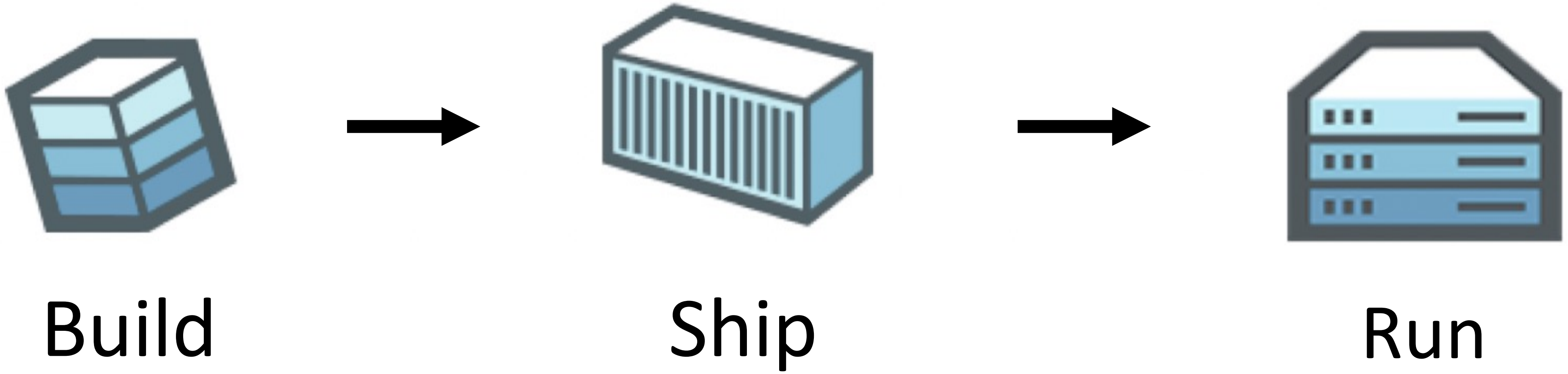
Automatically Generating Dockerfiles via Deep-Learning: Challenges and Promises

<u>Giovanni Rosa</u>, Antonio Mastropaolo, Simone Scalabrino, Gabriele Bavota and Rocco Oliveto

@giovannipink

University of Molise, Italy

17th International Conference on Software and System Processes
14-15 May 2023, MCEC (Melbourne)



Software Containers

85% of

organizations will adopt containers by **2025**

Gartner

85% of

organizations will adopt containers by **2025**

Gartner

#1 Most-Wanted

and

#1 Most Loved

tool

2022 Developer Survey

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

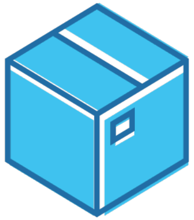
run

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

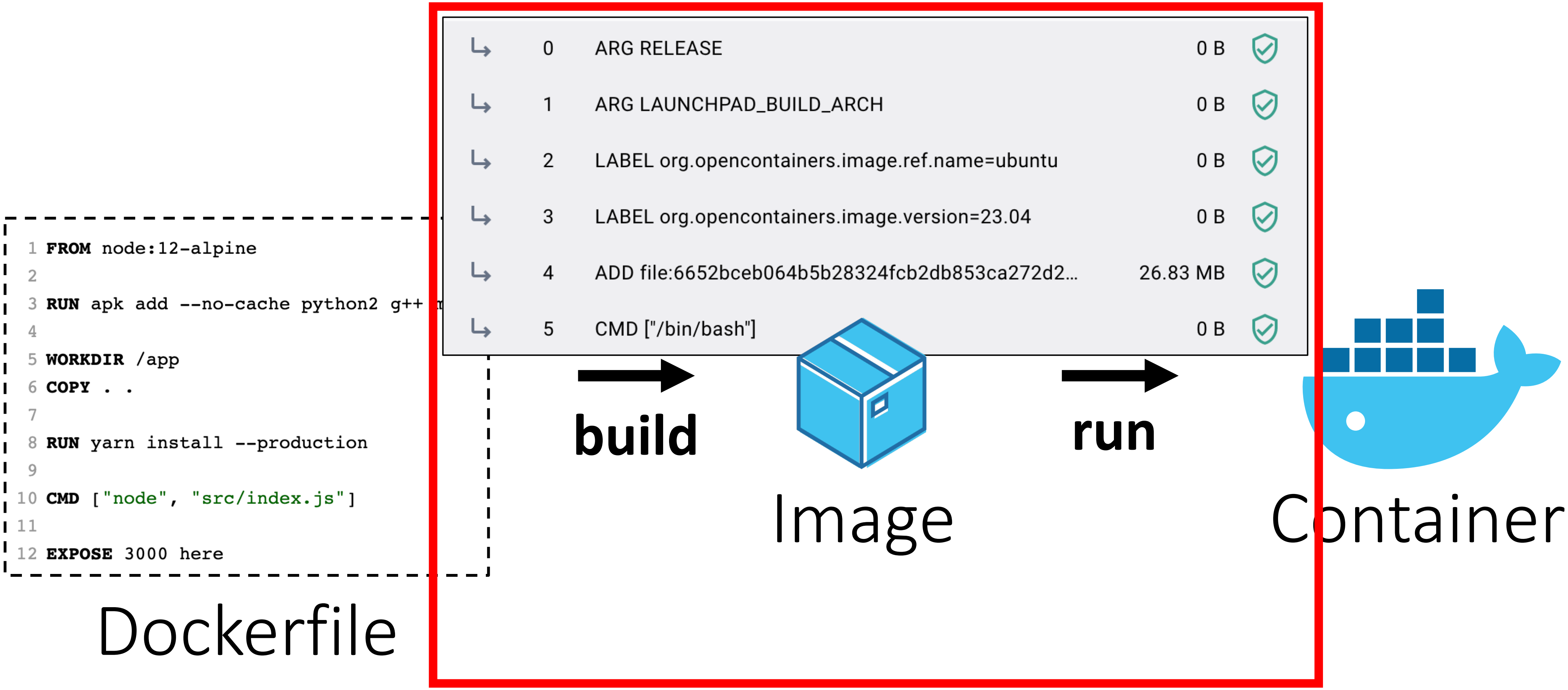


Image

run

Container

Docker in a Nutshell



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

Writing
Dockerfiles
is **challenging**

Time-consuming activity

Time-consuming activity

Technical knowledge required

Time-consuming activity

Technical knowledge required

Limited supporting tools

AI-based
Code Assistants

AI-based
Code Assistants

...limited to
dependencies
and
code completion

The automated generation **from scratch** is still an **open challenge**

What about **ChatGPT?**

Prompt:

Can you provide a Dockerfile using **Debian 9** and installing **Golang 1.9.4** via sources?

