

# Automatically Generating Dockerfiles via Deep-Learning: Challenges and Promises

Giovanni Rosa, Antonio Mastropaolo, Simone Scalabrino,  
Gabriele Bavota and Rocco Oliveto



 [@giovannipink](https://twitter.com/giovannipink)

University of Molise, Italy

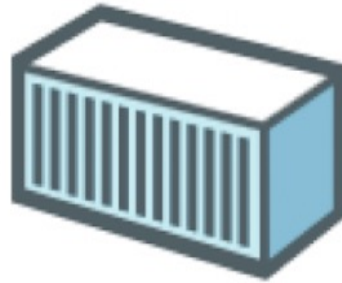


17th International Conference on  
Software and System Processes  
14-15 May 2023, MCEC (Melbourne)





Build



Ship

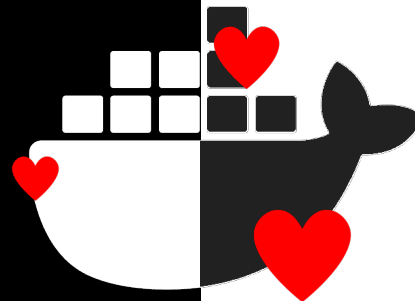


Run

Software Containers

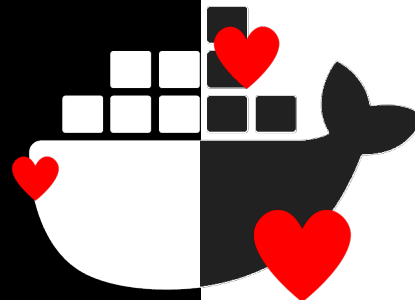
**85%** of  
organizations will  
adopt containers  
by **2025**

**Gartner**<sup>®</sup>



**85%** of  
organizations will  
adopt containers  
by **2025**

**Gartner**<sup>®</sup>



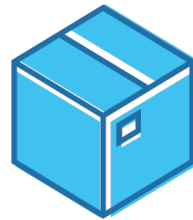
**#1 Most-Wanted**  
and  
**#1 Most Loved**  
tool



```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++ make
4
5 WORKDIR /app
6 COPY . .
7
8 RUN yarn install --production
9
10 CMD ["node", "src/index.js"]
11
12 EXPOSE 3000 here
```

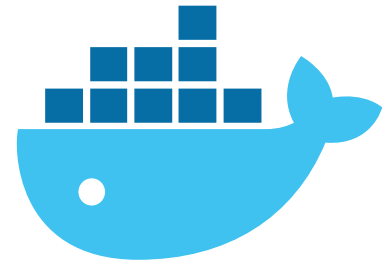
Dockerfile

→  
**build**



Image

→  
**run**



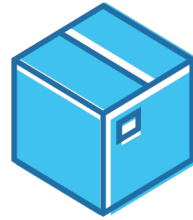
Container

# 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 . .
7
8 RUN yarn install --production
9
10 CMD ["node", "src/index.js"]
11
12 EXPOSE 3000 here
```

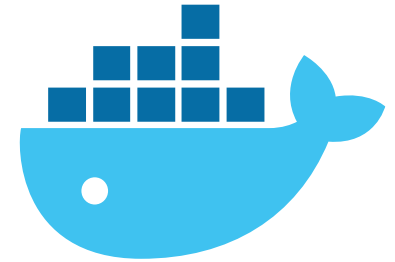
Dockerfile

→  
**build**



Image

→  
**run**



Container

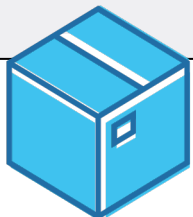
Docker in a Nutshell

```
1 FROM node:12-alpine
2
3 RUN apk add --no-cache python2 g++
4
5 WORKDIR /app
6 COPY . .
7
8 RUN yarn install --production
9
10 CMD ["node", "src/index.js"]
11
12 EXPOSE 3000 here
```

Dockerfile

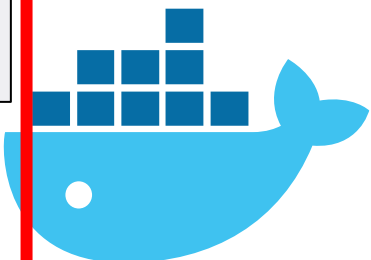
↳	0	ARG RELEASE	0 B	✓
↳	1	ARG LAUNCHPAD_BUILD_ARCH	0 B	✓
↳	2	LABEL org.opencontainers.image.ref.name=ubuntu	0 B	✓
↳	3	LABEL org.opencontainers.image.version=23.04	0 B	✓
↳	4	ADD file:6652bceb064b5b28324fcb2db853ca272d2...	26.83 MB	✓
↳	5	CMD ["/bin/bash"]	0 B	✓

build



Image

run



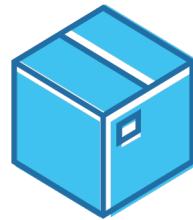
Container

# 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 . .
7
8 RUN yarn install --production
9
10 CMD ["node", "src/index.js"]
11
12 EXPOSE 3000 here
```

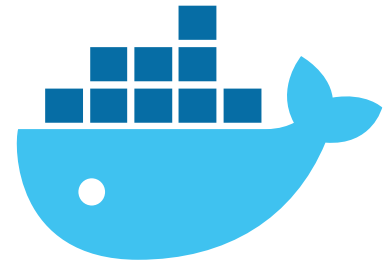
Dockerfile

→  
**build**



Image

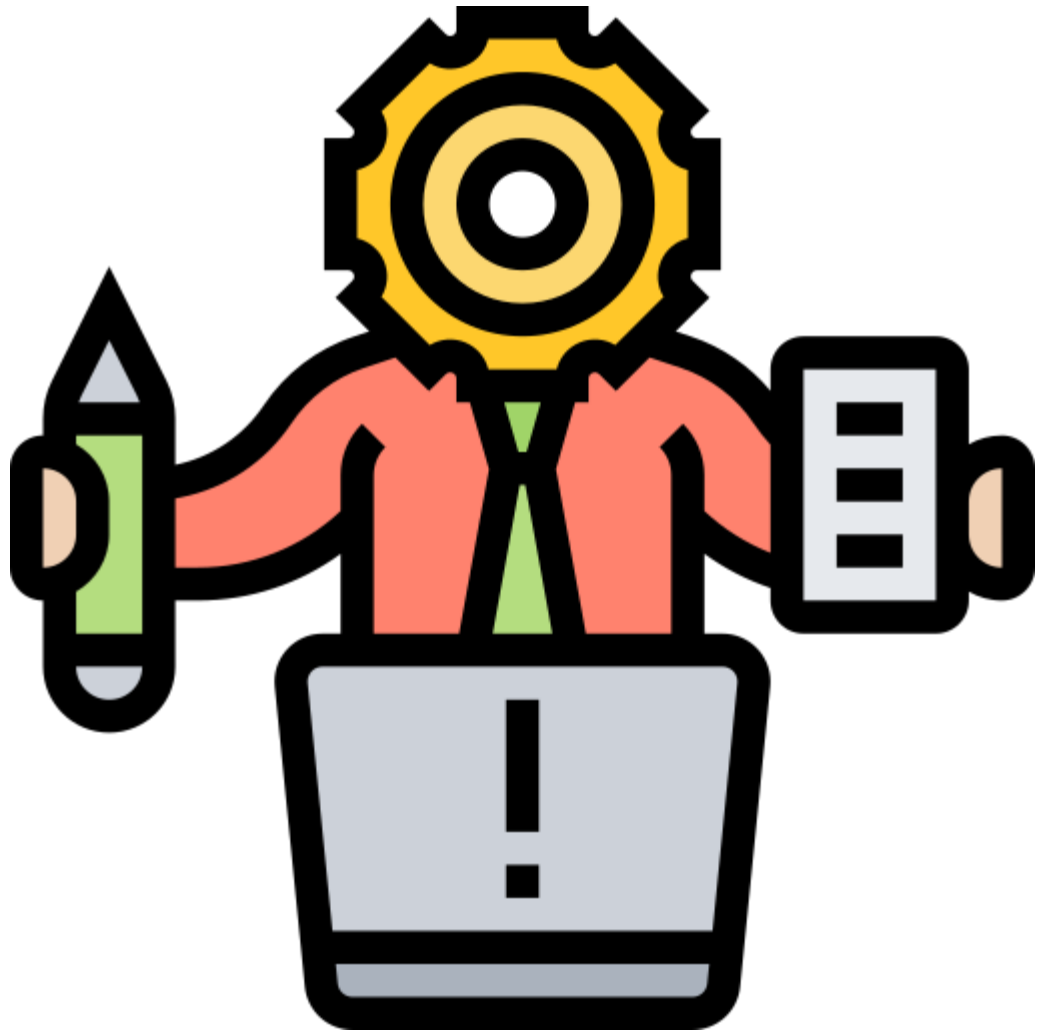
→  
**run**



Container

# Docker in a Nutshell





**Writing  
Dockerfiles  
is challenging**



# Time-consuming activity

Received November 20, 2021, accepted December 18, 2021, date of publication December 22, 2021,  
date of current version January 7, 2022.  
Digital Object Identifier 10.1109/CSSE.2021.9623502

