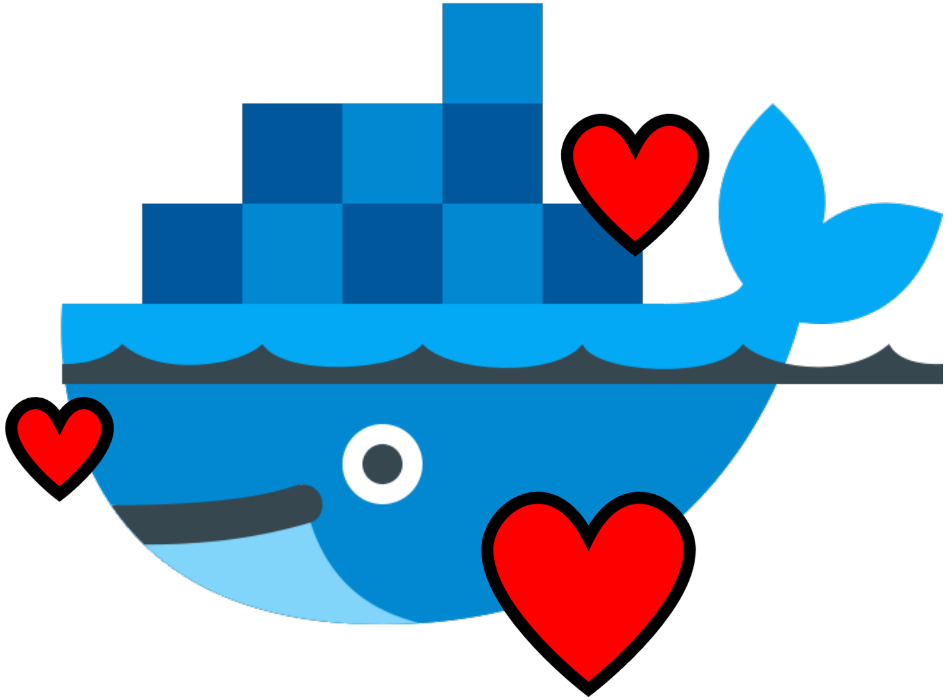
Assessing and Improving the Quality of Docker Artifacts

Giovanni Rosa
University of Molise, Italy

Advisors: Rocco Oliveto and Simone Scalabrino

ICSME ’22 Doctoral Symposium - Oct 4th 2022
Limassol, Cyprus



#1 Most-Wanted

and

#1 Most Loved

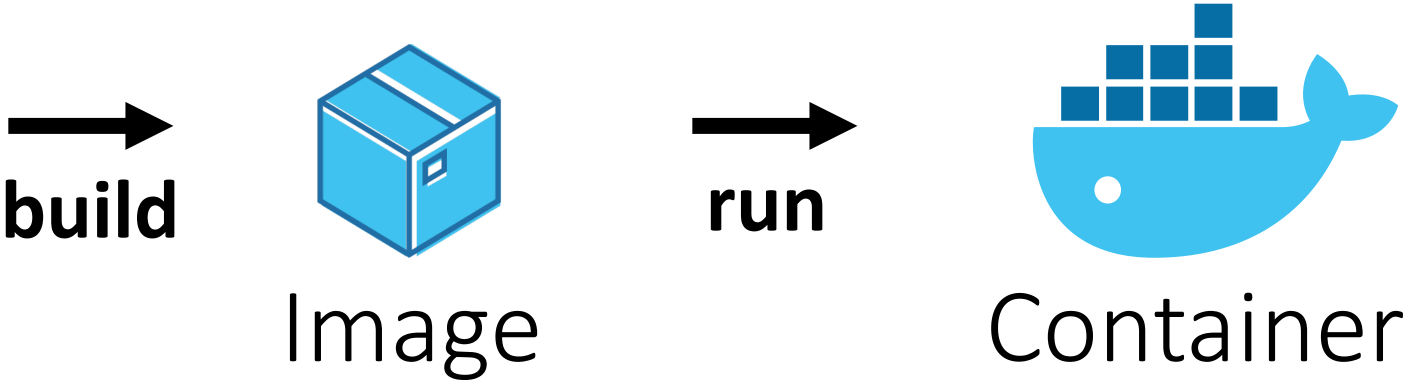
tool

2022
Developer
Survey

Why Docker?

```
FROM node:12-alpine

RUN apk add --no-cache python2 g++ make

WORKDIR /app
COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000 here
```

Dockerfile



build

Image

run

Container

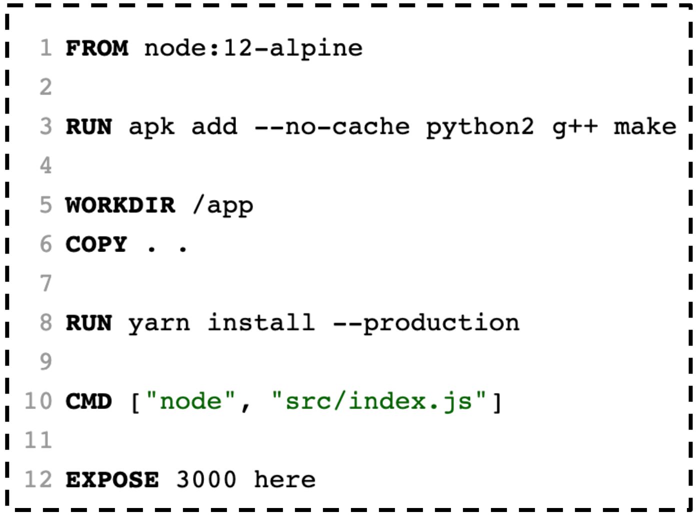
Docker in a nutshell

```
FROM node:12-alpine

RUN apk add --no-cache python2 g++ make

WORKDIR /app
COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000 here
```

