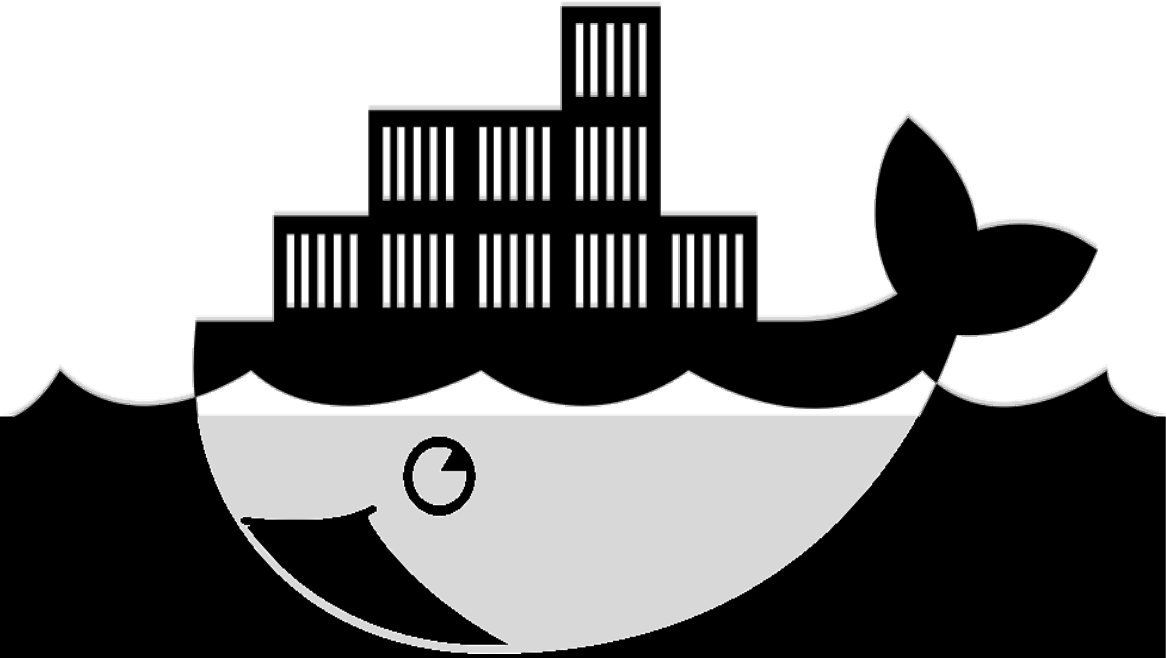
Not all Dockerfile Smells are the Same

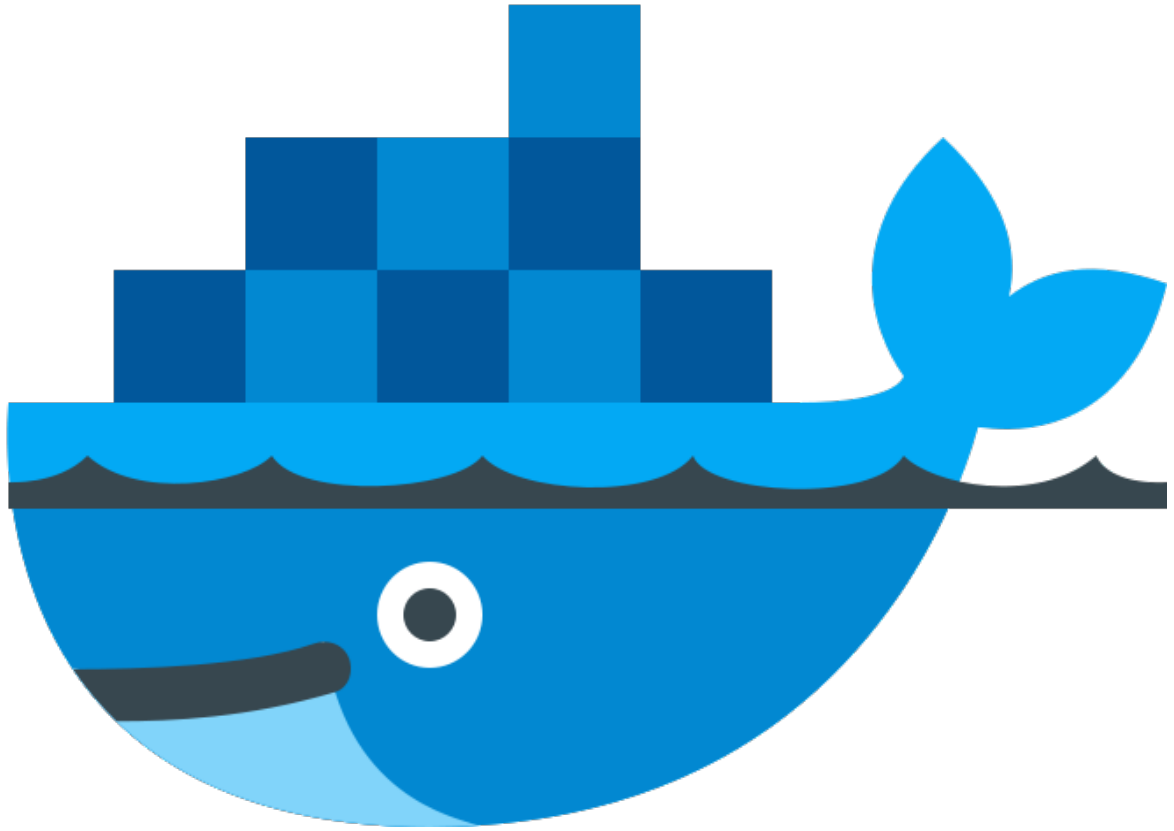
An Empirical Evaluation of Hadolint Writing Practices by **Experts**

