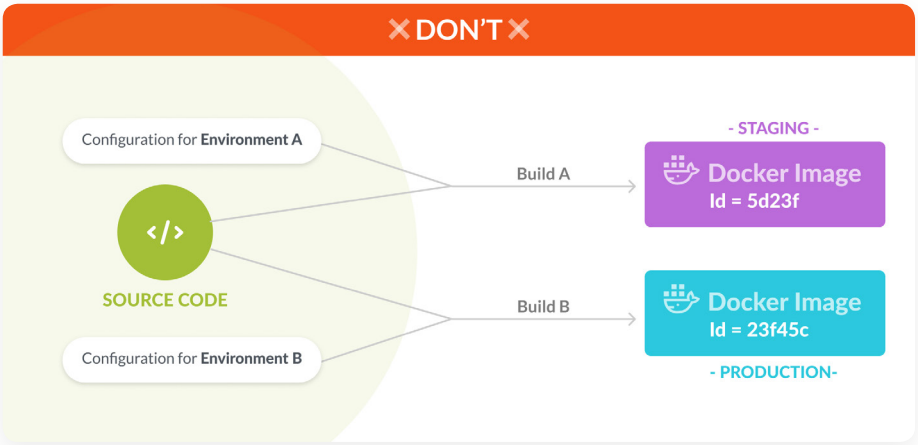
ANTI-PATTERN 8

Hardcoding secrets and configuration into container images

This anti-pattern is closely related to Anti-pattern 5 (different images per environment). In most cases when I ask companies why they need different images for qa/staging/production, the usual answer is that they contain different configurations and secrets.

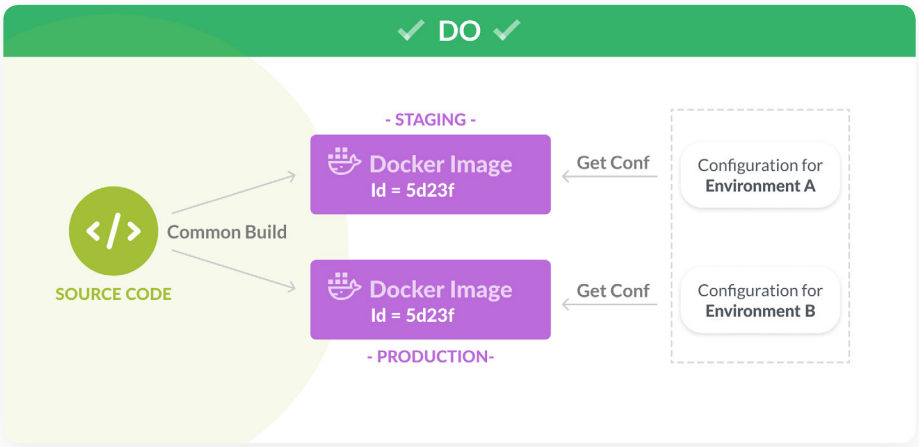
This not only breaks the main promise of Docker (deploy what you tested) but also makes all CI/CD pipelines very complex as they have to manage secrets/ configuration during build time.

The anti-pattern here is, of course, the hard-coding of configurations. Applications should not have embedded configurations. This should not be news for anybody who is familiar with 12-factor apps.



Talking about git hashes

Your applications should fetch configuration during runtime instead of build time. A Docker image should be configuration agnostic. Only during runtime configuration should be "attached" to the container. There are many solutions for this and most clustering/deployment systems can work with a solution for runtime configuration (configmaps, zookeeper, consul etc) and secrets (vault, keywhiz, confidant, cerberus).



Loading configuration during runtime

If your Docker image has hardcoded IPs and/or credentials you are definitely doing it wrong.

ANTI-PATTERN 9

Creating Docker files that do too much

I have come across articles who suggest that Dockerfiles should be used as a poor man's CI solution. Here is an actual example of a single Dockerfile.

```
# Run Sonar analysis
FROM newtmitch/sonar-scanner AS sonar
COPY src src
RUN sonar-scanner
# Build application
FROM node:11 AS build
WORKDIR /usr/src/app
COPY . .
RUN yarn install \
 yarn run lint \
 yarn run build \
 yarn run generate-docs
LABEL stage=build
```



```
# Run unit test
FROM build AS unit-tests
RUN yarn run unit-tests
LABEL stage=unit-tests
# Push docs to S3
FROM containerlabs/aws-sdk AS push-docs
ARG push-docs=false
COPY --from=build docs docs
RUN [[ "$push-docs" == true ]] && aws s3 cp -r docs s3://
my-docs-bucket/
# Build final app
FROM node:11-slim
EXPOSE 8080
WORKDIR /usr/src/app
COPY --from=build /usr/src/app/node_modules node_modules
COPY --from=build /usr/src/app/dist dist
USER node
CMD ["node", "./dist/server/index.js"]
```

While at first glance this Docker file might look like a good use of multi-stage builds, it is essentially a combination of previous anti-patterns.