## Developing Docker and Docker-Compose Specifications: A Developers' Survey

DAVID REIS<sup>1</sup>, BRUNO PIEDADE<sup>1</sup>, FILIPE F. CORREIA<sup>1,2</sup>,  
IAGO PEDRO DIAS<sup>1</sup>, AND ANDRÉ AGUIAR<sup>1</sup>

<sup>1</sup>FEUC, Universidade Federal de Uberlândia, Uberlândia, Brazil  
<sup>2</sup>FEIS, Universidade Federal do Espírito Santo, Vitória, Brazil

### Reis et. al 2021

**ABSTRACT** Cloud computing and Infrastructure-as-Code (IaC), supported by technologies such as Docker, have shaped how many software systems are built and deployed. Previous research has identified typical issues for some types of IaC specification but not why they come to be, or they have delved into collaboration aspects but not into technical ones. This work aims to characterize the activities around two particular kinds of IaC specification—Dockerfiles and *docker-compose.yml* files. We seek to know how they can be better supported and therefore study also what approaches and tools practitioners employ. We used an online questionnaire to gather data. The first part of the study reached 68 graduate students from a study program in informatics engineering, and the second one 120 professional software developers. The results show that most of the activities of the process of developing a Dockerfile are perceived as time-consuming, especially when the respondents are beginners with this activity. We also found that solving issues using trial-and-error approaches is very common. We do not use ancillary tools to support the development of Dockerfiles.

INDEX TERMS: Docker, Dockerfile, Docker Compose, Infrastructure-as-Code, survey

## Learning from, Understanding, and Supporting DevOps Artifacts for Docker

Josiah Henkel<sup>1</sup>,  
University of Wisconsin, Madison, USA

Christian Bird<sup>2</sup>,  
Microsoft, Redmond, USA

### Henkel et. al 2020

**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, *bintrace*, that enabled us to ingest 900,000 of semantic rule-based artifacts.

**KEYWORDS** Program semantics, Abstraction, Information systems, Data mining, Docker, DevOps, Mining, Static Checking

**1 INTRODUCTION** With the continued growth and rapid iteration of software, an increasing amount of attention is being placed on servers 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 applies to a large number of practices, including deployment, testing, managing, and supporting a software system [2]. Bass et al. seeks to make a bridge between both practices [2]. Bass et al. between committing a change to a system and the change being placed into normal production, while ensuring high quality [1]. DevOps activities include building, testing, packaging, releasing, and monitoring software. To aid developers in these processes, tools such as Terraform [9], CircleCI [1], Docker [1], 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, [4] and [22]) and various practices of DevOps have been studied extensively [20, 27, 31, 37, 38, 40].

DevOps tools exist in a heterogeneous 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 [1], and Terraform [6], 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 [23]). They are not "traditional" code, and therefore out of the reach of various efforts in automatic code analysis and analysis, but at the same time, these artifacts are common and analyzed with developers tasked with working on their projects. Our literature survey with developers tasked with working on their artifacts indicate that they learn just enough to "get the job done"

cloud computing, survey, programming languages. While some aspects and activities of the process may be the same, others seem to be different—from testing to debugging, to the error-proneness and longer feedback loops [8]. Although there is a fair amount of ancillary tools for the development of infrastructure specifications [5], [9], few works try to empirically demonstrate the improvements they may bring to the development process [10]–[12]. In particular, there is scarce empirical evidence that the *issues* that many of these tools address are worth addressing and that the approaches that they prescribe are addressing such issues effectively. In this work, we seek insights on how software professionals perceive the use they do of Docker and Docker-Compose and to generate new hypotheses of how best to address existing challenges. In the remainder of this paper, Section II overviews works related to our study. Section III identifies the main goal of the research and its questions, and Section IV the methodology to answer them. Sections V and VI then describe, respectively, data handling and analysis, offering this work's main contributions. Section VII overviews the main threats to validity.

ACM Reference Format: Henkel, Josiah, Bird, Christian, and Aguiar, André. 2020. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *32nd 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/10.1145/3377913.3380008>

Permission to make digital or physical copies of this work is permitted under the Creative Commons Attribution 4.0 International License. For more information, see <http://creativecommons.org/licenses/by/4.0/>.

© 2020 Association for Computing Machinery  
0730-0015/20/0000-0000 \$15.00  
<https://doi.org/10.1145/3377913.3380008>

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

18





# DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets

Horton et. al 2019

Abstract—Python like Stack Overflow and GitHub's Git system promote the sharing of ideas and programming techniques. The distribution of code snippets designed to illustrate particular tasks, Python, a popular and fast-growing programming language, see heavy use on both sites with nearly one million questions asked on Stack Overflow and 400 thousand public gist snippets on GitHub. Unfortunately, around 75% of the Python example code shared through these sites cannot be directly executed. When run in a clean environment, over 40% of public Python gists fail due to an import error for a missing dependency. This work focuses on automating the dependency resolution process of system-level dependencies. Being able to save dependencies in a code snippet is a code snippet resolution to a code snippet. We implement DockerizeMe, a code snippet resolution to a code snippet. We implement DockerizeMe, a code snippet resolution to a code snippet. We implement DockerizeMe, a code snippet resolution to a code snippet.

# Humpback: Code Completion System for Dockerfiles Based on Language Models

Hanayama et. al 2020

Abstract—Docker is the de-facto standard containerization platform. Docker is a relatively new technology, some domains of which have not been fully researched. In this study, we focus on code completion and aim to construct a system that supports the development of Dockerfiles. The proposed code completion system, Humpback, applies machine learning to overcome a Docker-specific code completion problem. Evaluation experiments show that Humpback has a high average accuracy of 96.9%.

## 1. Introduction

Server virtualization is broadly used for cost reduction and efficient resource utilization. Among various methods of virtualization, containerization has become mainstream [1]. Containerization provides an independent environment. Docker<sup>1</sup> is the de-facto standard containerization platform [2]. Containers in Docker are configured by writing imperative instructions in files called Dockerfiles. The process of managing infrastructure configuration through machine-readable Dockerfiles is called infrastructure as code (IaC). Infrastructure as code (IaC) is a relatively new technology, some domains of which have not been fully researched. In this study, we focus on code completion and aim to construct a system that supports the development of Dockerfiles. The proposed code completion system, Humpback, applies machine learning to overcome a Docker-specific code completion problem. Evaluation experiments show that Humpback has a high average accuracy of 96.9%.

<sup>1</sup> Docker is located with ANSEC 2020, 63-Dve, 2020, Singapore. <https://hub.docker.com/>  
<sup>2</sup> Hanayama et al. (S. Matsumoto), [humpback@inf.usioka.ac.jp](mailto:humpback@inf.usioka.ac.jp)  
<sup>3</sup> Hanayama et al. (S. Matsumoto), [humpback@inf.usioka.ac.jp](mailto:humpback@inf.usioka.ac.jp)  
<sup>4</sup> <https://inf.usioka.ac.jp/>  
<sup>5</sup> <https://inf.usioka.ac.jp/>

# DockerGen: A Knowledge Graph based Approach for Software Containerization

Ye et. al 2021

Abstract—Docker is the de-facto container technology for software system deployment and delivery. A Dockerfile specifies how to containerize a system into a Docker image. However, creating a Dockerfile is not trivial since resolving the dependencies (e.g., third-party libraries of diverse software requires comprehensive domain knowledge. In this paper, we propose DockerGen to containerize software packages automatically. DockerGen constructs a knowledge graph containing rich knowledge of building Docker images by analyzing nearly 220 thousand Dockerfiles. DockerGen exploits the knowledge graph to containerize the target software by creating a Dockerfile specifying the base image, dependencies, and the operation workflow. We evaluate DockerGen on 100 software packages of various categories. DockerGen achieves a 73% build success rate and a 59% configuration success rate. The experimental result indicates it is viable to automate software containerization based on a domain knowledge graph.