Giovanni Rosa **Simone Scalabrino** Gregorio Robles Rocco Oliveto

Università degli Studi del Molise

Universidad Rey Juan Carlos





Docker is the Reference Tool
for Software Containarization

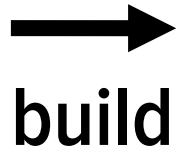
```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

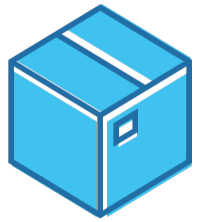
Dockerfile

Docker in a nutshell

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Dockerfile





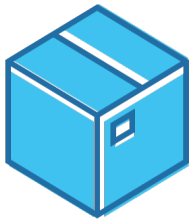
Image

Docker in a nutshell

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

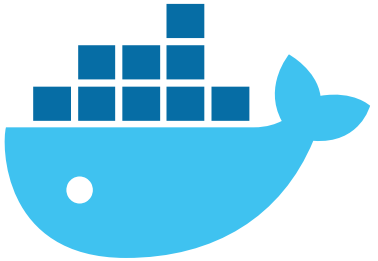
Dockerfile

build



Image

run



Container

Docker in a nutshell

Writing Dockerfiles is challenging

Build Failures

Security Issues

Internal Quality Issues

Build Failures

Security Issues

Internal Quality Issues

Build Failures

Security Issues

~~Internal Quality Issues~~

Dockerfile Smells

HaDoLint

(Haskell **Do**ckerfile **Lint**er)

A Linter for Dockerfiles

A Linter for Dockerfiles

2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

An Empirical Analysis of the Docker Container Ecosystem on GitHub

Cito et al., 2017

Jürgen Cito*, Gerald Schermann*, John Erik Wittern†, Philipp Leitner*, Sali Zumberi*, Harald C. Gall*
* University of Zurich, Switzerland
{lastname}@ifi.uzh.ch
† IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
wittenj@us.ibm.com

***Abstract*—Docker allows packaging an application with its dependencies into a standardized, self-contained unit (a so-called container), which can be used for software development and to run the application on any system. Dockerfiles are declarative definitions of an environment that aim to enable reproducible builds of the container. They can often be found in source code repositories and enable the hosted software to come to life in its execution environment. We conduct an exploratory empirical study with the goal of characterizing the Docker ecosystem, prevalent quality issues, and the evolution of Dockerfiles. We base our study on a data set of over 70000 Dockerfiles, and contrast this general population with samplings that contain the Top-100 and Top-1000 most popular Docker-using projects. We find that most quality issues (28.6%) arise from missing version pinning (i.e., specifying a concrete version for dependencies). Further, we were not able to build 34% of Dockerfiles from a representative sample of 560 projects. Integrating quality checks, e.g., to issue version pinning warnings, into the container build process could result into more reproducible builds. The most popular projects change more often than the rest of the Docker population, with 5.81 revisions per year and 5 lines of code changed on average. Most changes deal with dependencies, that are currently stored in a rather unstructured manner. We propose to introduce an abstraction that, for instance, could deal with the intricacies of different package managers and could improve migration to more light-weight images.**

***Keywords*-empirical software engineering; GitHub; Docker**

I. Introduction