Dockerfile → **build** → Image → **run** → Container

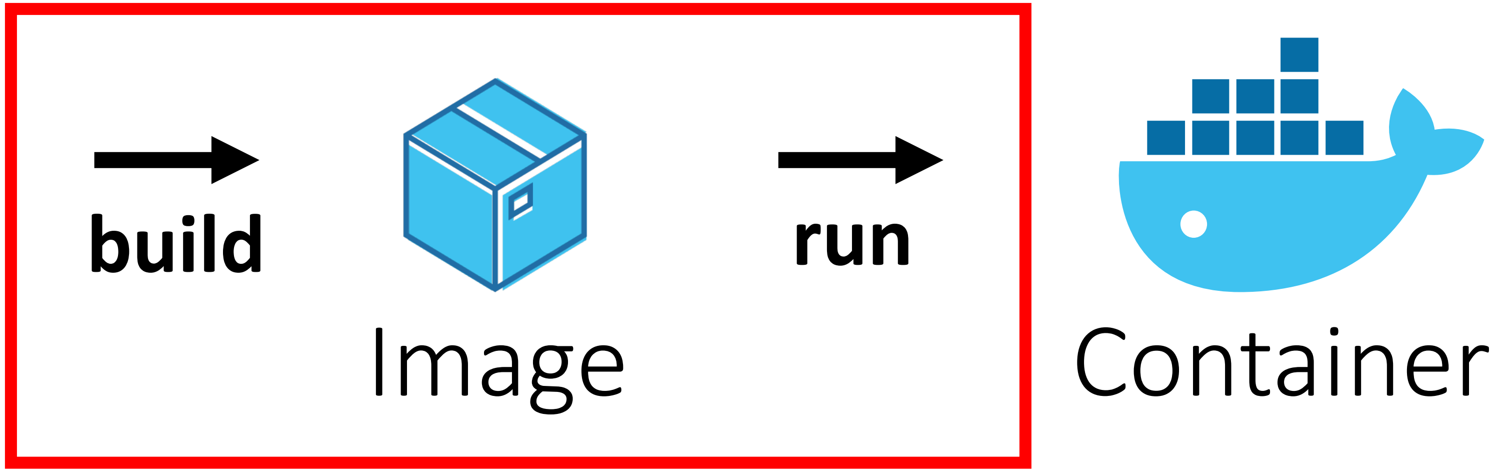
Docker in a nutshell

Docker in a nutshell



```
FROM node:12-alpine

RUN apk add --no-cache python2 g++ make

WORKDIR /app
COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000 here
```

Dockerfile

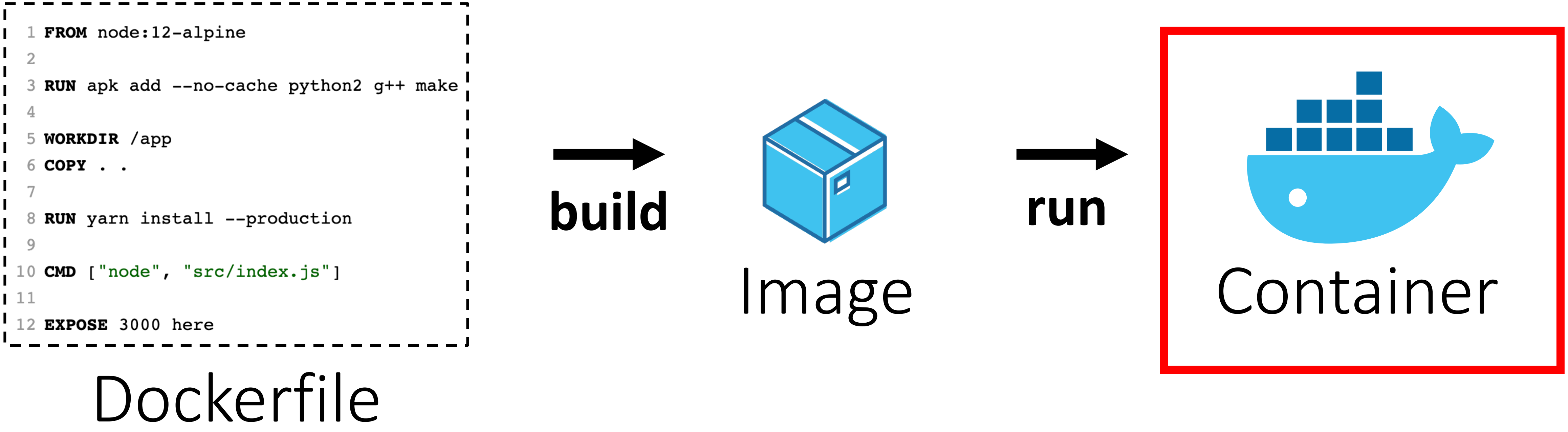


build

Image

run

Container



```
FROM node:12-alpine

RUN apk add --no-cache python2 g++ make

WORKDIR /app
COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

EXPOSE 3000 here
```