Not Bad!

but...

build error

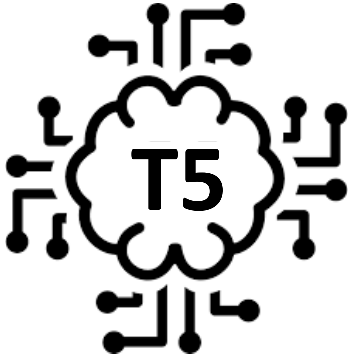
☹ ☹

dockerfile Copy code

```dockerfile
# Use Debian 9 as the base image
FROM debian:9

# Install required dependencies
RUN apt-get update && apt-get install -y \
    wget \
    build-essential \
    git

# Download and extract Go 1.9.4
RUN wget https://dl.google.com/go/go1.9.4.linux-amd64.tar.gz \
    && tar -C /usr/local -xzf go1.9.4.linux-amd64.tar.gz \
    && rm go1.9.4.linux-amd64.tar.gz

# Set Go environment variables
ENV GOROOT=/usr/local/go
ENV GOPATH=/go
ENV PATH=$GOROOT/bin:$GOPATH/bin:$PATH

# Create a workspace directory for Go projects
RUN mkdir -p $GOPATH/src $GOPATH/bin

# Set the working directory
WORKDIR $GOPATH/src

# Define the entry point
CMD ["bash"]
```

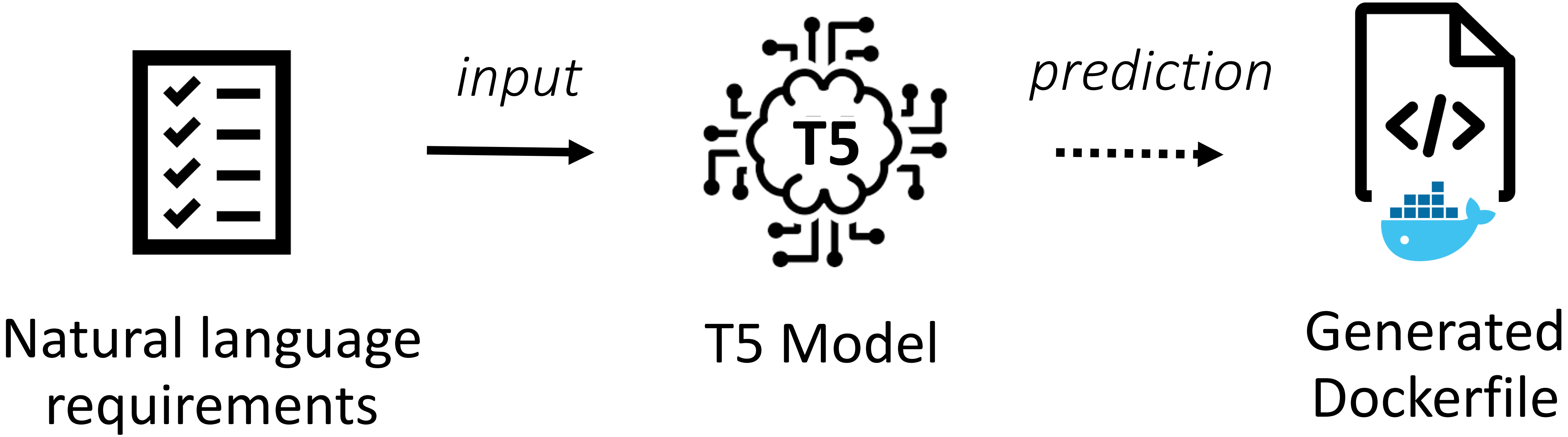
Regenerate response

What about
state-of-the-art DL models
for code-related tasks?

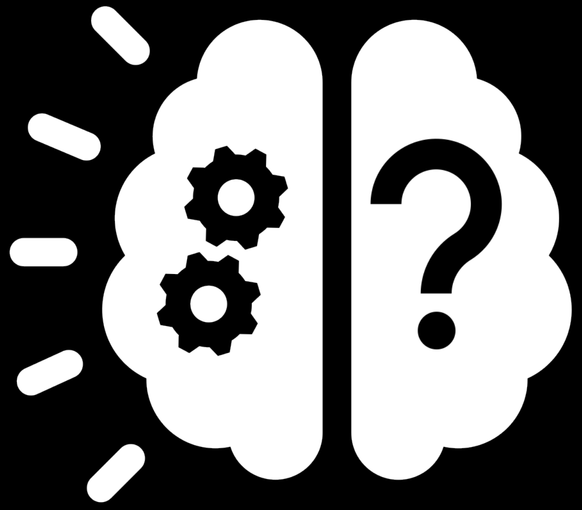


Phase 1:

Model Construction



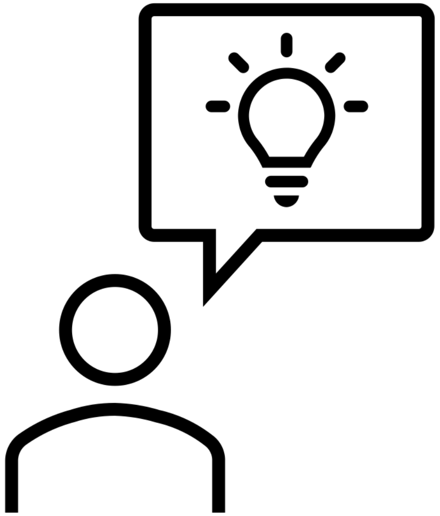
Dockerfile Generation via T5



How to represent **software requirements** for a Dockerfile?

Natural Language:
Too Broad!

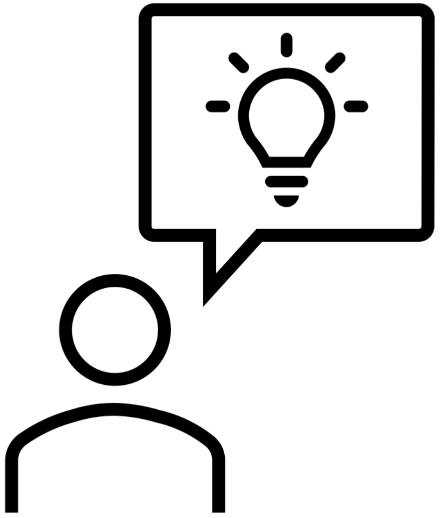
Operating System: “alpine”
Package Manager: “apk”
Package Requirements: [“python3”]
Download from Sources: FALSE
ENV variables: FALSE
Build ARGs: FALSE
LABEL: TRUE
EXPOSE for ports: TRUE
CMD: TRUE
ENTRYPOINT: FALSE