**Index Terms**—Docker, Dockerfile, containerization knowledge graph, software package, dependency

**I. INTRODUCTION**

Docker [1], the de-facto container technology [2], packages software systems and their dependencies into Docker images for continuous deployments and deliveries. A Docker image is usually constructed by executing a Dockerfile that contains a sequence of instructions specifying how to install and configure a software system [1]. At runtime, a Docker image instantiates one or more instances, i.e., Docker containers.

Creating a Docker image for a specific software requires comprehensive domain knowledge, including (1) language- and system-level dependencies, (2) compatibilities among software, libraries and operating systems (OSs), and (3) installation and configuration ways for the software and its dependencies. Without such knowledge, building a Docker image is time-consuming and error-prone [3].

In this paper, we propose a knowledge graph-based approach to containerizing software systems, namely DockerGen. DockerGen addresses the diversity of software systems and runtime environments by exploiting the domain knowledge acquired from existing Dockerfiles.

(1) DockerGen first extracts the knowledge of building Docker images from a large number of Dockerfiles (approximately 220 thousand so far) and constructs a Docker domain knowledge graph based on a meta-model.

(2) Given a target software, DockerGen finds all the dependencies for constructing a runtime environment, including a base image offering an OS, language- and system-level libraries, and tools of compatible versions, based on the knowledge graph. It determines the workflow of building the Docker image and generates a Dockerfile for the software.

We evaluate DockerGen by exploiting it to containerize 100 popular software packages of various types. It achieves a 73% build success rate and a 59% configuration success rate.

In summary, this work makes the following contributions.

- We extract knowledge from nearly 220 thousand Dockerfiles and build a Docker domain knowledge graph with over 900 thousand entities and nearly 2,900 thousand relations, e.g., software, OSs, Docker artifacts, and relations among them.
- We propose a knowledge-based approach for software containerization. The approach generates a Dockerfile for the target software by inferring the dependencies and determining the construction workflow.
- The experiment evaluates DockerGen on 100 software packages of various types. The experimental results reveal that it is promising to automate software containerization with domain knowledge.

**II. BACKGROUND AND MOTIVATION**

A Dockerfile specifies the workflow of building a Docker image with a set of domain-specific language (DSL) based instructions. The instructions declare a base image (`FROM`), execute shell commands (`RUN`) to install and configure software, and set environment variables (`ENV`). The example Dockerfile<sup>2</sup> (see Fig. 1) builds a Docker image constructing a PyTorch-based environment for developing machine learning applications and algorithms. The `FROM` instruction (line 1) specifies the base image offering the OS CentOS 7. Two `RUN` instructions install and configure some system-level tools and libraries (line 6-8 and `Ansconda3` (line 10-15), respectively. Finally, the command `conda install` in the third `RUN` instruction installs PyTorch (line 17). This Dockerfile specifies the workflow of building a PyTorch-centric Docker image. The system-level tools, libraries, and language-level dependency (i.e., `Ansconda3`) are prerequisites to or associated with the target software, i.e., PyTorch.



# AI-based Code Assistants





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

# 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

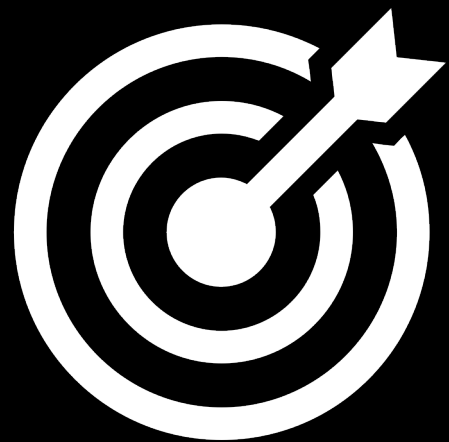
# Set Go environment variables
ENV GOROOT=/usr/local/go
ENV GOPATH=/go
ENV PATH=$GOROOT/bin:$GOPATH/bin:$PATH

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

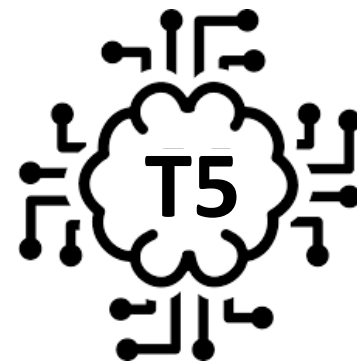
# Set the working directory
WORKDIR $GOPATH/src