Dockerfile

build

Image

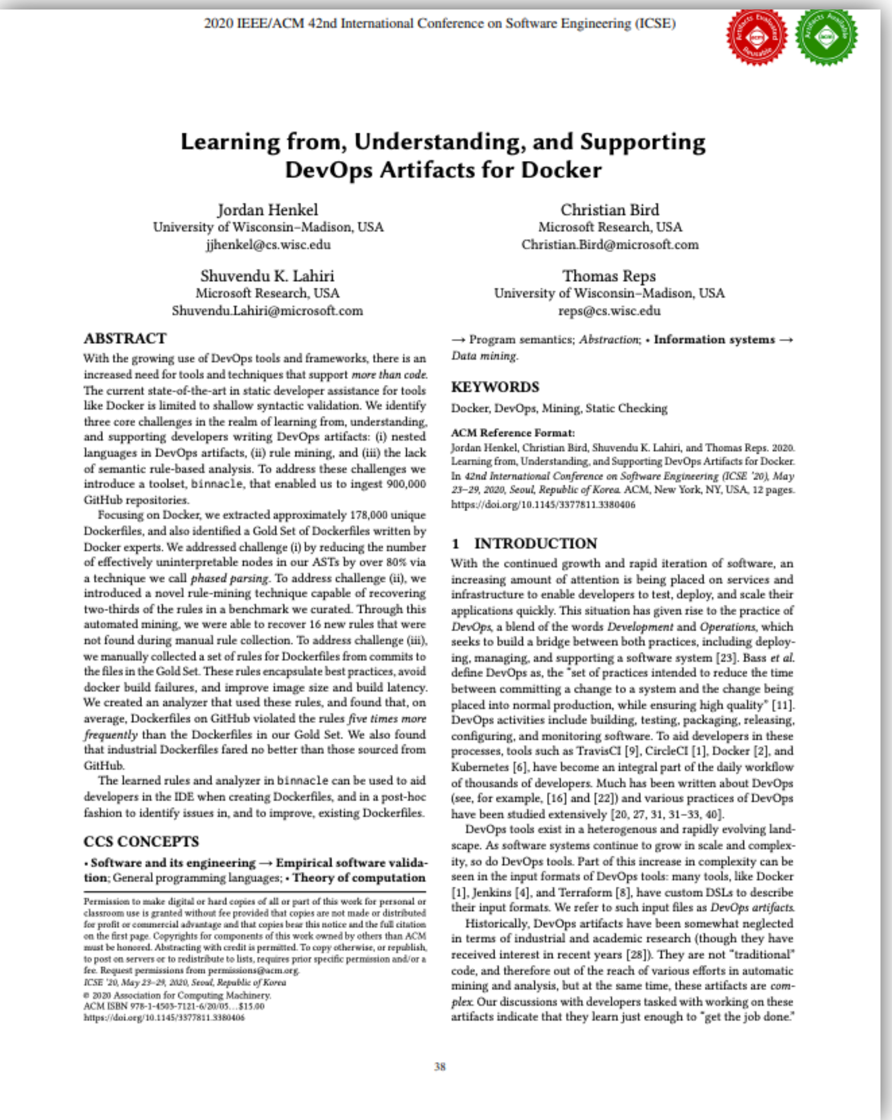
run

Container

Docker in a nutshell

Writing a Dockerfile may seem **simple**

Limited developer assistance



2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)

Learning from, Understanding, and Supporting DevOps Artifacts for Docker

Jordan Henkel
University of Wisconsin–Madison, USA
jjhenkel@cs.wisc.edu

Christian Bird
Microsoft Research, USA
Christian.Bird@microsoft.com

Shuvendu K. Lahiri
Microsoft Research, USA
Shuvendu.Lahiri@microsoft.com

Thomas Reps
University of Wisconsin–Madison, USA
reps@cs.wisc.edu

ABSTRACT

With the growing use of DevOps tools and frameworks, there is an increased need for tools and techniques that support *more than code*. The current state-of-the-art in static developer assistance for tools like Docker is limited to shallow syntactic validation. We identify three core challenges in the realm of learning from, understanding, and supporting developers writing DevOps artifacts: (i) nested languages in DevOps artifacts, (ii) rule mining, and (iii) the lack of semantic rule-based analysis. To address these challenges we introduce a toolset, `binnacle`, that enabled us to ingest 900,000 GitHub repositories.

Focusing on Docker, we extracted approximately 178,000 unique Dockerfiles, and also identified a Gold Set of Dockerfiles written by Docker experts. We addressed challenge (i) by reducing the number of effectively uninterpretable nodes in our ASTs by over 80% via a technique we call *phased parsing*. To address challenge (ii), we introduced a novel rule-mining technique capable of recovering two-thirds of the rules in a benchmark we curated. Through this automated mining, we were able to recover 16 new rules that were not found during manual rule collection. To address challenge (iii), we manually collected a set of rules for Dockerfiles from commits to the files in the Gold Set. These rules encapsulate best practices, avoid docker build failures, and improve image size and build latency. We created an analyzer that used these rules, and found that, on average, Dockerfiles on GitHub violated the rules *five times more frequently* than the Dockerfiles in our Gold Set. We also found that industrial Dockerfiles fared no better than those sourced from GitHub.

The learned rules and analyzer in `binnacle` can be used to aid developers in the IDE when creating Dockerfiles, and in a post-hoc fashion to identify issues in, and to improve, existing Dockerfiles.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; General programming languages; • **Theory of computation** → Program semantics; *Abstraction*; • **Information systems** → *Data mining*.

KEYWORDS

Docker, DevOps, Mining, Static Checking

ACM Reference Format:
Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. 2020. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380406

1 INTRODUCTION

With the continued growth and rapid iteration of software, an increasing amount of attention is being placed on services and infrastructure to enable developers to test, deploy, and scale their applications quickly. This situation has given rise to the practice of *DevOps*, a blend of the words *Development* and *Operations*, which seeks to build a bridge between both practices, including deploying, managing, and supporting a software system [23]. Bass *et al.* define DevOps as, the "set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" [11]. DevOps activities include building, testing, packaging, releasing, configuring, and monitoring software. To aid developers in these processes, tools such as TravisCI [9], CircleCI [1], Docker [2], and Kubernetes [6], have become an integral part of the daily workflow of thousands of developers. Much has been written about DevOps (see, for example, [16] and [22]) and various practices of DevOps have been studied extensively [20, 27, 31, 31–33, 40].

DevOps tools exist in a heterogenous and rapidly evolving landscape. As software systems continue to grow in scale and complexity, so do DevOps tools. Part of this increase in complexity can be seen in the input formats of DevOps tools: many tools, like Docker [1], Jenkins [4], and Terraform [8], have custom DSLs to describe their input formats. We refer to such input files as *DevOps artifacts*.

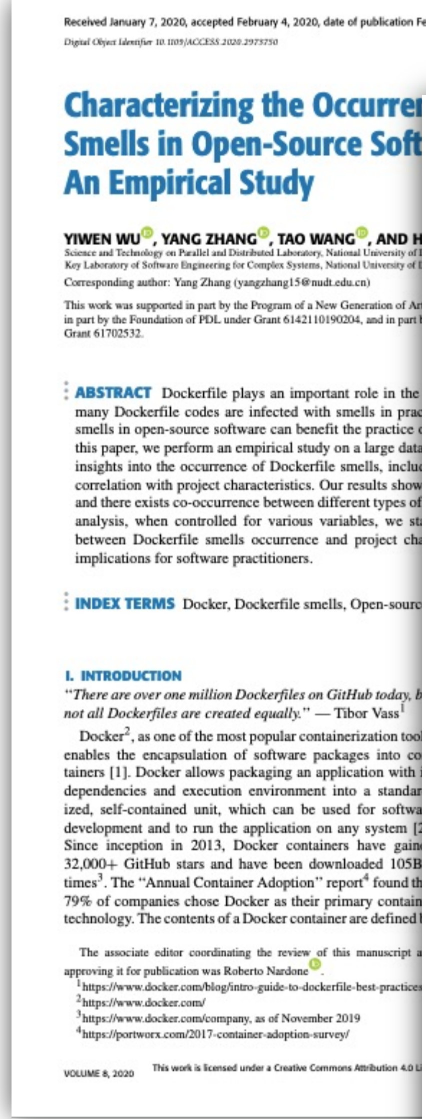
Historically, DevOps artifacts have been somewhat neglected in terms of industrial and academic research (though they have received interest in recent years [28]). They are not "traditional" code, and therefore out of the reach of various efforts in automatic mining and analysis, but at the same time, these artifacts are *complex*. Our discussions with developers tasked with working on these artifacts indicate that they learn just enough to "get the job done."