Containerization has recently gained interest as a light-weight virtualization technology to define software infrastructure. Containers allow to package an application with its dependencies and execution environment into a standardized, self-contained unit, which can be used for software development and to run the application on any system. Due to their rapid spread in the software development community, Docker containers have become the de-facto standard format [1]. The contents of a Docker container are declaratively defined in a *Dockerfile* that stores instructions to reach a certain infrastructure state [2], following the notion of Infrastructure-as-Code (IaC) [3]. Source code repositories containing Dockerfiles, thus, potentially enable the execution of program code in an isolated and fast environment with one command. Since its inception in 2013, repositories on GitHub have added 70197 Dockerfiles to their projects (until October 2016).

Given the fast rise in popularity, its ubiquitous nature in industry, and its surrounding claim of enabling reproducibility [4], we study the Docker ecosystem with respect to quality of Dockerfiles and their change and evolution behavior within software repositories. We developed a tool chain that transforms Dockerfiles and their evolution in Git repositories into a relational database model. We mined the entire population of Dockerfiles on GitHub as of October 2016, and summarize our findings on the ecosystem in general, quality aspects, and evolution behavior. The results of our study can inform standard bodies around containers and tool developers to develop better support to improve quality and drive ecosystem change.

We make the following contributions through our exploratory study:

Ecosystem Overview. We characterize the ecosystem of Docker containers on GitHub by analyzing the distribution of projects using Docker, broken down by primary programming language, project size, and the base infrastructure (*base image*) they inherit from. We learn, among other things, that most inherited base images are well-established, but heavy-weight operating systems, while light-weight alternatives are in the minority. However, this defeats the purpose of containers to lower the footprint of virtualization. We envision a recommendation system that analyzes Dockerfiles and transforms its dependency sources to work with light-weight base images.

Quality Assessment. We assess the quality of Dockerfiles on GitHub by classifying results of a Dockerfile Linter [5]. Most of the issues we encountered considered version pinning (i.e., specifying a concrete version for either base images or dependencies), accounting for 28.6% of quality issues. We also built the Dockerfiles for a representative sample of 560 repositories. 66% of Dockerfiles could be built successfully with an average build time of 145.9 seconds. Integrating quality checks into the "docker build" process to warn developers early about build-breaking issues, such as version pinning, can lead to more reproducible builds.