# Define the entry point
CMD ["bash"]

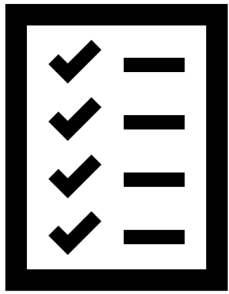
Regenerate response
```



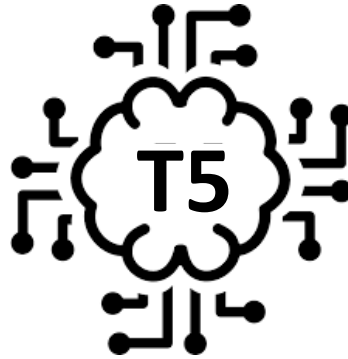
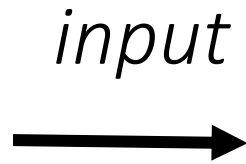
What about  
**state-of-the-art DL models**  
for code-related tasks?



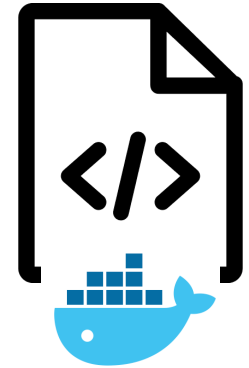
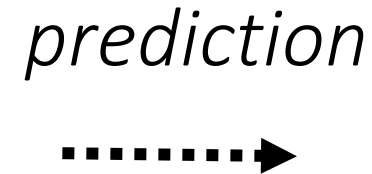
Phase 1:  
**Model Construction**



Natural language requirements

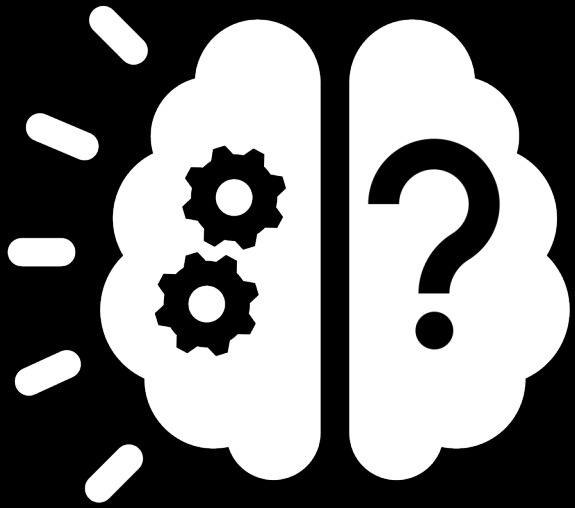


T5 Model



Generated Dockerfile

# 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

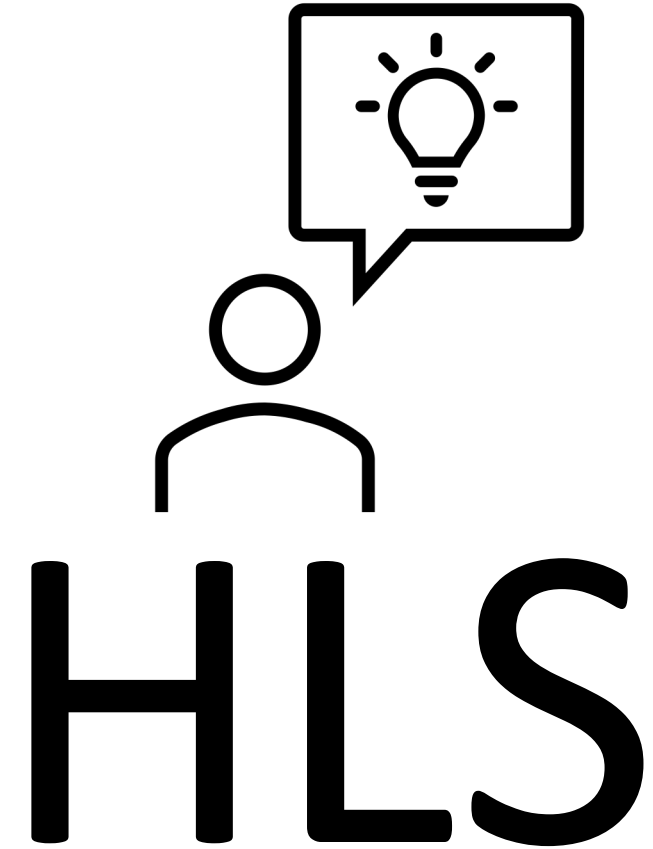
**Build ARGs:** FALSE

**LABEL:** TRUE

**EXPOSE for ports:** TRUE

**CMD:** TRUE

**ENTRYPOINT:** FALSE



High-Level Specification

**Operating System:** “alpine”

**Package Manager:** “apk”

**Package Requirements:** [“python3”]

**Download from Sources:** FALSE

**ENV variables:** FALSE

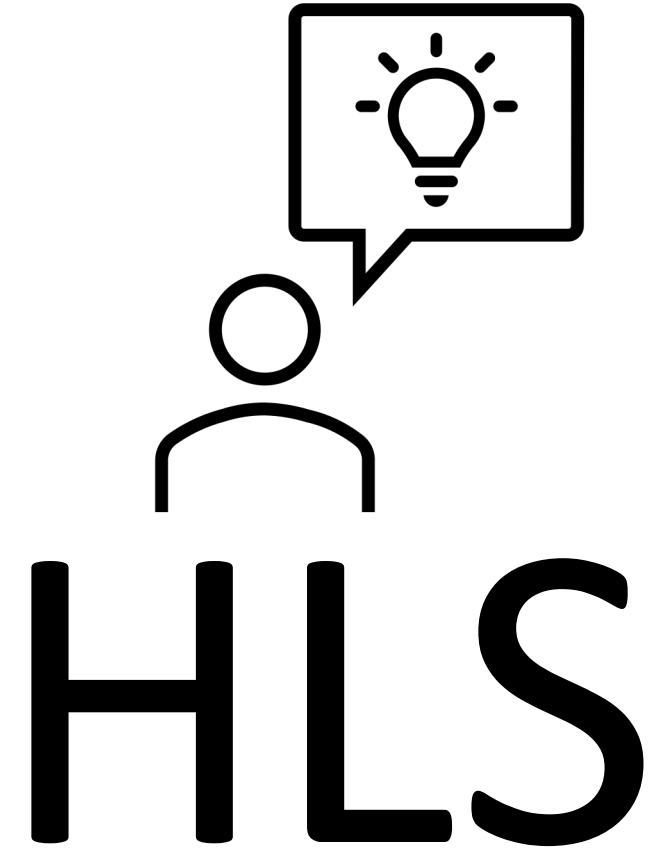
**Build ARGs:** FALSE

**LABEL:** TRUE

**EXPOSE for ports:** TRUE

**CMD:** TRUE

**ENTRYPOINT:** FALSE



High-Level Specification

**Operating System:** “alpine”

**Package Manager:** “apk”

**Package Requirements:** [“python3”]

**Download from Sources:** FALSE

**ENV variables:** FALSE

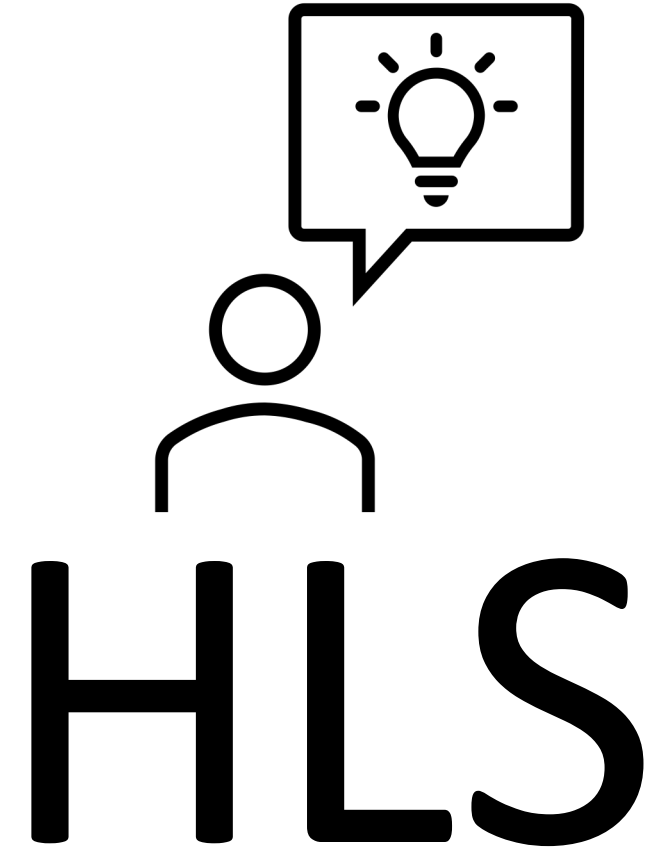
**Build ARGs:** FALSE

**LABEL:** TRUE

**EXPOSE for ports:** TRUE

**CMD:** TRUE

**ENTRYPOINT:** FALSE



High-Level Specification



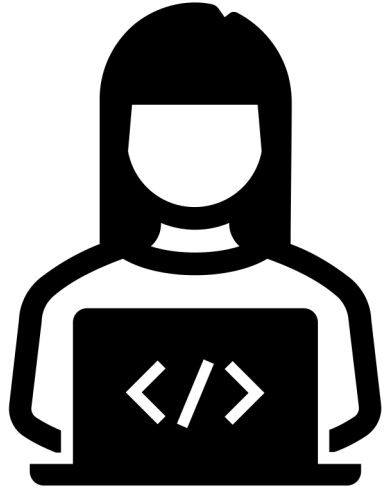
```
1 FROM alpine:3.17
2
3 LABEL maintainer="happy@container.net"
4
5 # Install python3
6 RUN apk add --update --no-cache python3
7
8 COPY . .
9
10 RUN pip install -r requirements.txt
11
12 WORKDIR /tiddlywiki
13
14 EXPOSE 8000
```

**OS:**  
"alpine3.17"

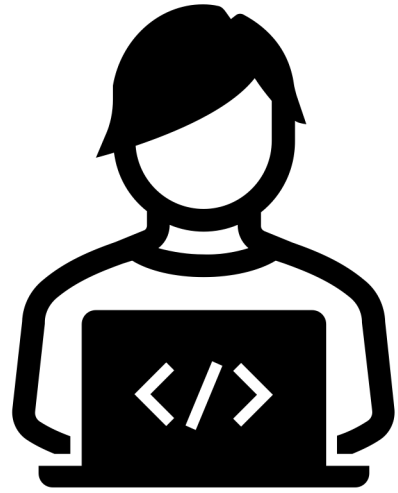
**Pkg. Manager:**  
"apk"

**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  
Department of Computing Science  
University of Alberta  
engkal@ualberta.ca

Abram Hindle  
Department of Computing Science  