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380406

Henkel et. al 2020

?



Received January 7, 2020, accepted February 4, 2020, date of publication February 13, 2020, date of current version February 27, 2020.
Digital Object Identifier 10.1109/ACCESS.2020.2973750

Characterizing the Occurrer
Smells in Open-Source Soft
An Empirical Study

YIWEN WU, YANG ZHANG, TAO WANG, AND H
Science and Technology on Parallel and Distributed Laboratory, National University of
Key Laboratory of Software Engineering for Complex Systems, National University of
Corresponding author: Yang Zhang (yangzhang15@nudt.edu.cn)

This work was supported in part by the Program of a New Generation of Ar
in part by the Foundation of PDL under Grant 6142110190204, and in part
Grant 61702532.

ABSTRACT Dockerfile plays an important role in the
many Dockerfile codes are infected with smells in prac
smells in open-source software can benefit the practice
this paper, we perform an empirical study on a large data
insights into the occurrence of Dockerfile smells, inclu
correlation with project characteristics. Our results show
and there exists co-occurrence between different types of
analysis, when controlled for various variables, we st
between Dockerfile smells occurrence and project ch
implications for software practitioners.

INDEX TERMS Docker, Dockerfile smells, Open-sourc

I. INTRODUCTION

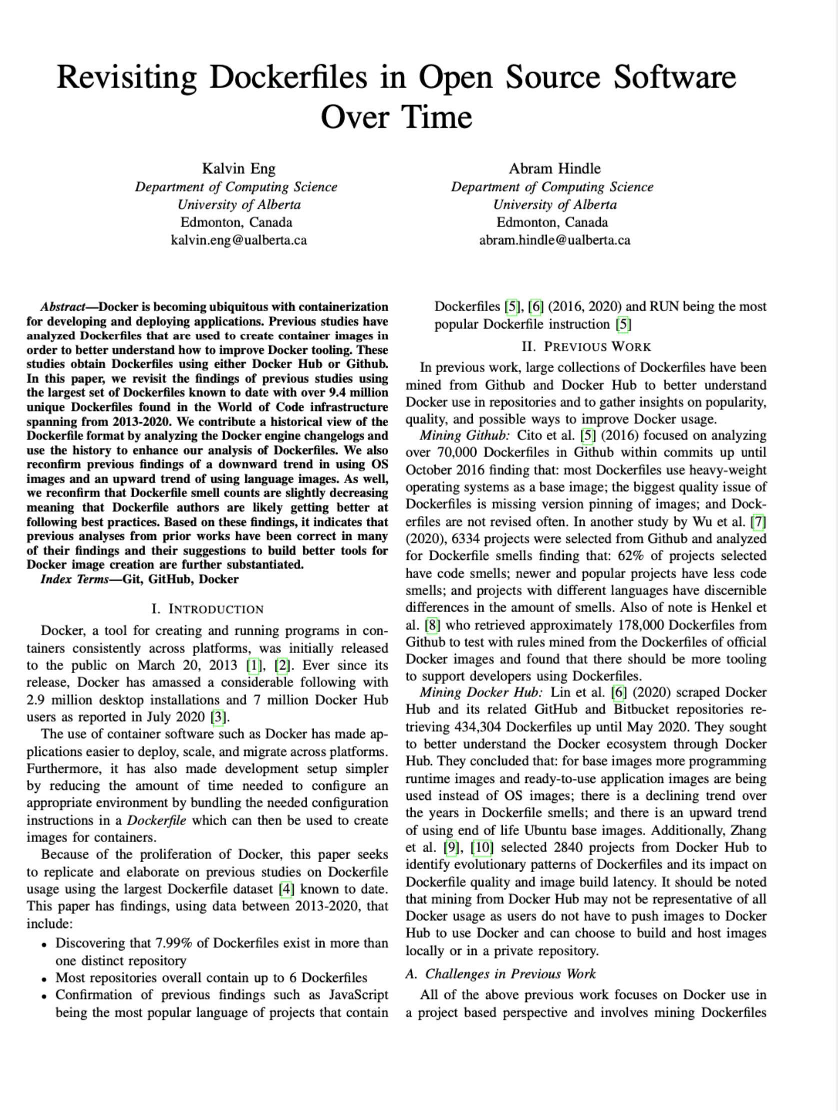
"There are over one million Dockerfiles on GitHub today,
not all Dockerfiles are created equally." — Tibor Vass[1]

Docker[2], as one of the most popular containerization too
enables the encapsulation of software packages into co
tainers [1]. Docker allows packaging an application with
dependencies and execution environment into a standar
ized, self-contained unit, which can be used for softwa
development and to run the application on any system
Since inception in 2013, Docker containers have gain
32,000+ GitHub stars and have been downloaded 105B
times[3]. The "Annual Container Adoption" report[4] found th
79% of companies chose Docker as their primary contain
technology. The contents of a Docker container are defined

The associate editor coordinating the review of this manuscript
approving it for publication was Roberto Nardone.