HLS

High-Level Specification

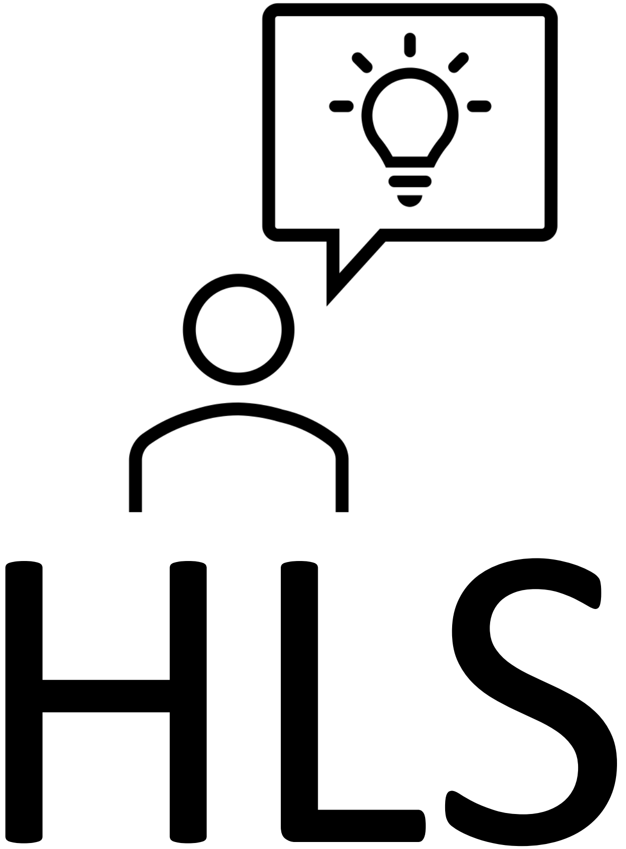
Operating System: “alpine”
Package Manager: “apk”
Package Requirements: [“python3”]
Download from Sources: FALSE
ENV variables: FALSE
Build ARGs: FALSE
LABEL: TRUE
EXPOSE for ports: TRUE
CMD: TRUE
ENTRYPOINT: FALSE



HLS

High-Level Specification

Operating System: “alpine”
Package Manager: “apk”
Package Requirements: [“python3”]
Download from Sources: FALSE
ENV variables: FALSE
Build ARGs: FALSE
LABEL: TRUE
EXPOSE for ports: TRUE
CMD: TRUE
ENTRYPOINT: FALSE



High-Level Specification

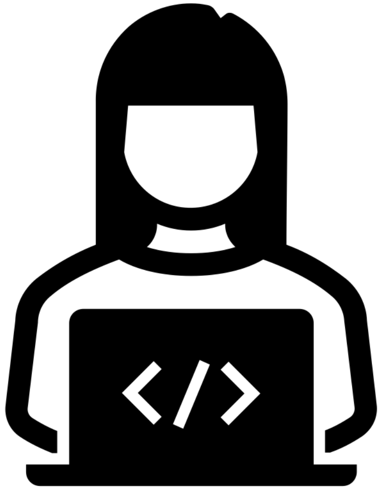
```
FROM alpine:3.17

LABEL maintainer="happy@container.net"

# Install python3
RUN apk add --update --no-cache python3

COPY . .

RUN pip install -r requirements.txt

WORKDIR /tiddlywiki

EXPOSE 8000
```

OS:
“alpine3.17”

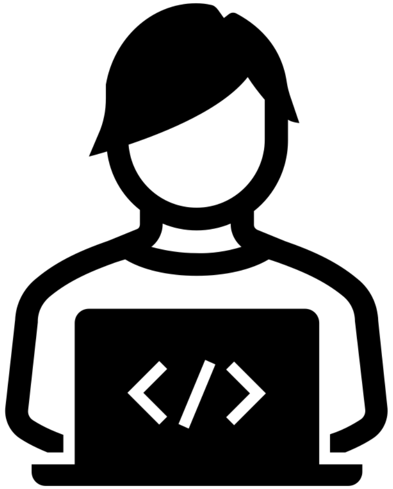
Pkg. Manager:
“apk”

Pkg. Requirements:
“python3”