Evolution Behavior. We classify different kinds of changes between consecutive versions of Dockerfiles to characterize their evolution within a repository. On average, Dockerfiles only changed 3.11 times per year, with a mean 3.98 lines of code changed per revision. However, more popular projects revise up to 5.81 per year with 5 lines changed. Dependencies see a high rate of change over time, reinforcing our findings to improve dependency handling from the analysis of the

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.67

84%

Dockerfiles affected by smells

Do developers care?

Do developers care?

Let’s ask the experts!

Preliminary Study

What smells can be found in Dockerfiles made by experts?

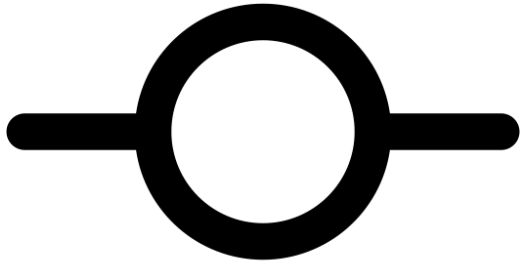
RQ1

39 Official Docker Repos

Experimental Procedure

RQ1

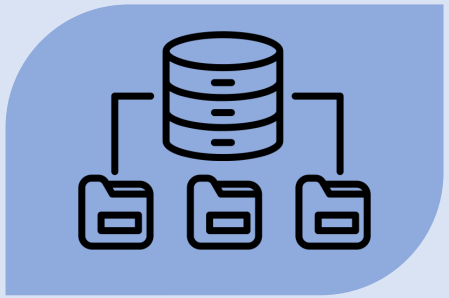
Docker Repos



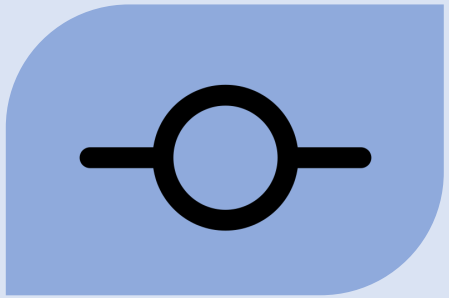
37k Git Commits

Experimental Procedure

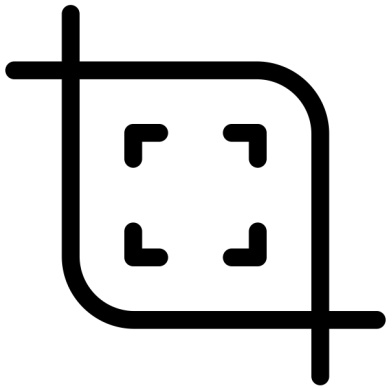
RQ1



Docker Repos



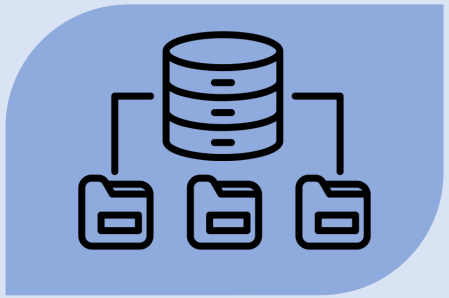
Repo History



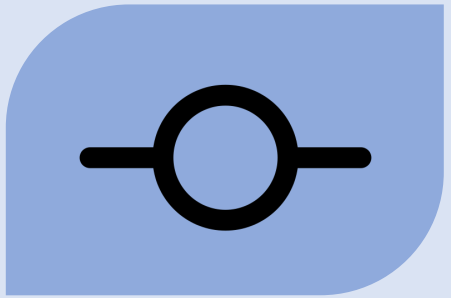
724 Dockerfile Snapshots

Experimental Procedure

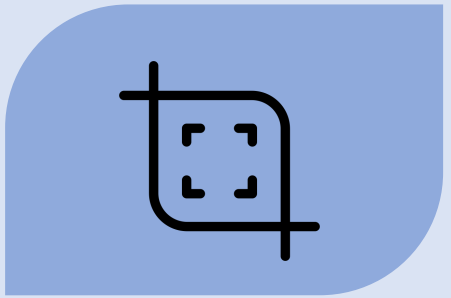
RQ1



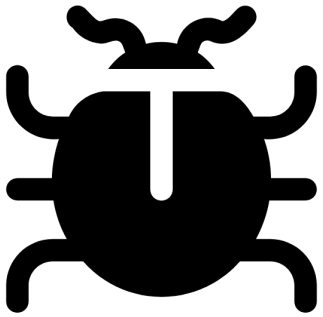
Docker Repos



Repo History



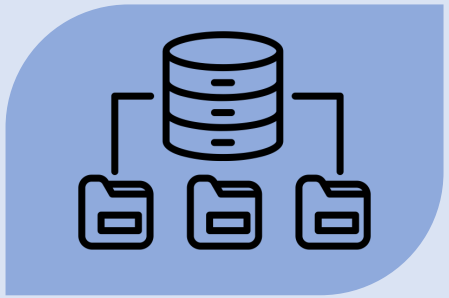
Dockerfiles



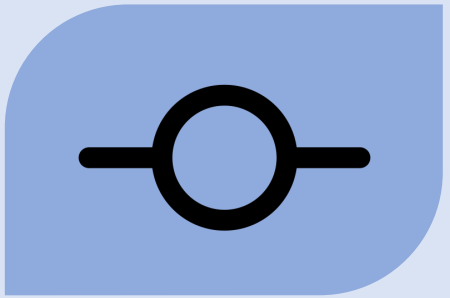
Unique Smell Occurrences

Experimental Procedure

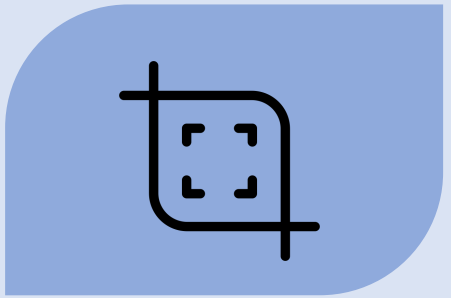
RQ1



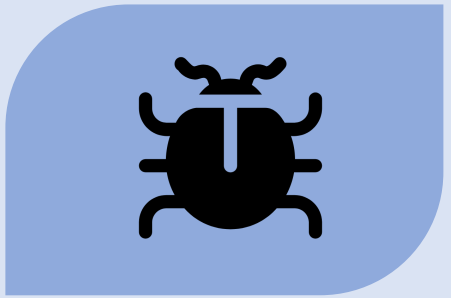
Docker Repos



Repo History



Dockerfiles



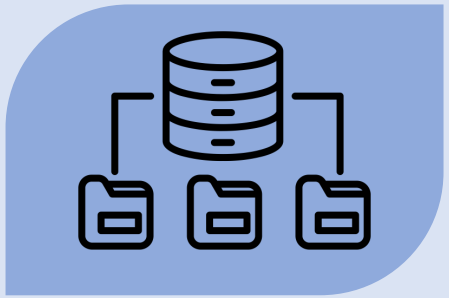
Smells

6k

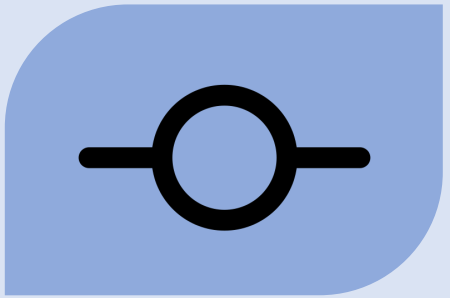
Smells Affecting
Official Dockerfiles

Results

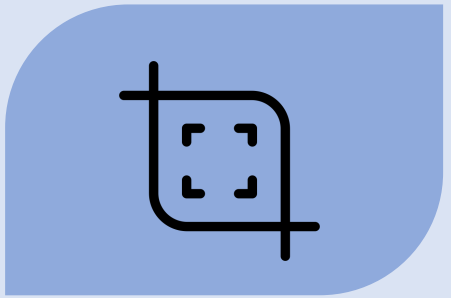
RQ1



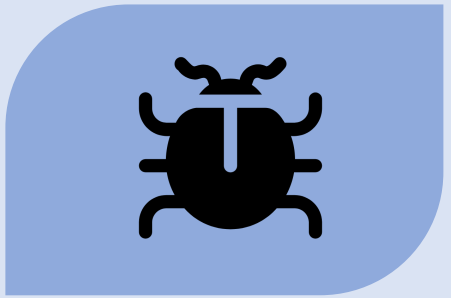
Docker Repos



Repo History



Dockerfiles



Smells

6k

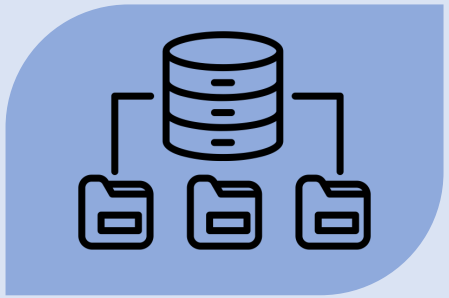
Smells Affecting
Official Dockerfiles

Use of WORKDIR

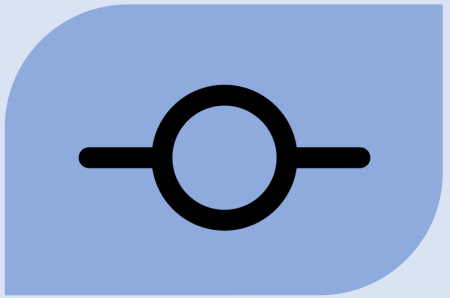
3

Results

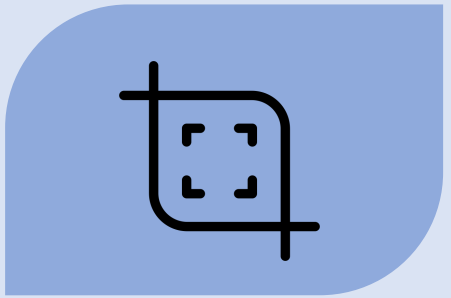
RQ1



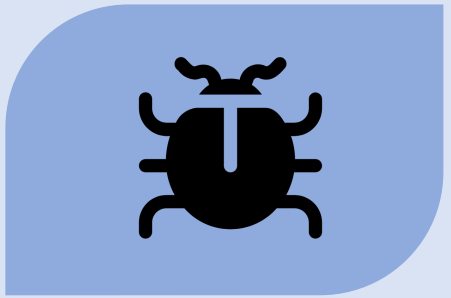
Docker Repos



Repo History



Dockerfiles



Smells

6k

Smells Affecting
Official Dockerfiles

Use of WORKDIR

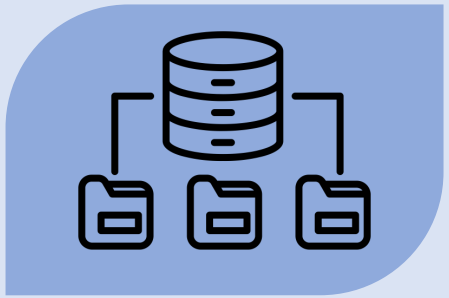
3

Version Pinning

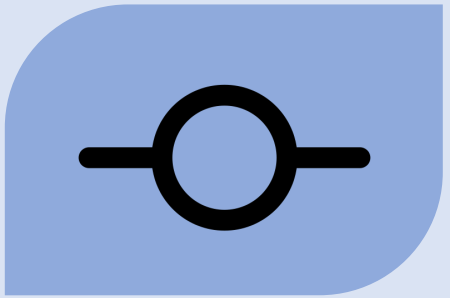
2

Results

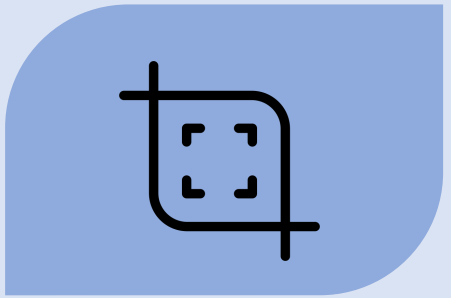
RQ1



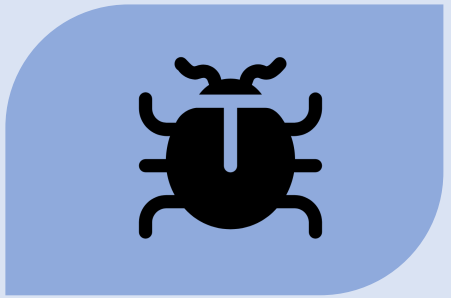
Docker Repos



Repo History



Dockerfiles



Smells

6k

Smells Affecting
Official Dockerfiles

Use of WORKDIR — 3

Version Pinning — 2

Shell Pipefail — 1

Results

Asking the Experts

Are the Dockerfile Smells detected by Hadolint actual Bad Practices?

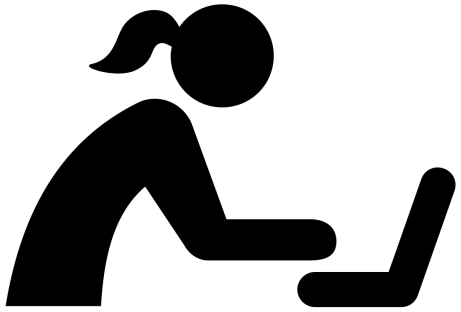
RQ2

39 Official Docker Repos

Experimental Procedure

RQ2

Docker Repos



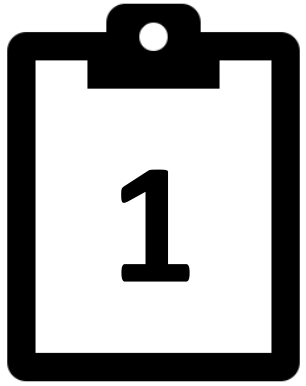
1k Expert Developers

Experimental Procedure

RQ2

Docker Repos

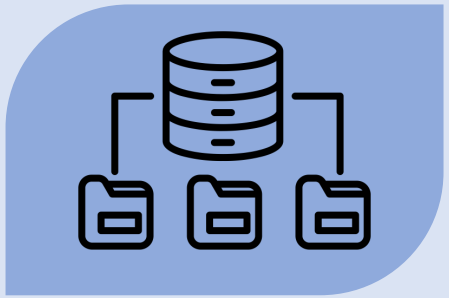
Contributors



Demographic Information

Experimental Procedure

RQ2



Docker Repos

Contributors

Pre-Survey

Smell Identification Tasks

Experimental Procedure

RQ2

Docker Repos

Contributors

Pre-Survey

Identification

1

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Clean Dockerfiles

Experimental Procedure

RQ2

Docker Repos

Contributors

Pre-Survey

Identification

1

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Clean Dockerfiles

2