University of Alberta  
ahindle@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

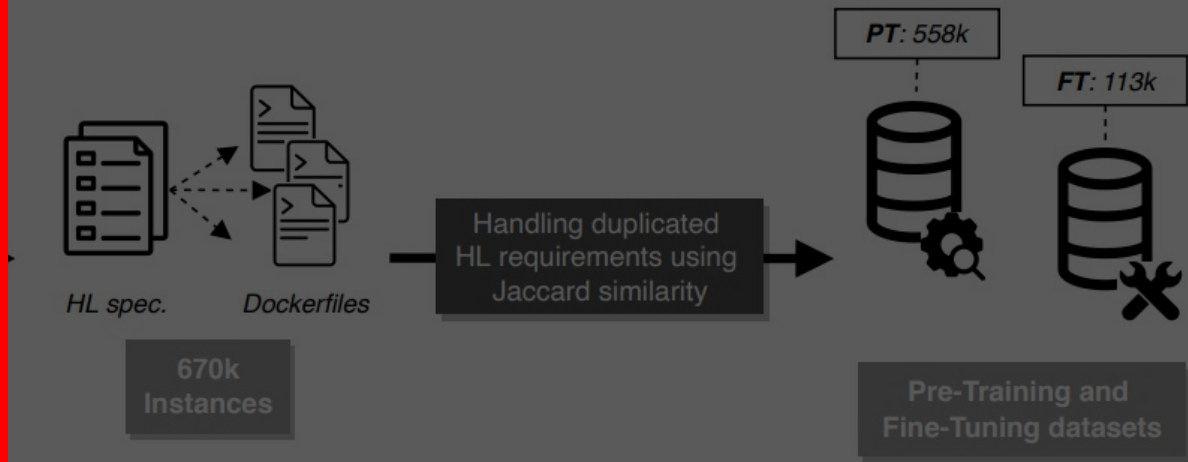
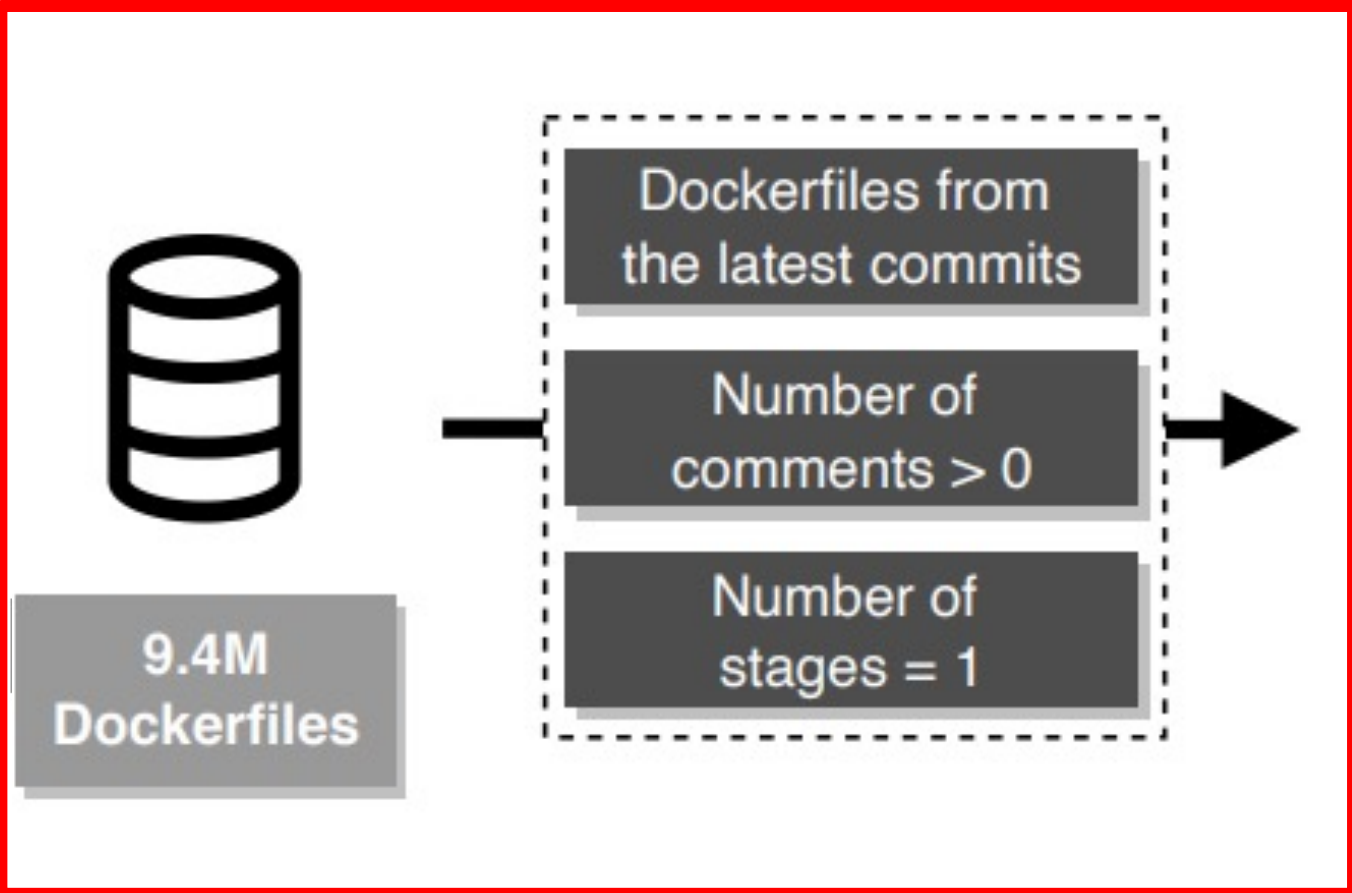
All of the above previous work focuses on Docker use in a project based perspective and involves mining Dockerfiles

# 9.4M

# unique Dockerfiles

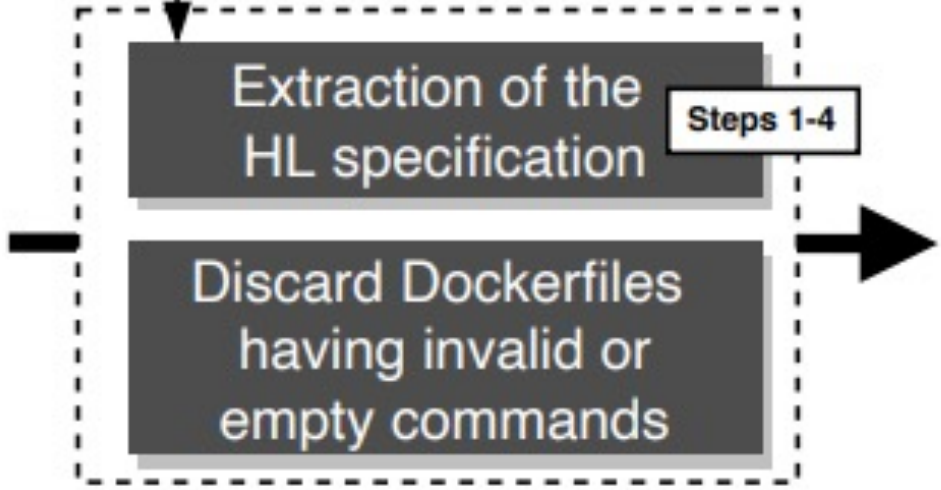
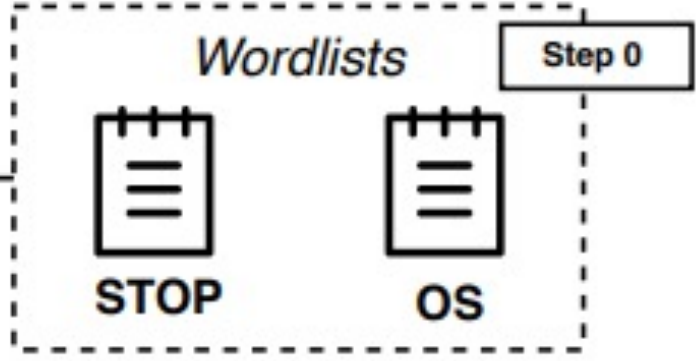
# from 2013 to 2020

# Dockerfile Dataset

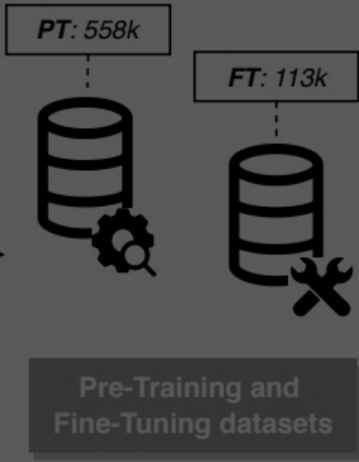


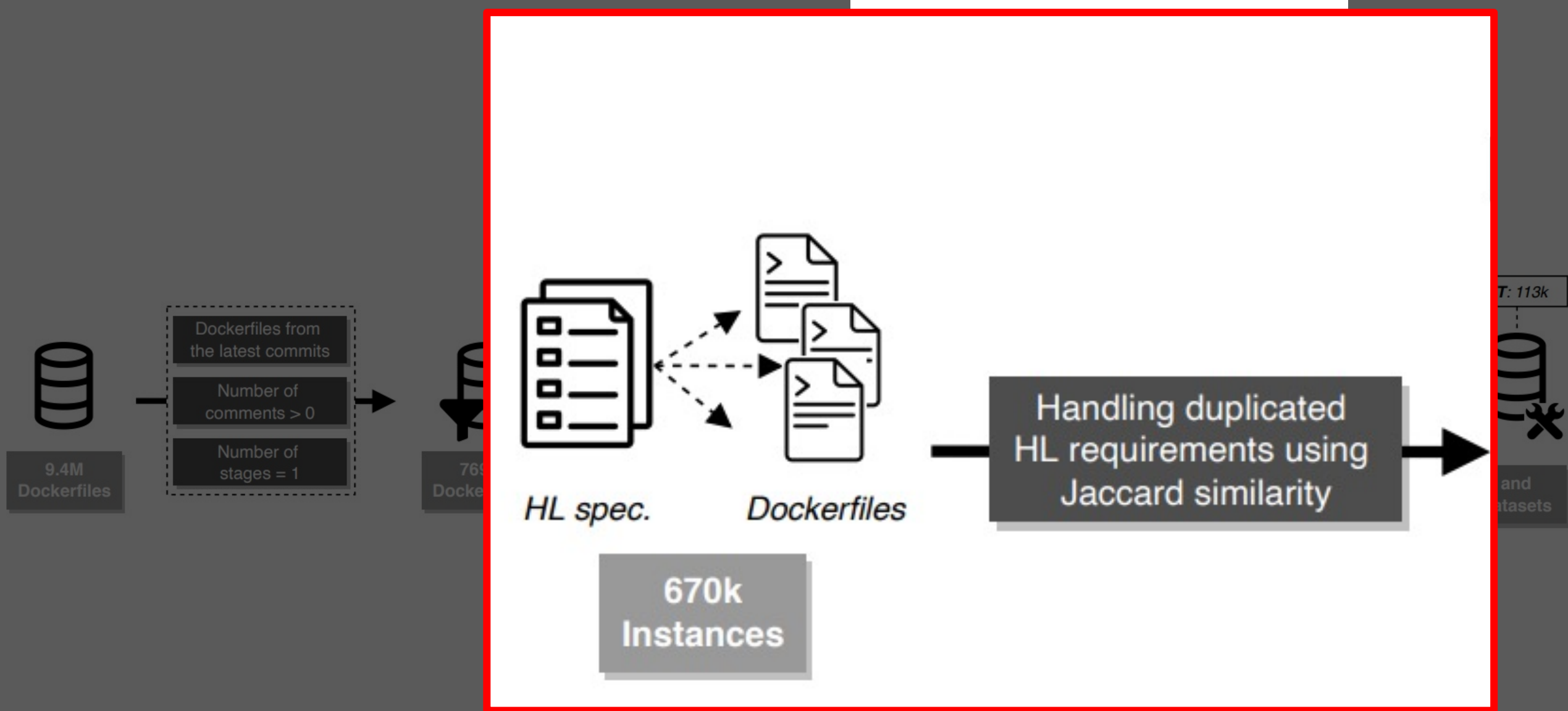
9.4M Dockerfiles

769k Dockerfiles

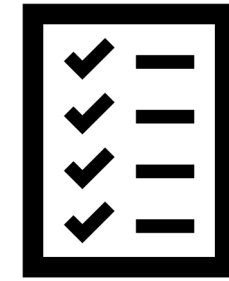


Handling duplicated requirements using Jaccard similarity





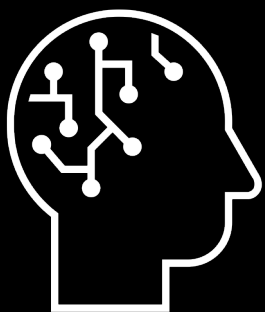
**100k** Model Tuning



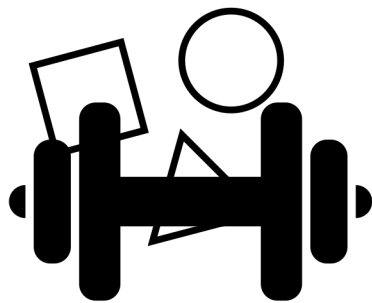
**11k** Test



Resulting Dataset



# T5 model construction



Pre-Training

**560k** instances

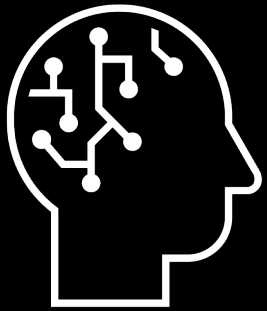


Fine-Tuning

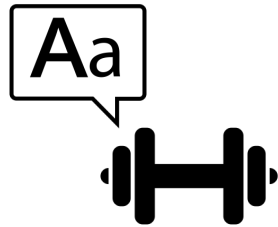
**90k** instances



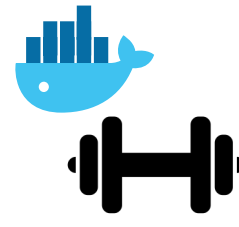
# 3 pre-training settings



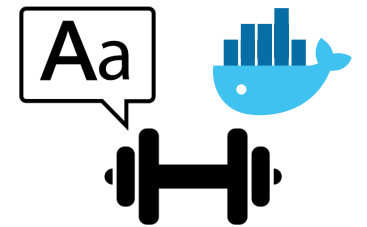
T5 model  
construction



English  
Only

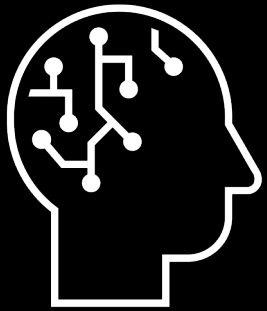


Dockerfile  
Only

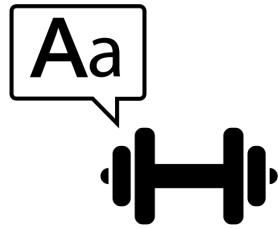


Dockerfile  
& English

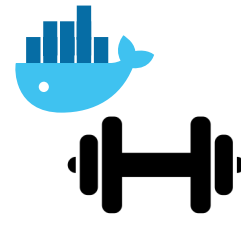
# Fine Tuning



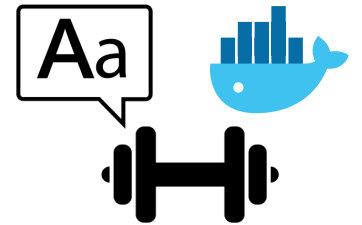
T5 model  
construction



English  
Only

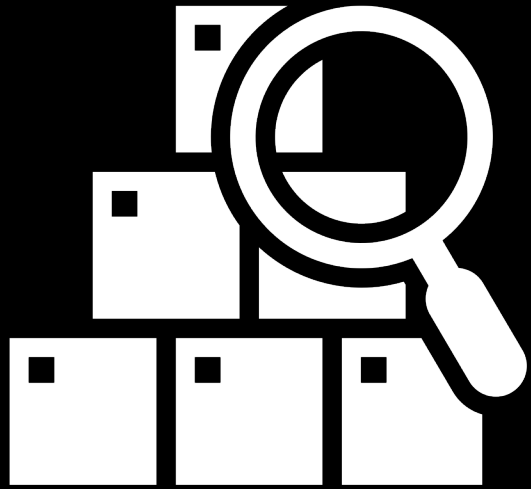


Dockerfile  
Only

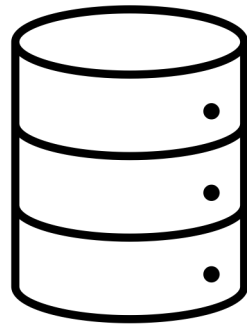


Dockerfile  
& English

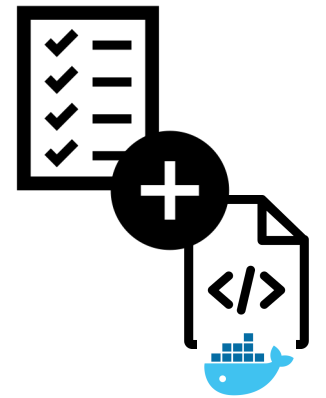
Phase 2:  
**Model evaluation**



**2** baselines



**90k**  
instances





# Elasticsearch

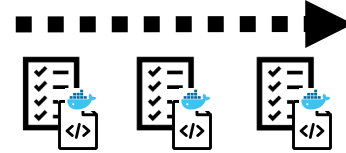


HLS



ES node

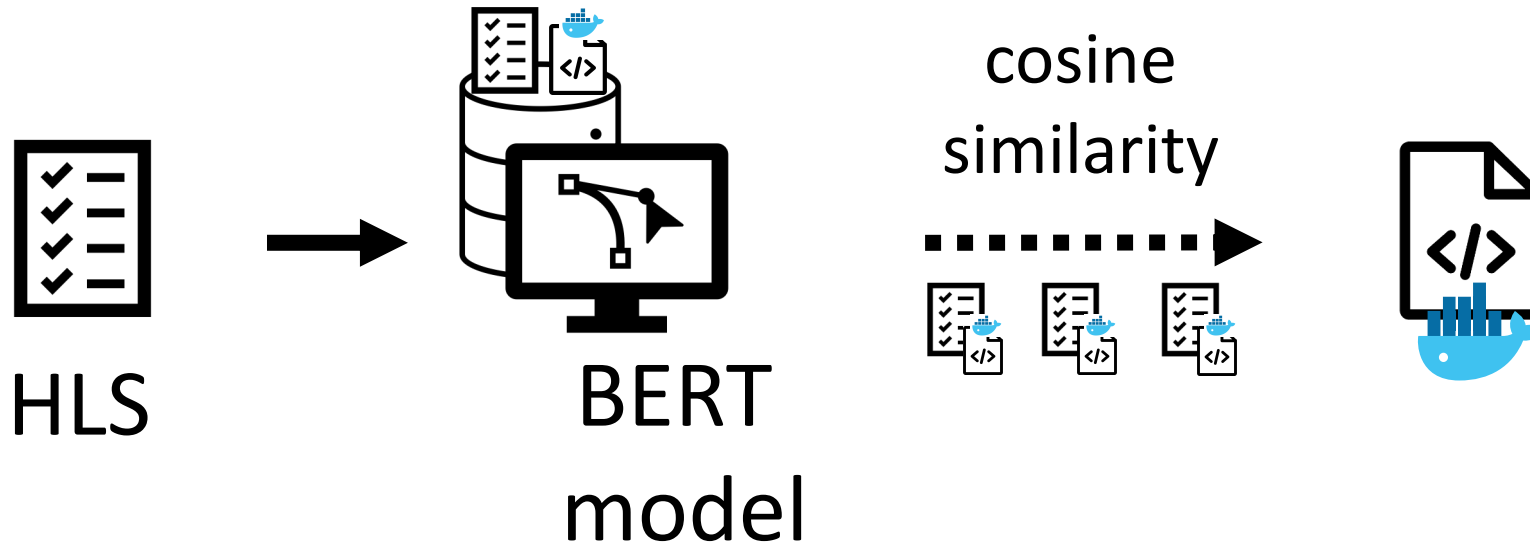
bool query



## IR-Baseline 1



# SentTransformers



IR-Baseline 1



**3** dimensions

**11k** Test

Evaluation

# RQ1

## Adherence to the input High-Level Specification