High-Level Specification



+2 years



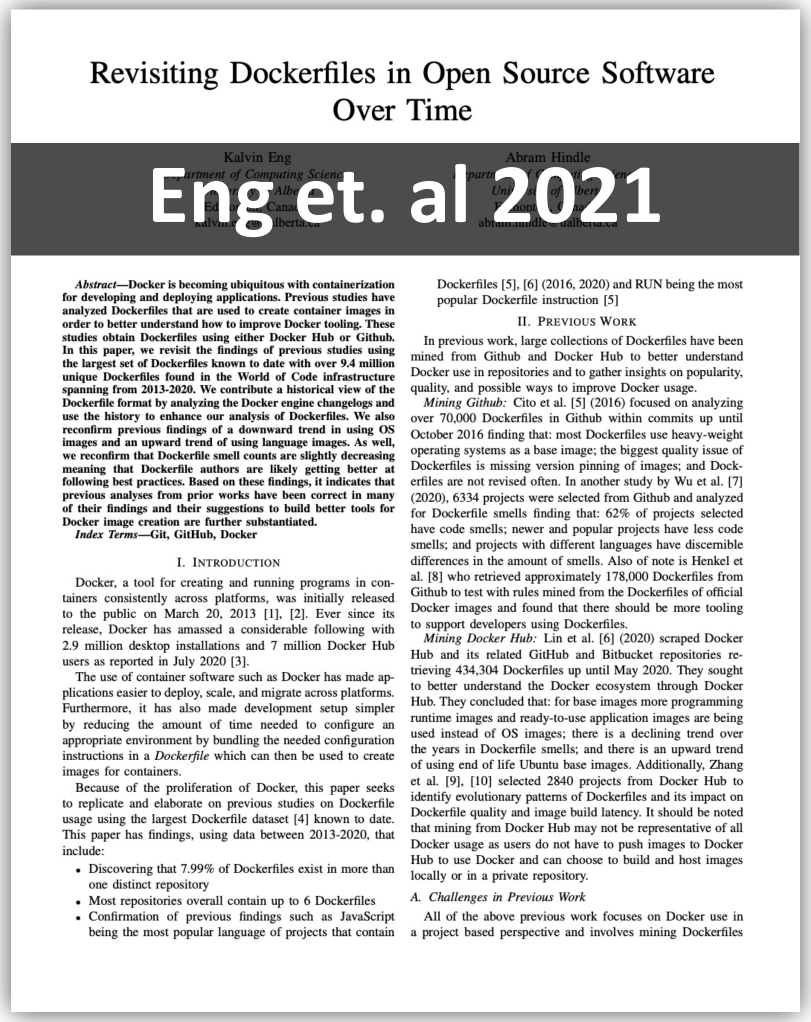
12

software developers

>50%

agrees with the **requirements specification**

Asking Developers’ Opinion



Revisiting Dockerfiles in Open Source Software Over Time

Kalvin Eng

Abram Hindle

abram.hindle@ualberta.ca

Eng et. al 2021

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

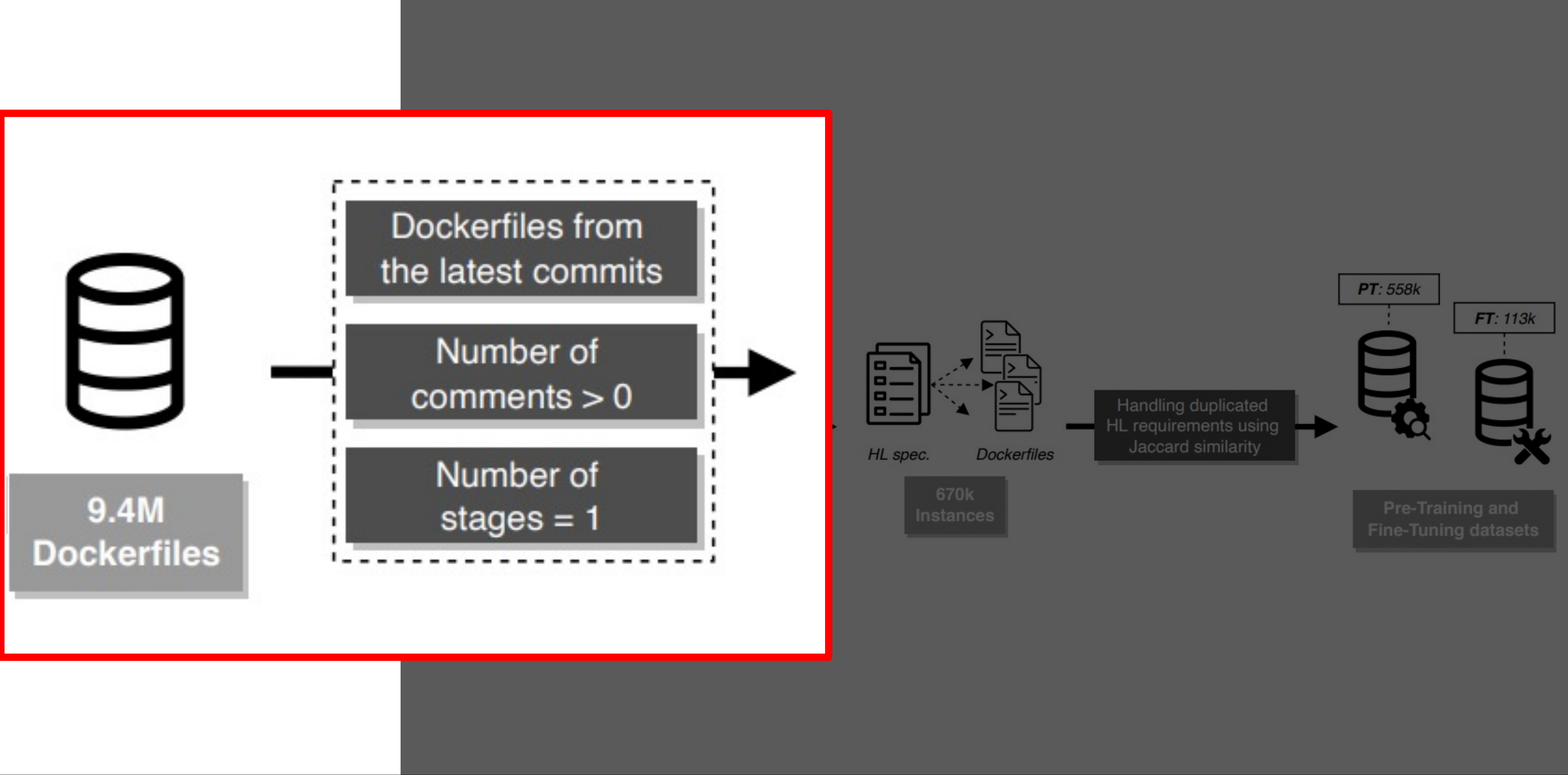
All of the above previous work focuses on Docker use in a project based perspective and involves mining Dockerfiles