```
1 FROM node:~~12-alpine~~
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Smell Injection

Experimental Procedure

RQ2

Docker Repos

Contributors

Pre-Survey

Identification

1

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Clean Dockerfiles

2

```
1 FROM node:~~12-alpine~~
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Smell Injection

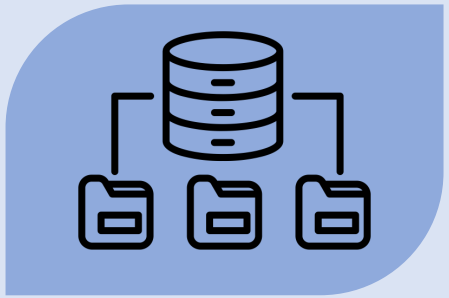
3

```
1 FROM node:~~12-alpine~~
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
```

Identification

Experimental Procedure

RQ2



Docker Repos

Contributors

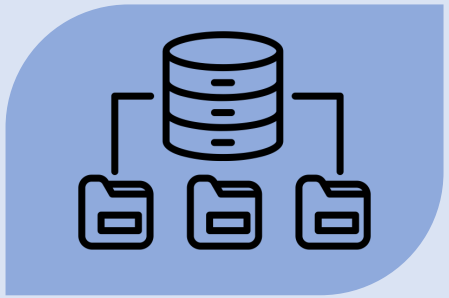
Pre-Survey

Identification

37
Participants

Results

RQ2



Docker Repos