```
1 {
2   "os": "alpine",
3   "pkg_manager": "any",
4   "requirements": [
5     "python3"
6   ],
7   "uses_env": false,
8   "uses_arg": false,
9   "uses_label": true,
10  "uses_expose": true,
11  "uses_cmd": true,
12  "uses_entrypoint": false,
13  "download_of_external_packages": false
14 }
```

Input HLS

VS

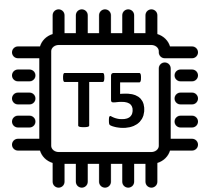
```
1 {
2   "os": "alpine",
3   "pkg_manager": "any",
4   "requirements": [
5     "python3"
6   ],
7   "uses_env": false,
8   "uses_arg": false,
9   "uses_label": true,
10  "uses_expose": true,
11  "uses_cmd": true,
12  "uses_entrypoint": false,
13  "download_of_external_packages": false
14 }
```

Generated/Retrieved HLS

+1  
+1  
+1  
+1  
+1  
+1  
+1  
+1

Metric:  
**Field-by-field  
match**



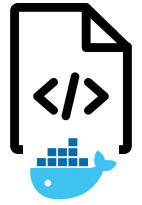


OS	~1.00	0.92	0.88
Pkg. Manager	0.98	1.00	1.00
Pkg. Requirements	0.87	0.88	0.76
Download from sources	0.82	0.84	0.52
ENV variables	0.89	0.81	0.17
Build ARGs	0.99	0.88	0.17
LABEL	~1.00	0.87	0.37
EXPOSE for ports	0.80	0.83	0.45
CMD	0.74	0.83	0.26
ENTRYPOINT	0.84	0.85	0.45

## Results for RQ1

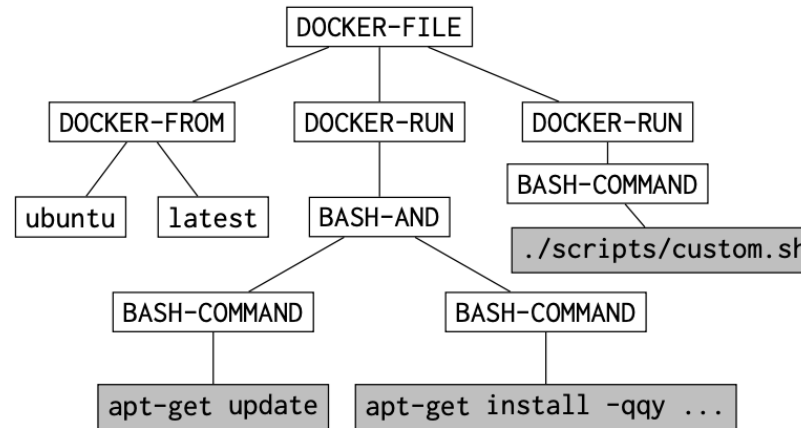
# RQ2

## Structural similarity between Dockerfiles