[1] https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices
[2] https://www.docker.com/
[3] https://www.docker.com/company, as of November 2019
[4] https://portworx.com/2017-container-adoption-survey/

Wu et. al 2020



Revisiting Dockerfiles in Open Source Software Over Time

Kalvin Eng
Department of Computing Science
University of Alberta
Edmonton, Canada
kalvin.eng@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
abram.hindle@ualberta.ca

***Abstract*—Docker is becoming ubiquitous with containerization for developing and deploying applications. Previous studies have analyzed Dockerfiles that are used to create container images in order to better understand how to improve Docker tooling. These studies obtain Dockerfiles using either Docker Hub or Github. In this paper, we revisit the findings of previous studies using the largest set of Dockerfiles known to date with over 9.4 million unique Dockerfiles found in the World of Code infrastructure spanning from 2013-2020. We contribute a historical view of the Dockerfile format by analyzing the Docker engine changelogs and use the history to enhance our analysis of Dockerfiles. We also reconfirm previous findings of a downward trend in using OS images and an upward trend of using language images. As well, we reconfirm that Dockerfile smell counts are slightly decreasing meaning that Dockerfile authors are likely getting better at following best practices. Based on these findings, it indicates that previous analyses from prior works have been correct in many of their findings and their suggestions to build better tools for Docker image creation are further substantiated.**

***Index Terms*—Git, GitHub, Docker**

I. INTRODUCTION

Docker, a tool for creating and running programs in containers consistently across platforms, was initially released to the public on March 20, 2013 [1], [2]. Ever since its release, Docker has amassed a considerable following with 2.9 million desktop installations and 7 million Docker Hub users as reported in July 2020 [3].

The use of container software such as Docker has made applications easier to deploy, scale, and migrate across platforms. Furthermore, it has also made development setup simpler by reducing the amount of time needed to configure an appropriate environment by bundling the needed configuration instructions in a *Dockerfile* which can then be used to create images for containers.

Because of the proliferation of Docker, this paper seeks to replicate and elaborate on previous studies on Dockerfile usage using the largest Dockerfile dataset [4] known to date. This paper has findings, using data between 2013-2020, that include:

- Discovering that 7.99% of Dockerfiles exist in more than one distinct repository
- Most repositories overall contain up to 6 Dockerfiles
- Confirmation of previous findings such as JavaScript being the most popular language of projects that contain Dockerfiles [5], [6] (2016, 2020) and RUN being the most popular Dockerfile instruction [5]

II. PREVIOUS WORK

In previous work, large collections of Dockerfiles have been mined from Github and Docker Hub to better understand Docker use in repositories and to gather insights on popularity, quality, and possible ways to improve Docker usage.

Mining Github: Cito et al. [5] (2016) focused on analyzing over 70,000 Dockerfiles in Github within commits up until October 2016 finding that: most Dockerfiles use heavy-weight operating systems as a base image; the biggest quality issue of Dockerfiles is missing version pinning of images; and Dockerfiles are not revised often. In another study by Wu et al. [7] (2020), 6334 projects were selected from Github and analyzed for Dockerfile smells finding that: 62% of projects selected have code smells; newer and popular projects have less code smells; and projects with different languages have discernible differences in the amount of smells. Also of note is Henkel et al. [8] who retrieved approximately 178,000 Dockerfiles from Github to test with rules mined from the Dockerfiles of official Docker images and found that there should be more tooling to support developers using Dockerfiles.

Mining Docker Hub: Lin et al. [6] (2020) scraped Docker Hub and its related GitHub and Bitbucket repositories retrieving 434,304 Dockerfiles up until May 2020. They sought to better understand the Docker ecosystem through Docker Hub. They concluded that: for base images more programming runtime images and ready-to-use application images are being used instead of OS images; there is a declining trend over the years in Dockerfile smells; and there is an upward trend of using end of life Ubuntu base images. Additionally, Zhang et al. [9], [10] selected 2840 projects from Docker Hub to identify evolutionary patterns of Dockerfiles and its impact on Dockerfile quality and image build latency. It should be noted that mining from Docker Hub may not be representative of all Docker usage as users do not have to push images to Docker Hub to use Docker and can choose to build and host images locally or in a private repository.

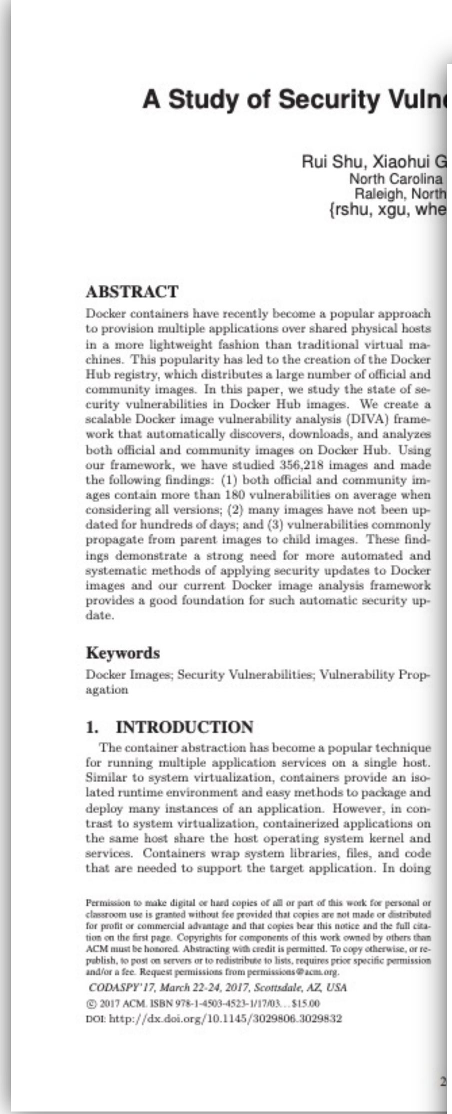
A. Challenges in Previous Work

All of the above previous work focuses on Docker use in a project based perspective and involves mining Dockerfiles

Eng et. al 2021

Dockerfile smells

What about quality?



A Study of Security Vulne

Rui Shu, Xiaohui G
North Carolina
Raleigh, North
{rshu, xgu, whe

ABSTRACT

Docker containers have recently become a popular approach to provision multiple applications over shared physical hosts in a more lightweight fashion than traditional virtual machines. This popularity has led to the creation of the Docker Hub registry, which distributes a large number of official and community images. In this paper, we study the state of security vulnerabilities in Docker Hub images. We create a scalable Docker image vulnerability analysis (DIVA) framework that automatically discovers, downloads, and analyzes both official and community images on Docker Hub. Using our framework, we have studied 356,218 images and made the following findings: (1) both official and community images contain more than 180 vulnerabilities on average when considering all versions; (2) many images have not been updated for hundreds of days; and (3) vulnerabilities commonly propagate from parent images to child images. These findings demonstrate a strong need for more automated and systematic methods of applying security updates to Docker images and our current Docker image analysis framework provides a good foundation for such automatic security update.