Contributors

Pre-Survey

Identification

37

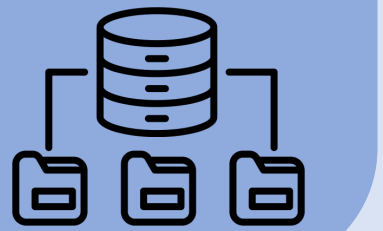
Participants

64%

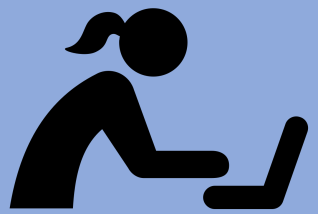
Hadolint Smells **not** Identified

Results

RQ2



Docker Repos



Contributors

Pre-Survey

Identification

37

Participants

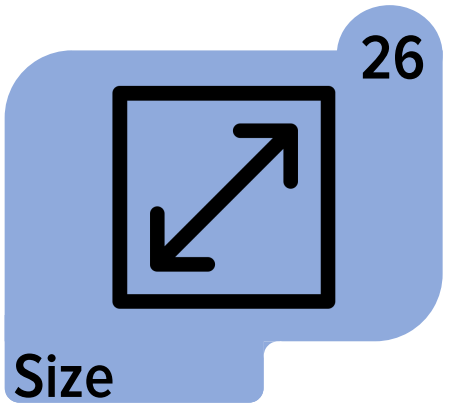
64%

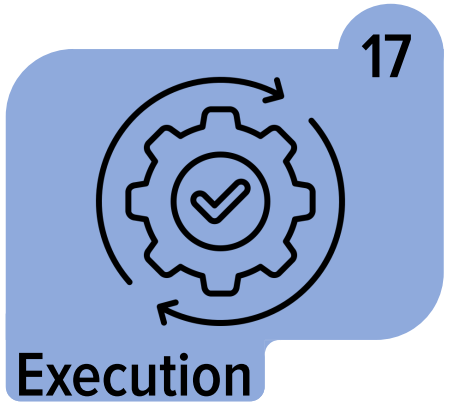
Hadolint Smells **not** Identified

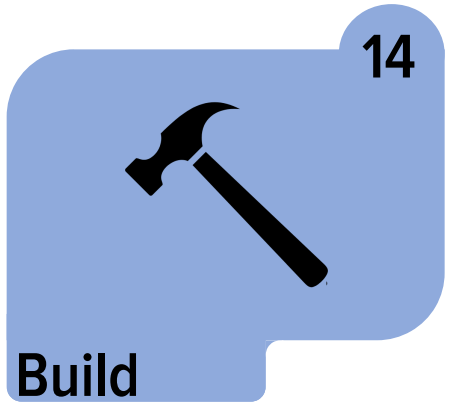
26

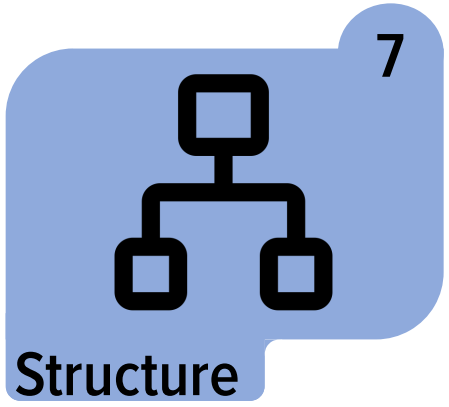
Unexpected Smells

Results









Taxonomy of New Recommendations

Binnacle

DRIVE

Dockercleaner

Mapping with Existing Catalogs

Binnacle

DRIVE

Dockercleaner

20

Not Mapped

Mapping with Existing Catalogs