- It assumes the presence of a SonarQube server (anti-pattern 2).
- It has potential side effects as it can push to S3 (anti-pattern 3).
- It acts both as a development as well as a deployment image (anti- pattern 4).

Docker is not a CI system on its own. Container technology can be used as part of a CI/CD pipeline, but this technique is something completely different. Don't confuse commands that need to run in the Docker container with commands that need to run in a CI build job.

The author of this Dockerfile advocates that you should use build arguments that interact with the labels and switch on/off specific build phases (so you could disable sonar for example). But this approach is just adding complexity for the sake of complexity.

The way to fix this Dockerfile is to split it into 5 other Dockerfiles. One is used for the application deployment and all others are different pipeline steps in your CI/CD pipeline. A single Dockerfile should have a single purpose/goal

ANTI-PATTERN 10

Creating Docker files that do too little

Because containers also include their dependencies, they are great for isolating library and framework versions per application. Developers are already familiar with the issues of trying to install multiple versions of the same tool on their workstation. Docker promises to solve this problem by allowing you to describe in your Dockerfile exactly what your application needs and nothing more.

But this Docker promise only holds true if you actually employ it. As an operator, I should not really care about the programming tool you use in your Docker image. I should be able to create a Docker image of a Java application, then a Python one and then a NodeJs one, without actually having a development environment for each language on my laptop.

A lot of companies however still see Docker as a dumb package format and just use it to package a finished artifact/application that was already created outside of the container. This anti-pattern is very famous with Java heavy organizations and even official documentation seems to promote it.

Here is the suggested Dockerfile from the official Spring Boot Docker guide.

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

This Dockerfile just packages an existing jar file. How was the Jar file created? Nobody knows. It is not described in the Dockerfile. If I am an operator I am forced to install all Java development libraries locally just to build this Dockerfile. And if you work in an organization that works with multiple programming languages this process gets quickly out of hand not only for operators but also for build nodes.

I am using Java as an example here but this anti-pattern is present in other situations as well. Dockerfiles that don't work unless you have first performed an

“npm install” locally first are a very common occurrence.

The solution to this anti-pattern is the same for anti-pattern 2 (Dockerfiles that are not self-contained). Make sure that your Dockerfiles describe the whole process of something. Your operators/SREs will love you even more if you follow this approach. In the case of the Java example before the Dockerfile should be modified as below:

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

This Dockerfile described exactly how the application is created and can be run by anybody on any workstation without the need for local Java installation. You can improve this Dockerfile even further with multi-stage builds (exercise for the reader).

Summary

A lot of companies have trouble adopting containers because they attempt to shoehorn their existing VM practices into containers. It is best to spend some time to rethink all the advantages that containers have and understand how you can create your process from scratch with that new knowledge.

In this guide, I have presented several bad practices with container usage and also the solution to each one.

1. Attempting to use VM practices on containers. Solution: understand what containers are.
2. Creating Docker files that are not transparent. Solution: write Dockerfiles from scratch instead of adopting existing scripts.
3. Creating Dockerfiles that have side effects. Solution: move side effects to your CI/CD solution and keep Dockerfiles side-effect free.
4. Confusing images used for deployment with those used for development. Solution: don't ship development tools and test frameworks into production servers.
5. Building different images per environment. Solution: build an image only once and promote it across various environments
6. Pulling code from git intro production servers and building images on the fly. Solution: use a Docker registry
7. Promoting git hashes between teams. Solution: promote container images between teams
8. Hardcoding secrets into container images. Solution: build an image only once and use runtime configuration injection
9. Using Docker as CI/CD. Solution: use Docker as a deployment artifact and choose a CI/CD solution for CI/CD
10. Assuming that containers are a dumb packaging method. Solution: Create Dockerfiles that compile/package source code on their own from scratch.

Look at your workflows, ask developers (if you are an operator) or operators (if you are a developer) and try to find if your company falls into one or more of these bad practices.