Keywords

Docker Images; Security Vulnerabilities; Vulnerability Propagation

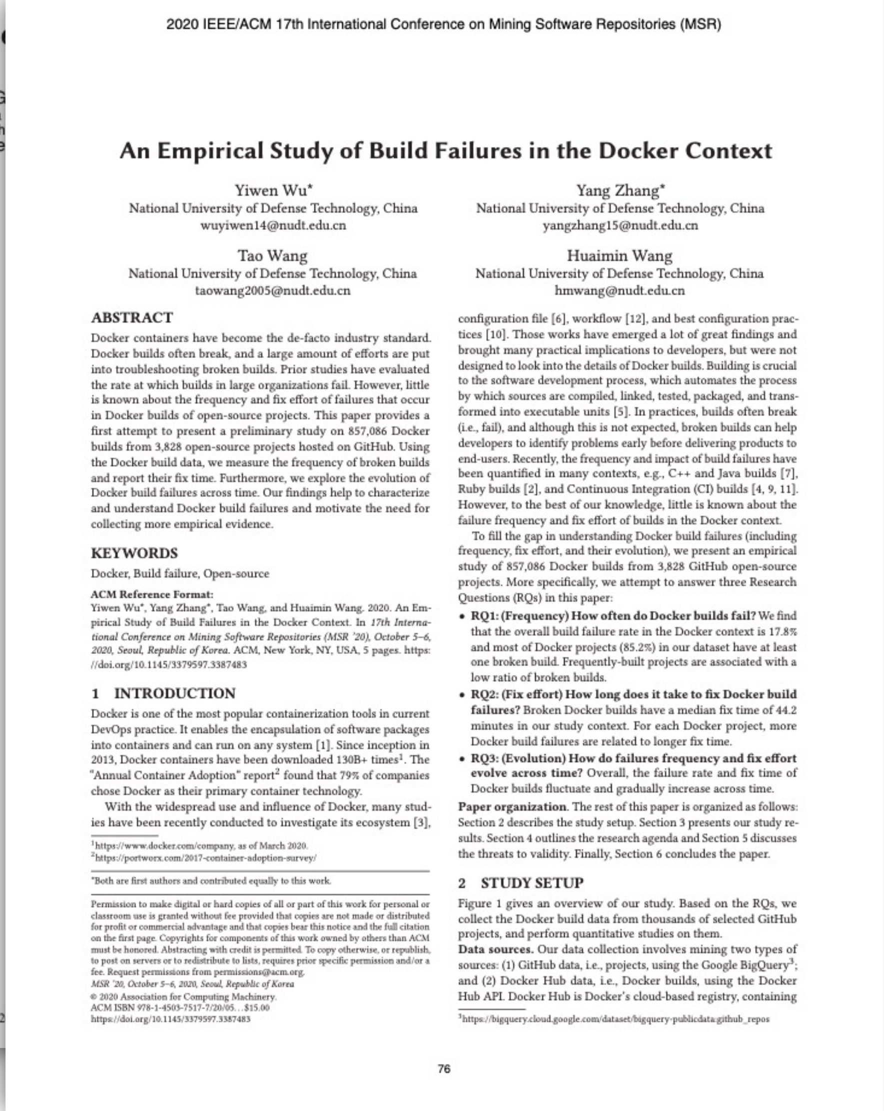
1. INTRODUCTION

The container abstraction has become a popular technique for running multiple application services on a single host. Similar to system virtualization, containers provide an isolated runtime environment and easy methods to package and deploy many instances of an application. However, in contrast to system virtualization, containerized applications on the same host share the host operating system kernel and services. Containers wrap system libraries, files, and code that are needed to support the target application. In doing

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA

DOI: http://dx.doi.org/10.1145/3029806.3029832

Shu et. al 2017



2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)

An Empirical Study of Build Failures in the Docker Context

Yiwen Wu*
National University of Defense Technology, China
wuyiwen14@nudt.edu.cn

Yang Zhang*
National University of Defense Technology, China
yangzhang15@nudt.edu.cn

Tao Wang
National University of Defense Technology, China
taowang2005@nudt.edu.cn

Huaimin Wang
National University of Defense Technology, China
hmwang@nudt.edu.cn

ABSTRACT

Docker containers have become the de-facto industry standard. Docker builds often break, and a large amount of efforts are put into troubleshooting broken builds. Prior studies have evaluated the rate at which builds in large organizations fail. However, little is known about the frequency and fix effort of failures that occur in Docker builds of open-source projects. This paper provides a first attempt to present a preliminary study on 857,086 Docker builds from 3,828 open-source projects hosted on GitHub. Using the Docker build data, we measure the frequency of broken builds and report their fix time. Furthermore, we explore the evolution of Docker build failures across time. Our findings help to characterize and understand Docker build failures and motivate the need for collecting more empirical evidence.

KEYWORDS

Docker, Build failure, Open-source

ACM Reference Format:
Yiwen Wu*, Yang Zhang*, Tao Wang, and Huaimin Wang. 2020. An Empirical Study of Build Failures in the Docker Context. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387483

1 INTRODUCTION

Docker is one of the most popular containerization tools in current DevOps practice. It enables the encapsulation of software packages into containers and can run on any system [1]. Since inception in 2013, Docker containers have been downloaded 130B+ times[1]. The "Annual Container Adoption" report[2] found that 79% of companies chose Docker as their primary container technology.

With the widespread use and influence of Docker, many studies have been recently conducted to investigate its ecosystem [3], configuration file [6], workflow [12], and best configuration practices [10]. Those works have emerged a lot of great findings and brought many practical implications to developers, but were not designed to look into the details of Docker builds. Building is crucial to the software development process, which automates the process by which sources are compiled, linked, tested, packaged, and transformed into executable units [5]. In practices, builds often break (i.e., fail), and although this is not expected, broken builds can help developers to identify problems early before delivering products to end-users. Recently, the frequency and impact of build failures have been quantified in many contexts, e.g., C++ and Java builds [7], Ruby builds [2], and Continuous Integration (CI) builds [4, 9, 11]. However, to the best of our knowledge, little is known about the failure frequency and fix effort of builds in the Docker context.