```
FROM ubuntu:latest
RUN apt-get update && \
    apt-get install -qqy ...
RUN ./scripts/custom.sh
```

Input Dockerfile

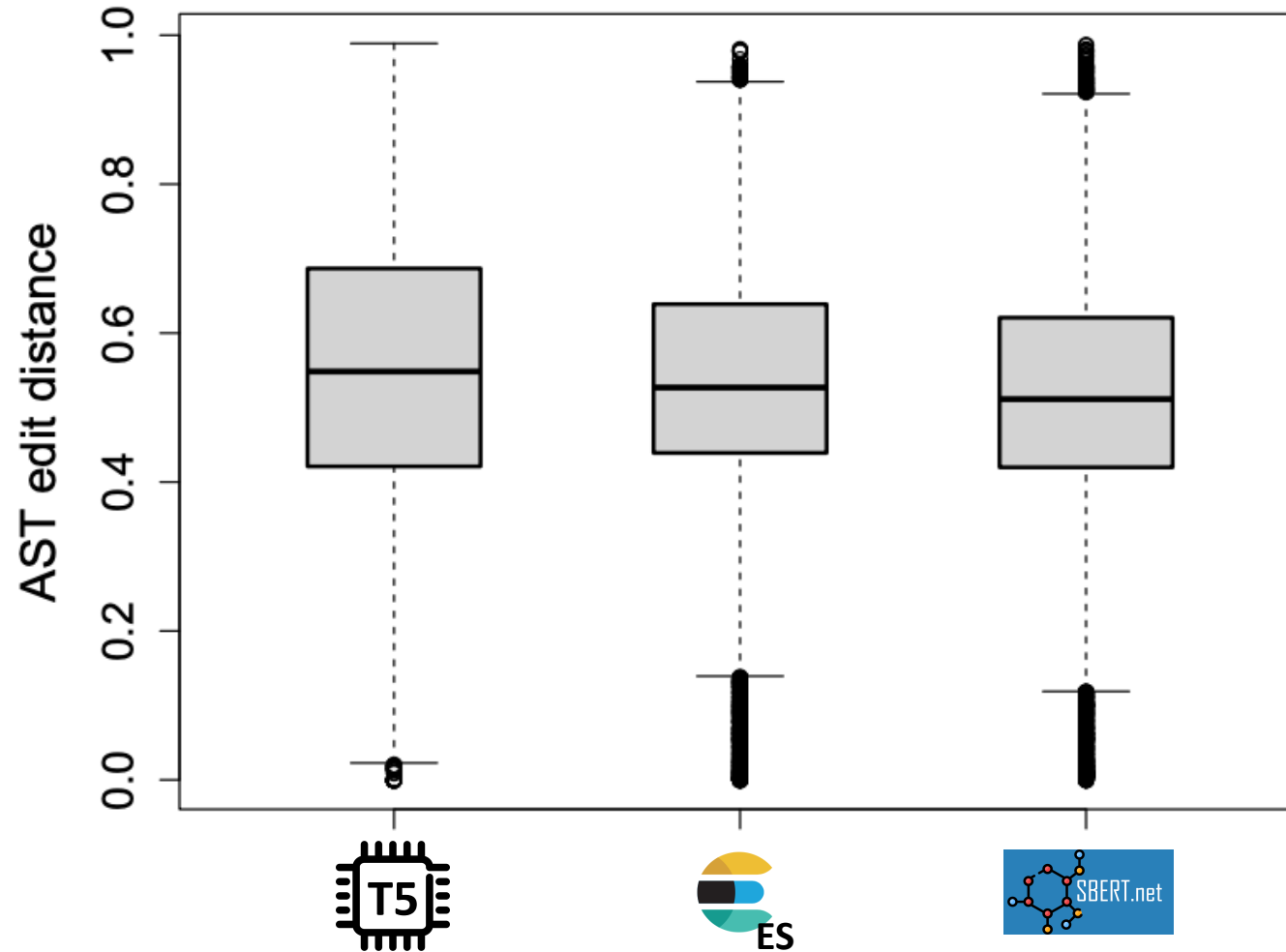


Parsed AST  
(Henkel et. al 2020)

Metric:  
**AST**  
**edit distance**

# RQ2

## Structural similarity between Dockerfiles

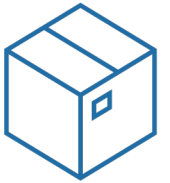


The lower is better

# Results for RQ2

# RQ3

## Similarity between Docker images



Input Dockerfile