9.4M

unique Dockerfiles

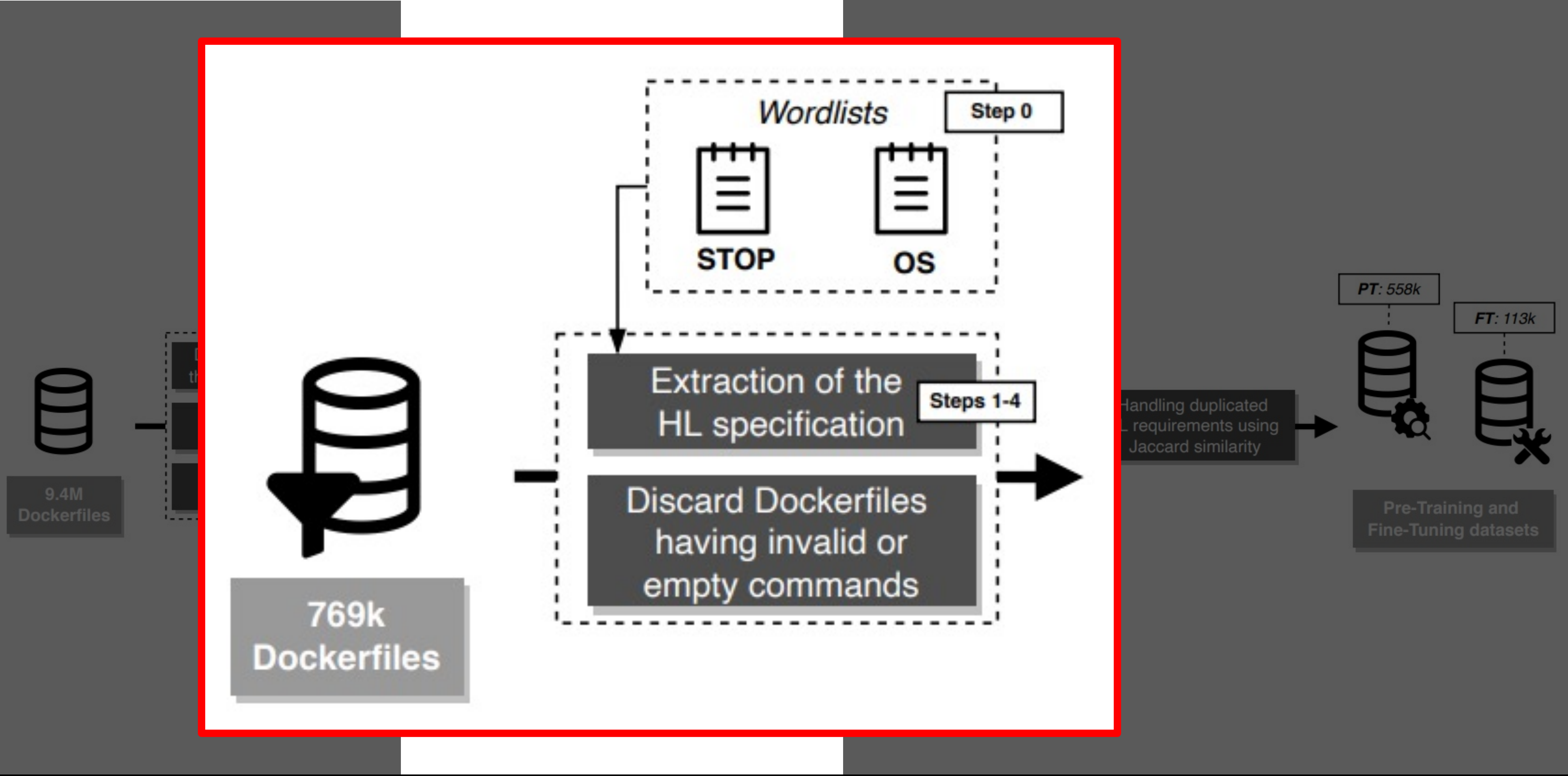
from **2013** to **2020**

Dockerfile Dataset

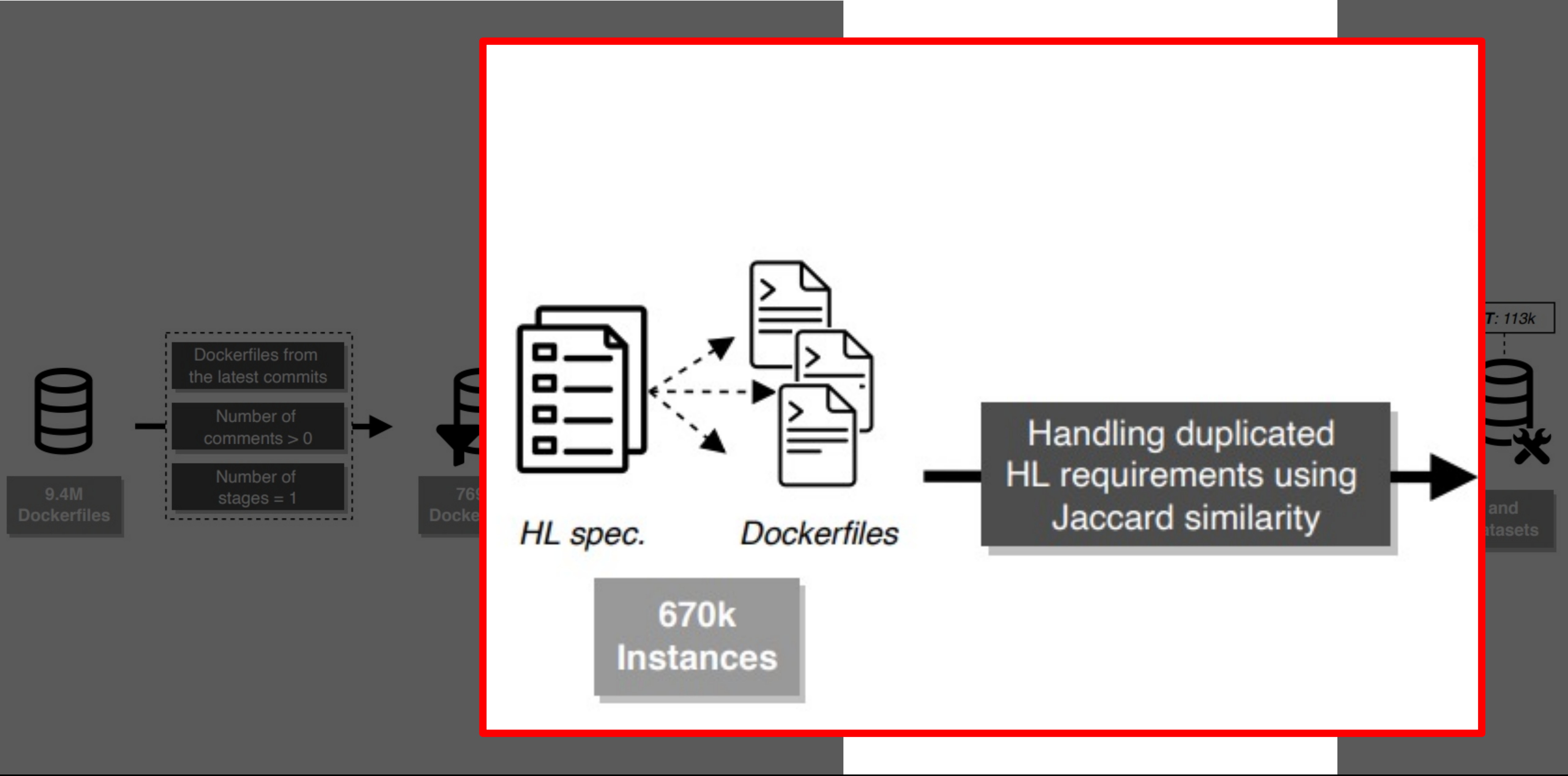


1

Dataset Filtering



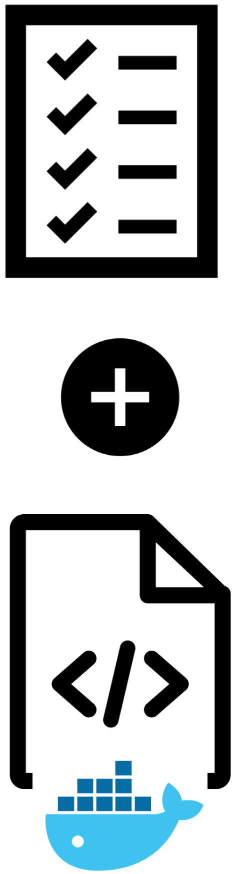
2 Dataset Filtering



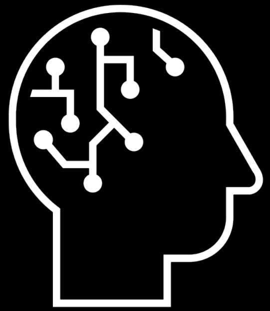
3 Dataset Filtering

100k Model Tuning

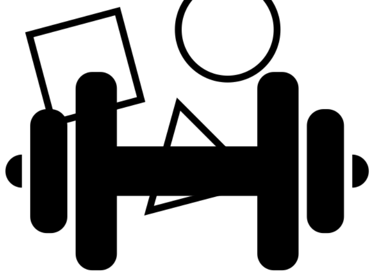
11k Test



Resulting Dataset



T5 model construction

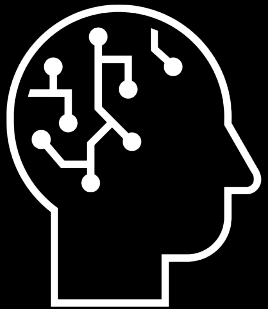


Pre-Training

560k instances

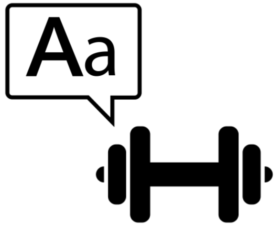
Fine-Tuning

90k instances

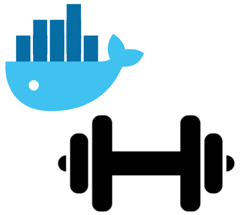


T5 model construction

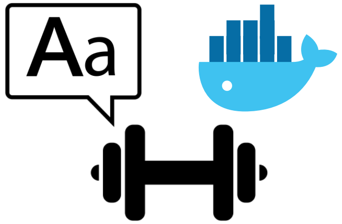
3 pre-training settings



English Only



Dockerfile Only



Dockerfile & English

T5 model construction

Fine Tuning

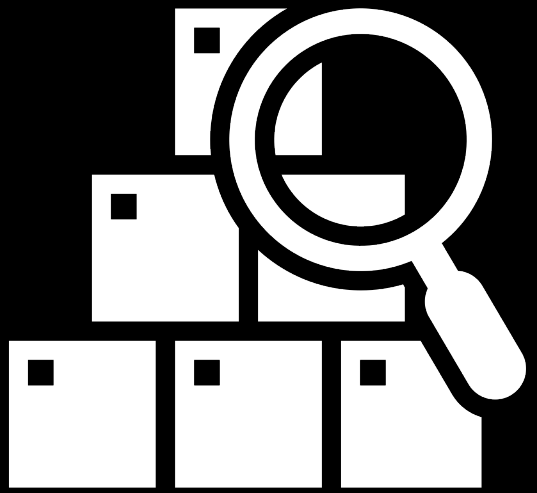
English Only

Dockerfile Only

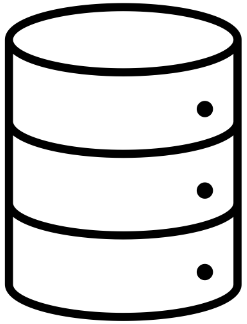
Dockerfile & English

Phase 2:

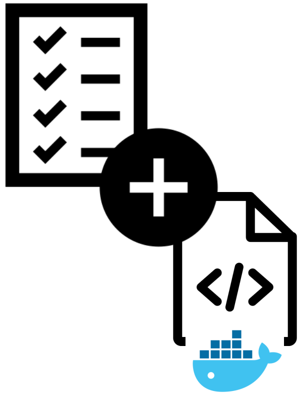
Model evaluation



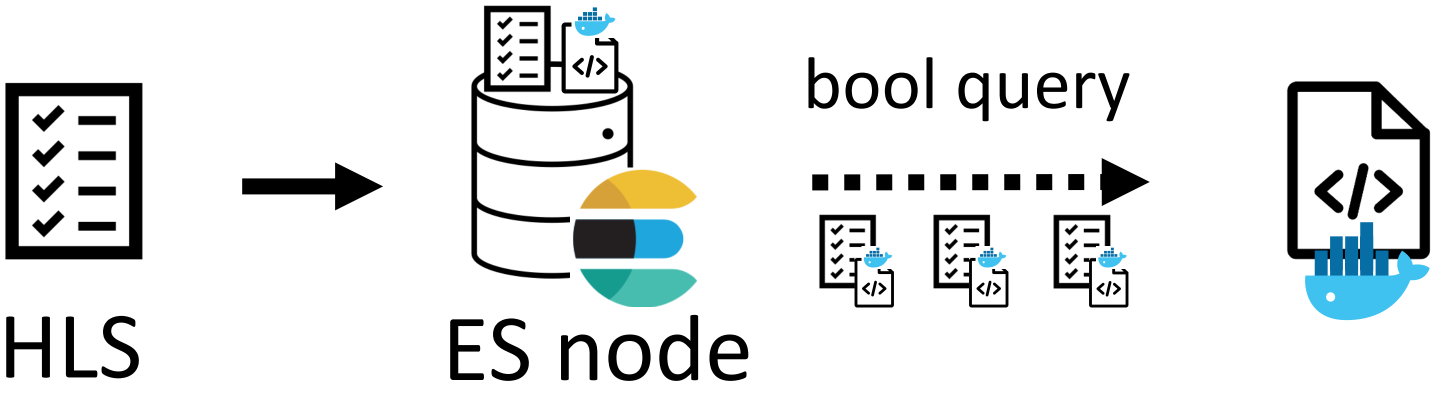
2 baselines



90k instances

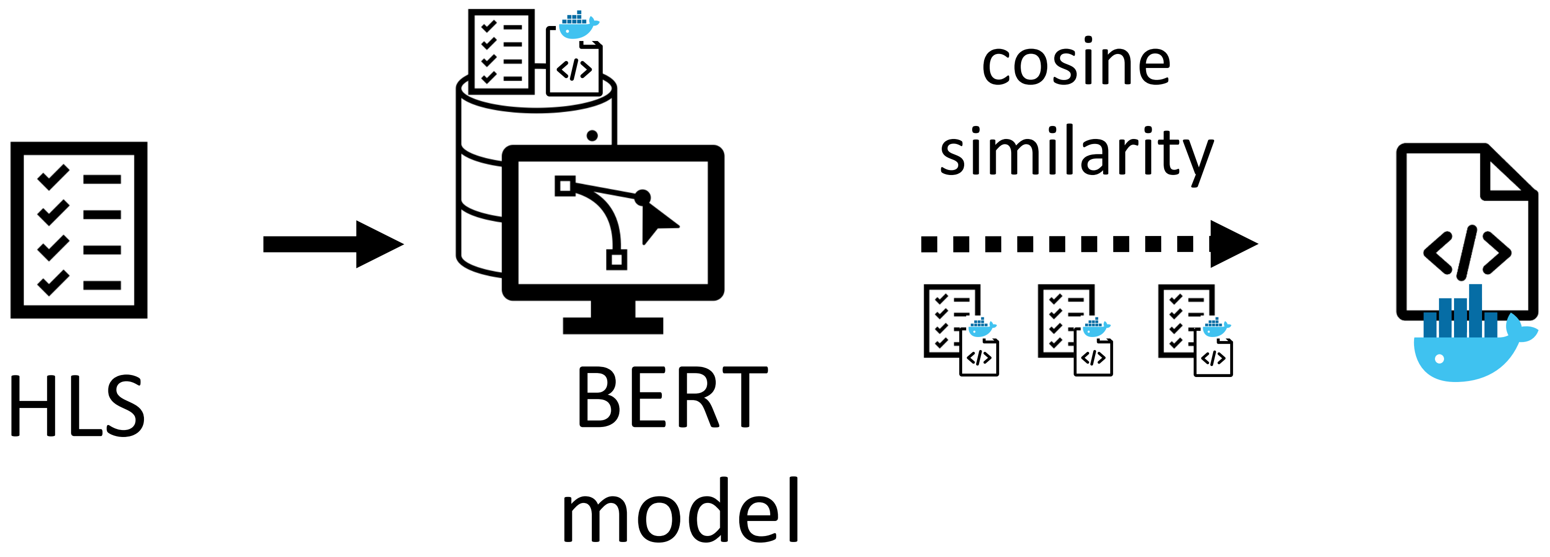


Elasticsearch



IR-Baseline 1

SentTransformers



IR-Baseline 1

3 dimensions

11k Test

Evaluation

RQ1 Adherence to the input High-Level Specification

```json
{
  "os": "alpine",
  "pkg_manager": "any",
  "requirements": [
      "python3"
  ],
  "uses_env": false,
  "uses_arg": false,
  "uses_label": true,
  "uses_expose": true,
  "uses_cmd": true,
  "uses_entrypoint": false,
  "download_of_external_packages": false
}
```

Input HLS

VS

```json
{
  "os": "alpine",                              +1
  "pkg_manager": "any",                        +1
  "requirements": [
      "python3"                                +1
  ],
  "uses_env": false,                           +1
  "uses_arg": false,                           +1
  "uses_label": true,                          +1
  "uses_expose": true,                         +1
  "uses_cmd": true,                            +1
  "uses_entrypoint": false,                    +1
  "download_of_external_packages": false       +1
}
```

Generated/Retrieved HLS

Metric:

Field-by-field match

	T5	ES	SBERT.net
OS	**~1.00**	0.92	0.88
Pkg. Manager	0.98	**1.00**	**1.00**
Pkg. Requirements	0.87	**0.88**	0.76
Download from sources	0.82	**0.84**	0.52
ENV variables	**0.89**	0.81	0.17
Build ARGs	**0.99**	0.88	0.17
LABEL	**~1.00**	0.87	0.37
EXPOSE for ports	0.80	**0.83**	0.45
CMD	0.74	**0.83**	0.26
ENTRYPOINT	0.84	**0.85**	0.45