To fill the gap in understanding Docker build failures (including frequency, fix effort, and their evolution), we present an empirical study of 857,086 Docker builds from 3,828 GitHub open-source projects. More specifically, we attempt to answer three Research Questions (RQs) in this paper:

- **RQ1: (Frequency) How often do Docker builds fail?** We find that the overall build failure rate in the Docker context is 17.8% and most of Docker projects (85.2%) in our dataset have at least one broken build. Frequently-built projects are associated with a low ratio of broken builds.
- **RQ2: (Fix effort) How long does it take to fix Docker build failures?** Broken Docker builds have a median fix time of 44.2 minutes in our study context. For each Docker project, more Docker build failures are related to longer fix time.
- **RQ3: (Evolution) How do failures frequency and fix effort evolve across time?** Overall, the failure rate and fix time of Docker builds fluctuate and gradually increase across time.

Paper organization. The rest of this paper is organized as follows: Section 2 describes the study setup. Section 3 presents our study results. Section 4 outlines the research agenda and Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 STUDY SETUP

Figure 1 gives an overview of our study. Based on the RQs, we collect the Docker build data from thousands of selected GitHub projects, and perform quantitative studies on them.

Data sources. Our data collection involves mining two types of sources: (1) GitHub data, i.e., projects, using the Google BigQuery[3]; and (2) Docker Hub data, i.e., Docker builds, using the Docker Hub API. Docker Hub is Docker's cloud-based registry, containing

---

[1] https://www.docker.com/company, as of March 2020.
[2] https://portworx.com/2017-container-adoption-survey/

*Both are first authors and contributed equally to this work.

[3] https://bigquery.cloud.google.com/dataset/bigquery-publicdata:github_repos

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387483

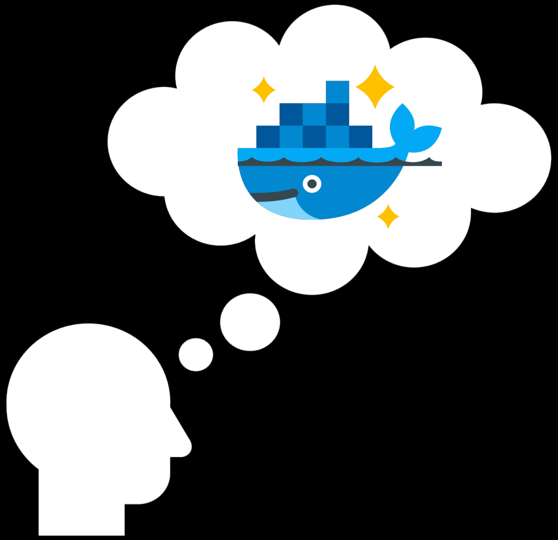
Wu et. al 2021

Security vulnerabilities

Build reliability issues

What about quality?

What we can do to improve the quality of Dockerfiles?



Step 1:

Improving quality by fixing Dockerfile smells

Step 1:

Improving quality by fixing Dockerfile smells

Code smells are **not the only factor** measuring quality

How do Dockerfile quality is perceived by developers?

What about the aspects related to the Docker image quality?

Step 2:

Quality Features impacting on the adoption of a Docker image

- What are the **quality aspects** of a Docker image (and its Dockerfile)?
- How do developers **perceive** them?

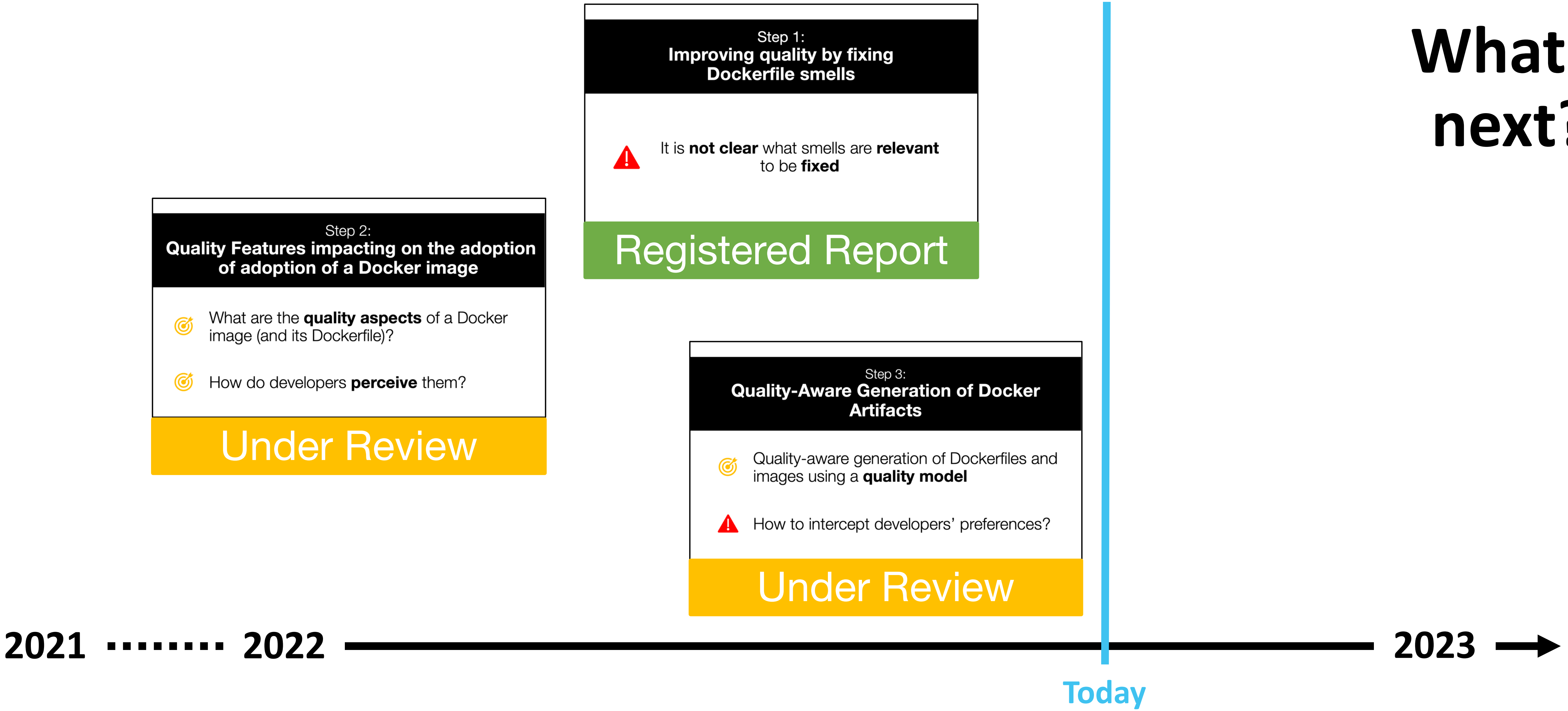
Step 3:

Quality-Aware Generation of Dockerfiles and Docker images

🎯 Quality-aware generation of Dockerfiles and images using a **quality model**

⚠ How to intercept developers’ preferences?

What’s next?



Step 1:
Improving quality by fixing Dockerfile smells

- ⚠ It is **not clear** what smells are **relevant** to be **fixed**

Registered Report

Step 2:
Quality Features impacting on the adoption of adoption of a Docker image

- ◎ What are the **quality aspects** of a Docker image (and its Dockerfile)?
- ◎ How do developers **perceive** them?

Under Review

Step 3:
Quality-Aware Generation of Docker Artifacts

- ◎ Quality-aware generation of Dockerfiles and images using a **quality model**
- ⚠ How to intercept developers’ preferences?

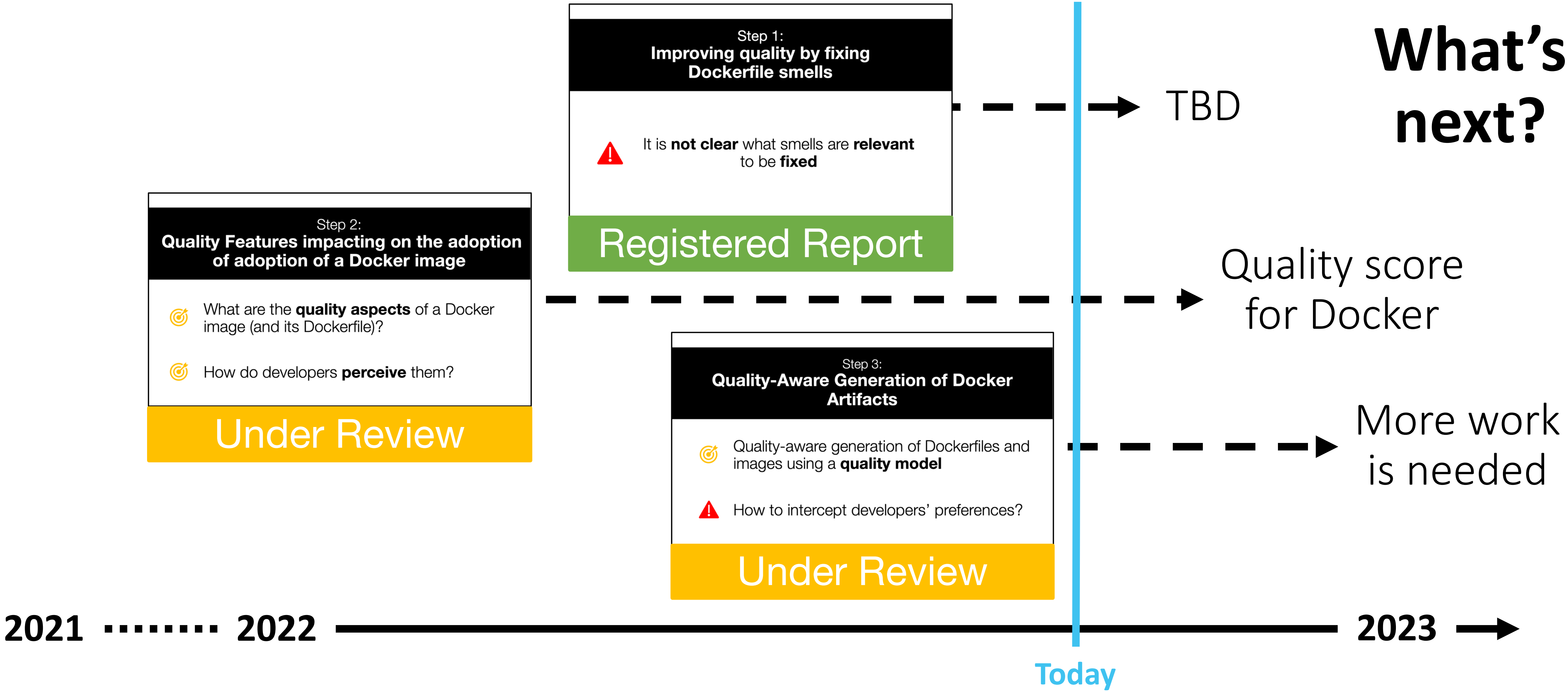
Under Review

Summary

Giovanni Rosa
STAKE Lab
University of Molise, Italy

giovanni.rosa@unimol.it

What’s next?



Step 1:
Improving quality by fixing Dockerfile smells

⚠ It is **not clear** what smells are **relevant** to be **fixed**

Registered Report

→ TBD

Step 2:
Quality Features impacting on the adoption of adoption of a Docker image

◎ What are the **quality aspects** of a Docker image (and its Dockerfile)?

◎ How do developers **perceive** them?

Under Review

→ Quality score for Docker

Step 3:
Quality-Aware Generation of Docker Artifacts

◎ Quality-aware generation of Dockerfiles and images using a **quality model**

⚠ How to intercept developers’ preferences?

Under Review

→ More work is needed

Giovanni Rosa
STAKE Lab
University of Molise, Italy

giovanni.rosa@unimol.it

Questions?