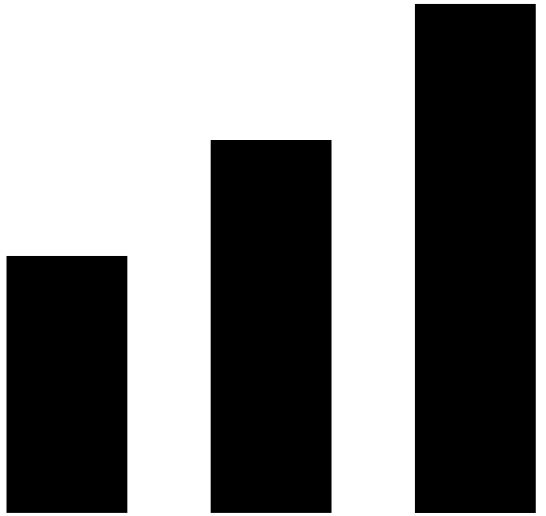
Hadolint

Binnacle

DRIVE

Dockercleaner

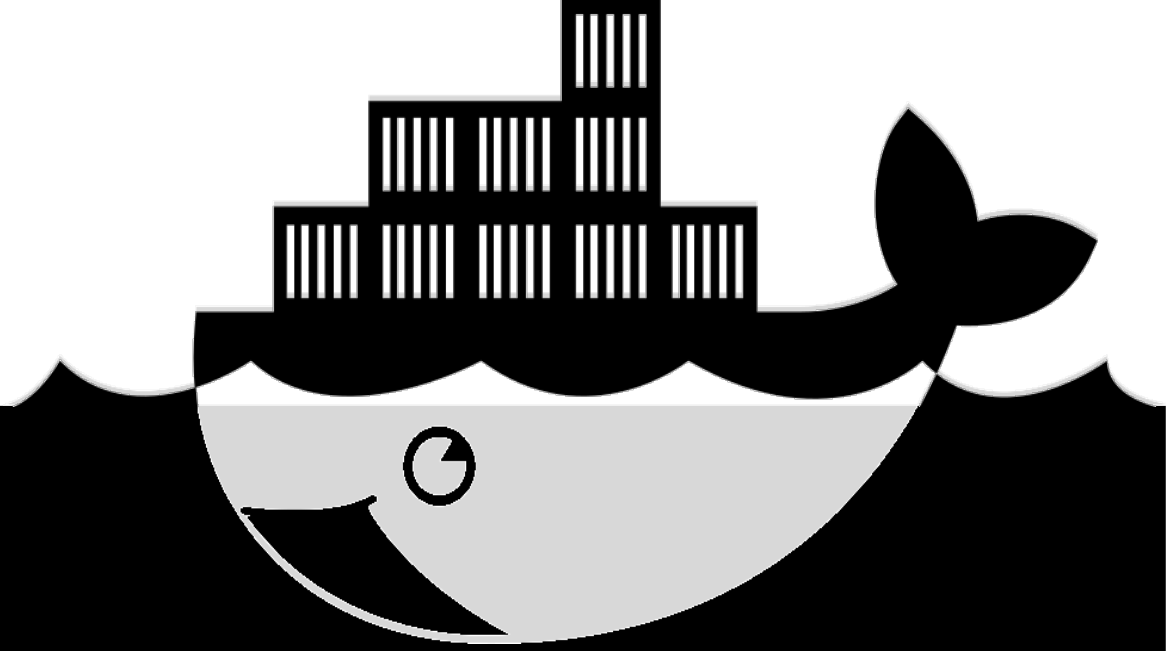
New smells



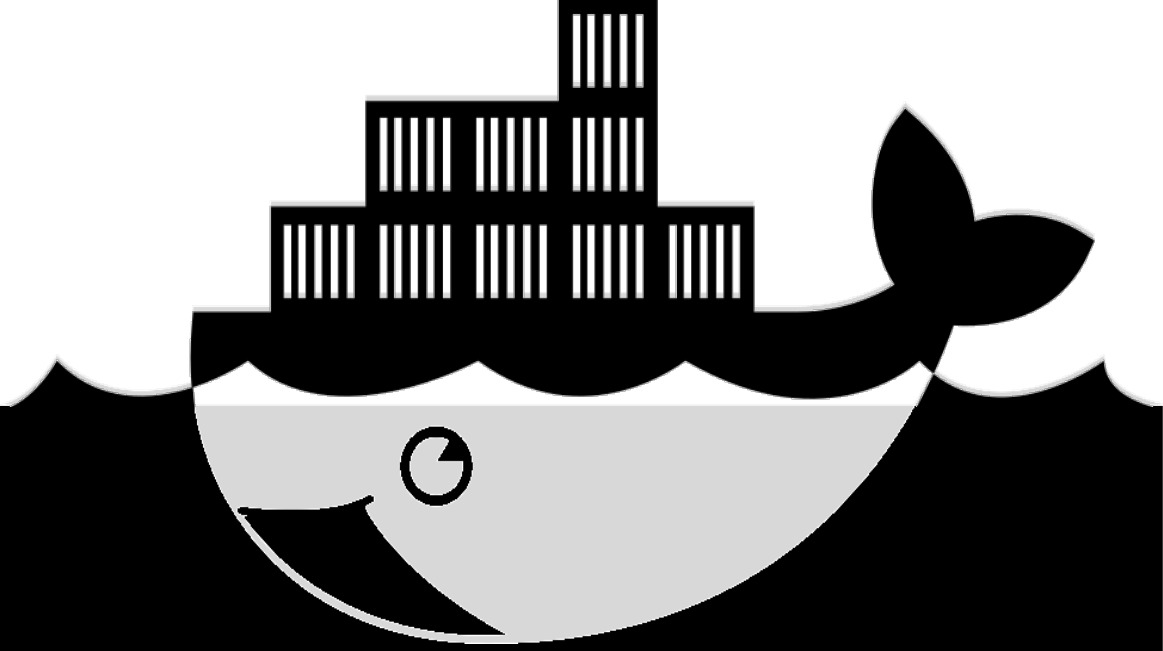
Ranking of
Dockerfile Smells

Mapping with Existing Catalogs

Wrapping up

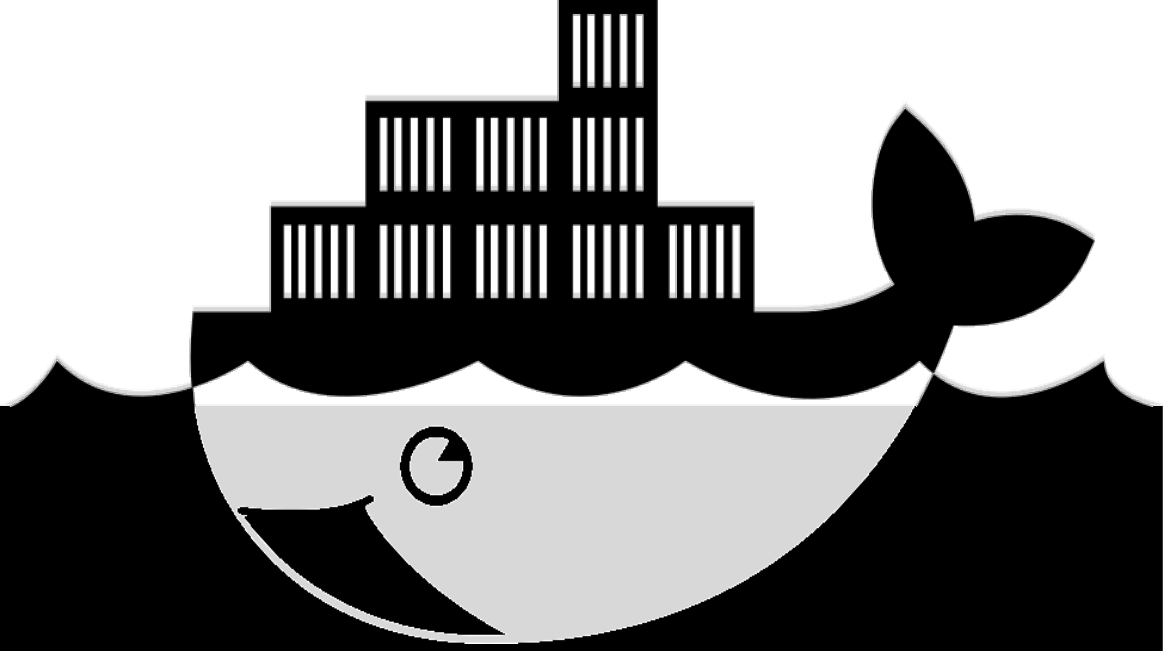


Wrapping up



1. Only a few Hadolint smells are important to experts

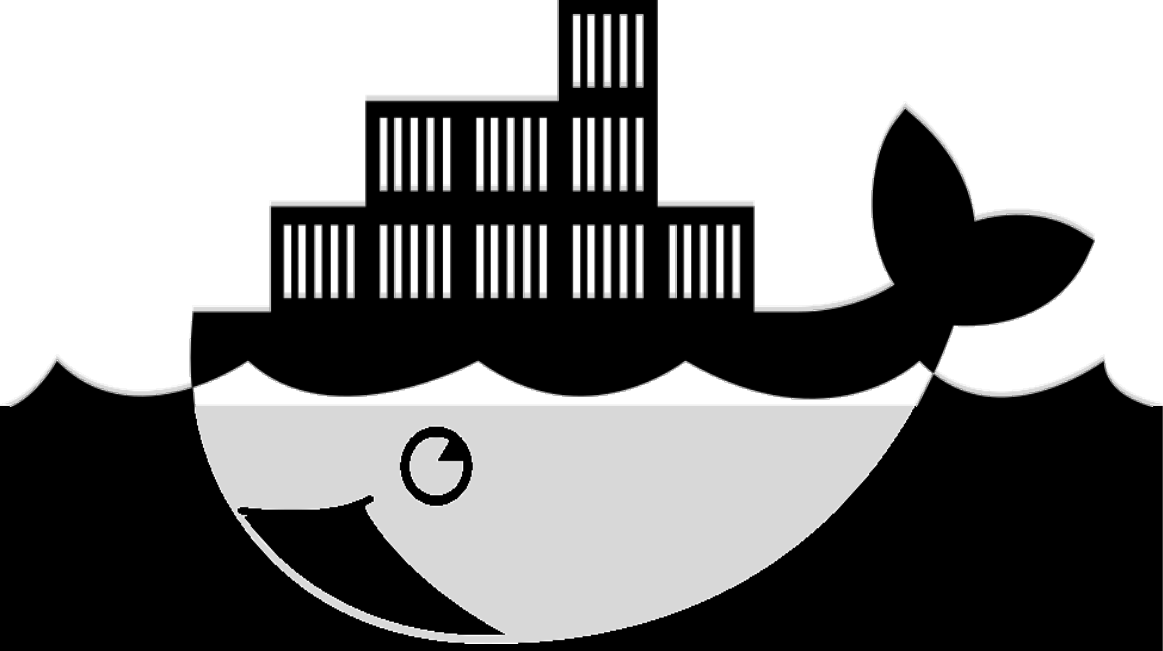
Wrapping up



1. Only a few Hadolint smells are important to experts
2. Experts focus on performance and security

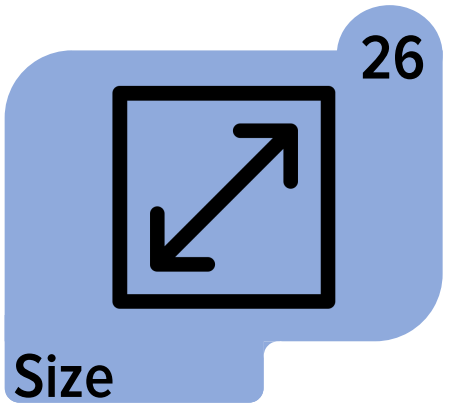
Not all Dockerfile Smells are the Same

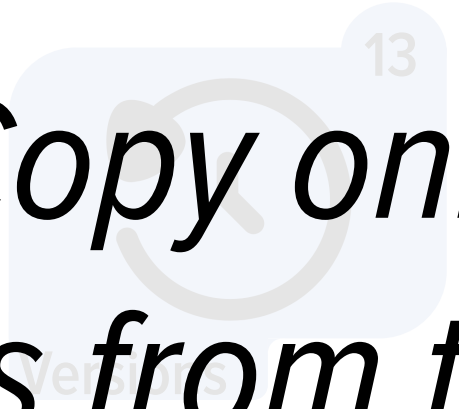