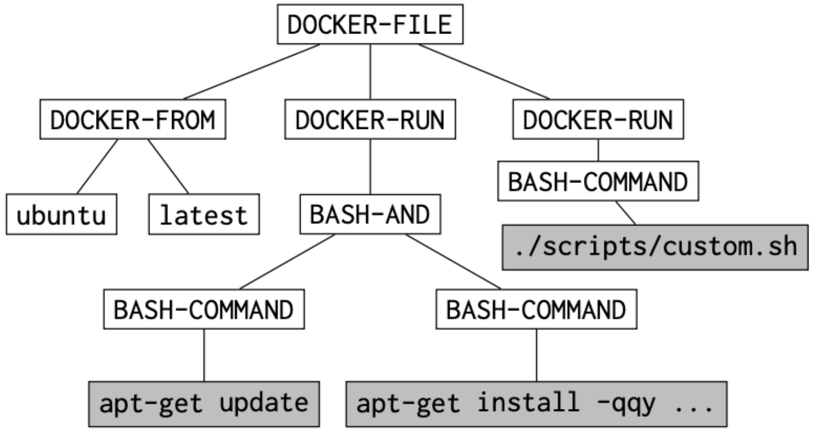
Results for RQ1

RQ2 Structural similarity between Dockerfiles

```
FROM ubuntu:latest

RUN apt-get update && \
    apt-get install -qqy ...

RUN ./scripts/custom.sh
```

Input Dockerfile

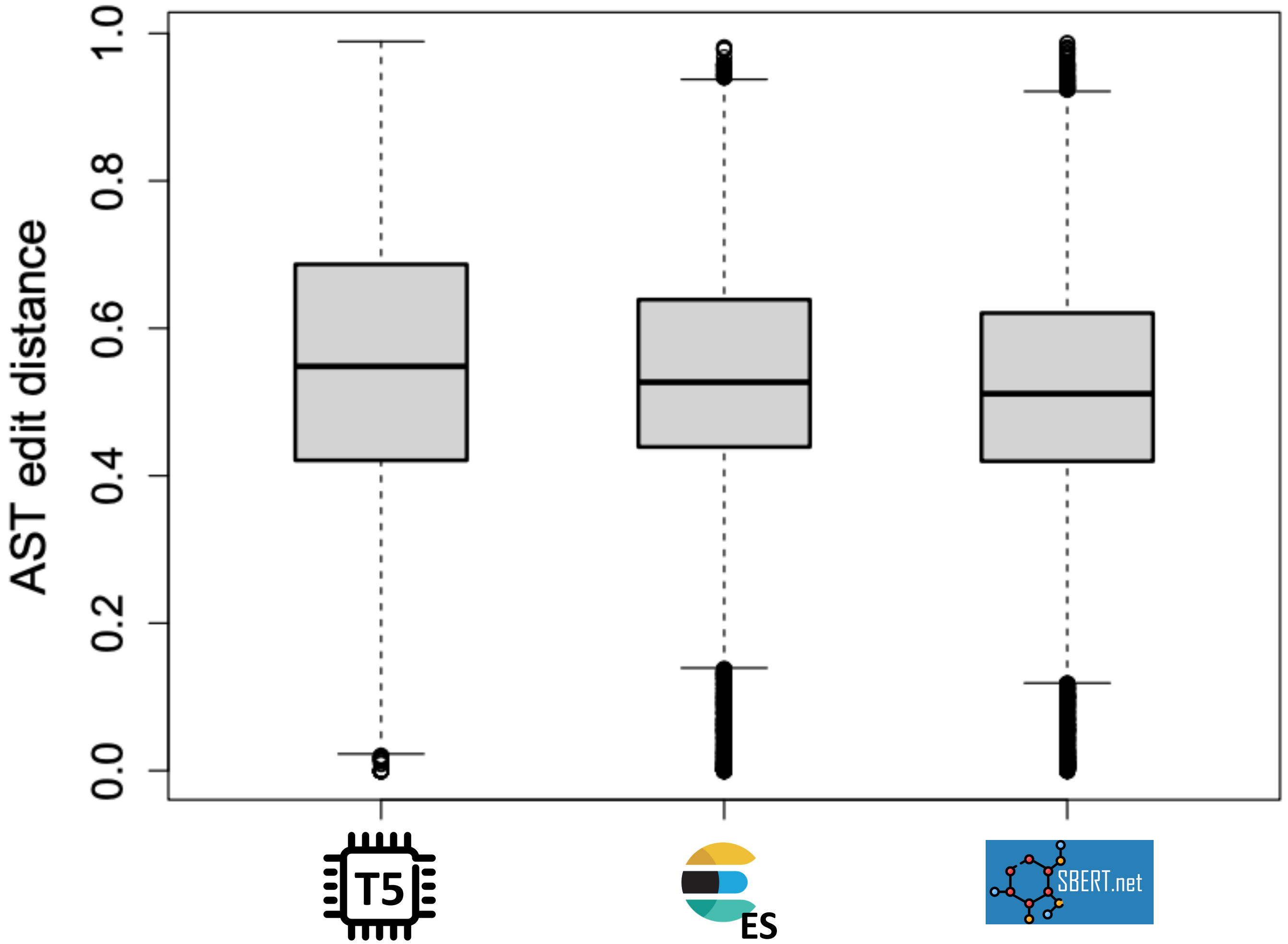
→



Parsed AST
(Henkel et. al 2020)

Metric:
AST
edit distance

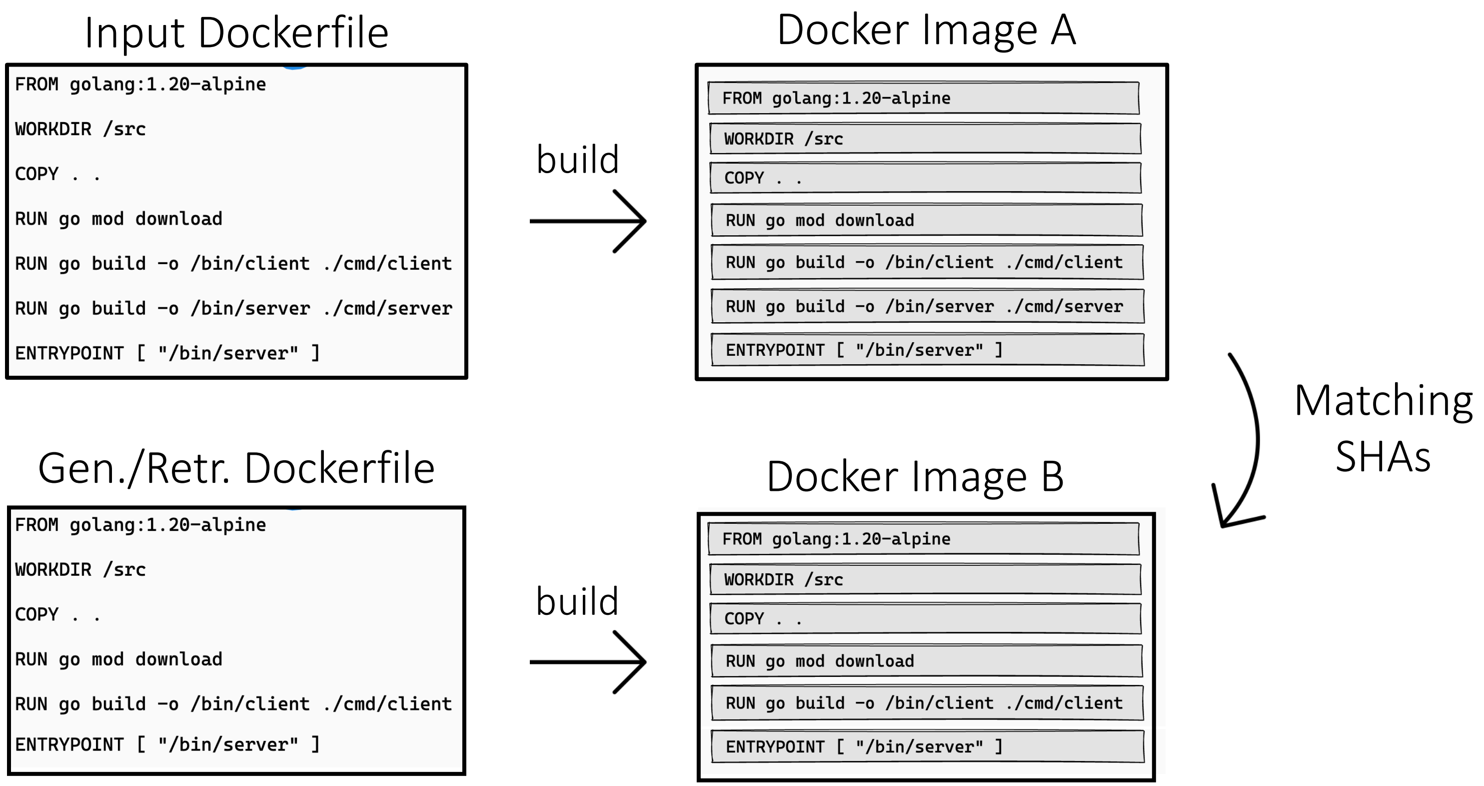
RQ2 Structural similarity between Dockerfiles



The lower is better

Results for RQ2

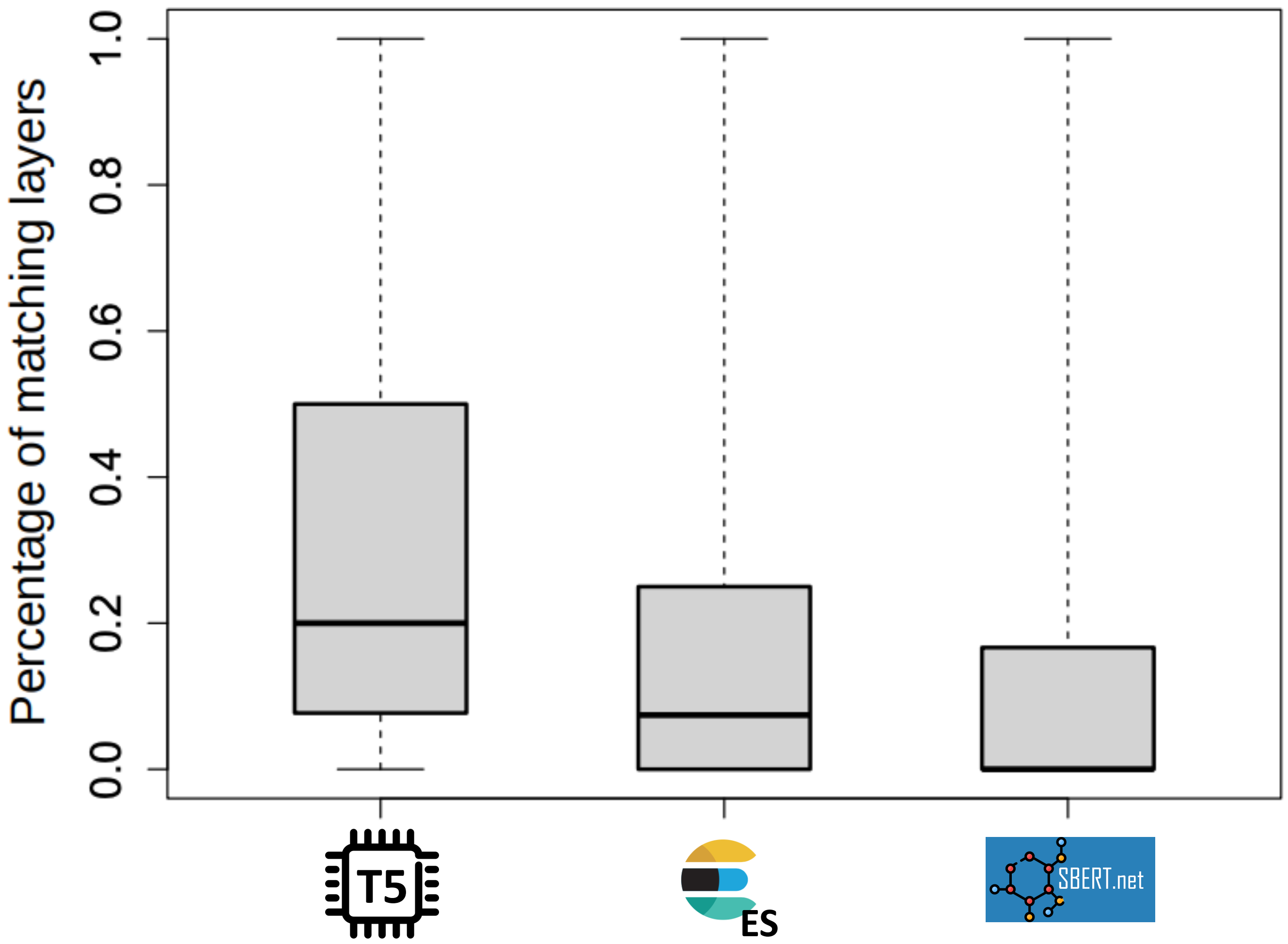
RQ3 Similarity between Docker images



Metric:

Percentage of matching layers

RQ3 Similarity between Docker images



Results for RQ3

Summary

T5 achieves slightly better results than IR ...

Summary

T5 achieves slightly better results than IR ...

More resource-consuming compared to IR

Generated Dockerfiles require manual adjustments

What we have learned?

Challenge #1

Not enough training instances

Challenge
#1

Not enough training instances

Data augmentation

Challenge #2

A different training procedure must be used

Challenge
#2

A different training procedure must be used

Different stopping criterion

Dockerfile abstractions

Summary

Writing Dockerfiles is **challenging**

input → *prediction*

Natural language requirements

T5 Model

Generated Dockerfile

Dockerfile Generation via T5

3 dimensions

11k test

Evaluation

T5 model construction

Fine Tuning

English Only

Dockerfile Only

Dockerfile & English

Summary

- T5 achieves slightly better results than IR ...
- More resource-consuming compared to IR
- Generated Dockerfiles require manual adjustments

What we have learned?

Giovanni Rosa

https://giovannirosa.com

Thank you!