An Empirical Evaluation of Hadolint Writing Practices by Experts



Giovanni Rosa Simone Scalabrino Gregorio Robles Rocco Oliveto

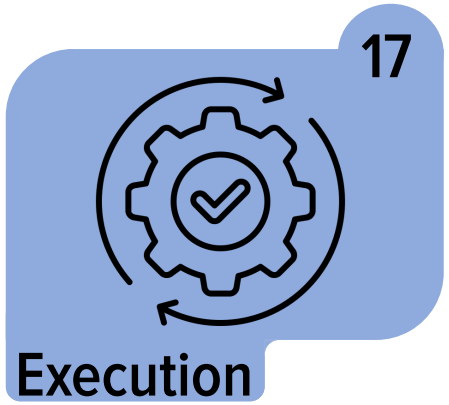
1. Only a few Hadolint smells are important to experts
2. Experts focus on performance and security





“Copy only the necessary files from the build context”

Taxonomy of New Recommendations

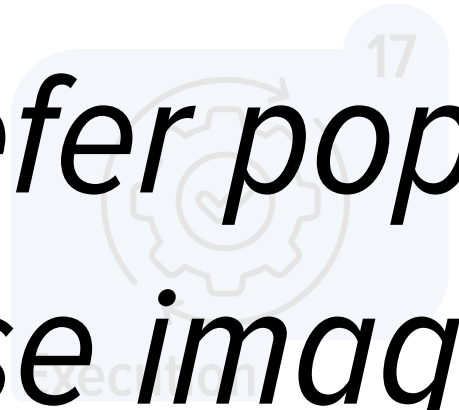


“Prefer a binary executable for ENTRYPOINT”

Taxonomy of New Recommendations

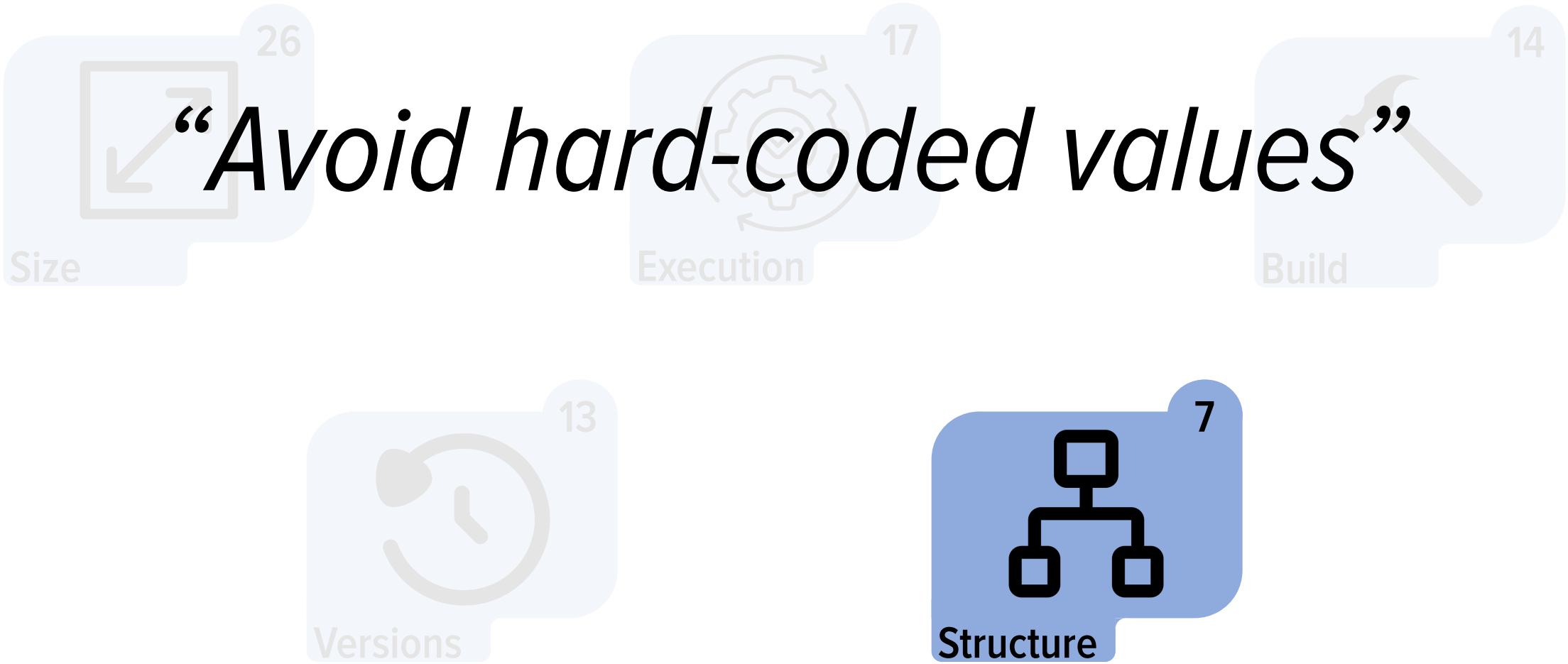
“Copy dependencies before sources”

Taxonomy of New Recommendations



“Prefer popular base images”

Taxonomy of New Recommendations



“Avoid hard-coded values”

Taxonomy of New Recommendations