```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT [ "/bin/server" ]
```

build  
→

Docker Image A

```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT [ "/bin/server" ]
```

Gen./Retr. Dockerfile

```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
ENTRYPOINT [ "/bin/server" ]
```

build  
→

Docker Image B

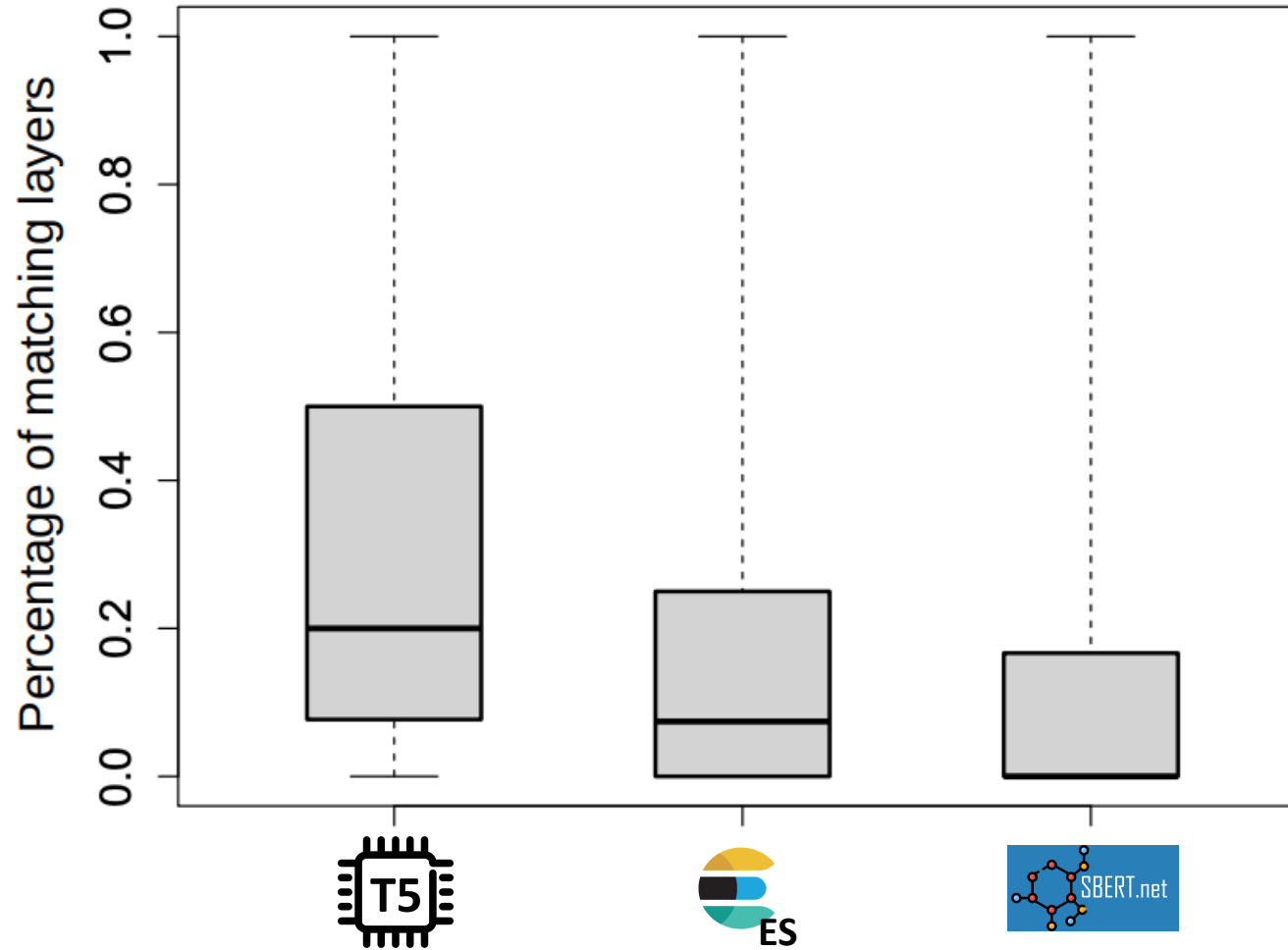
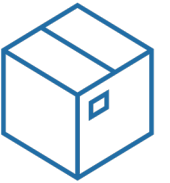
```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
ENTRYPOINT [ "/bin/server" ]
```

Matching  
SHAs  
↙

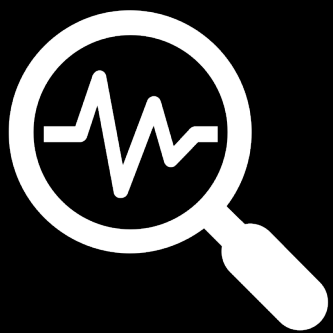
Metric:  
**Percentage of  
matching layers**

# RQ3

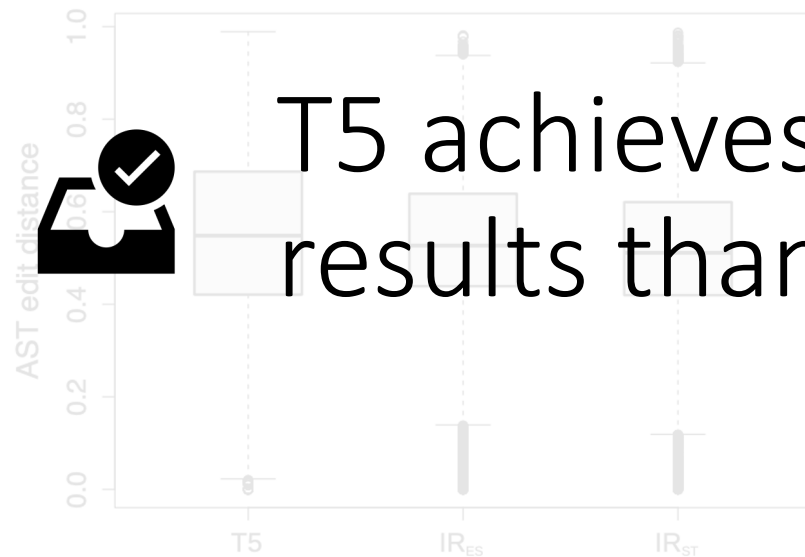
## Similarity between Docker images



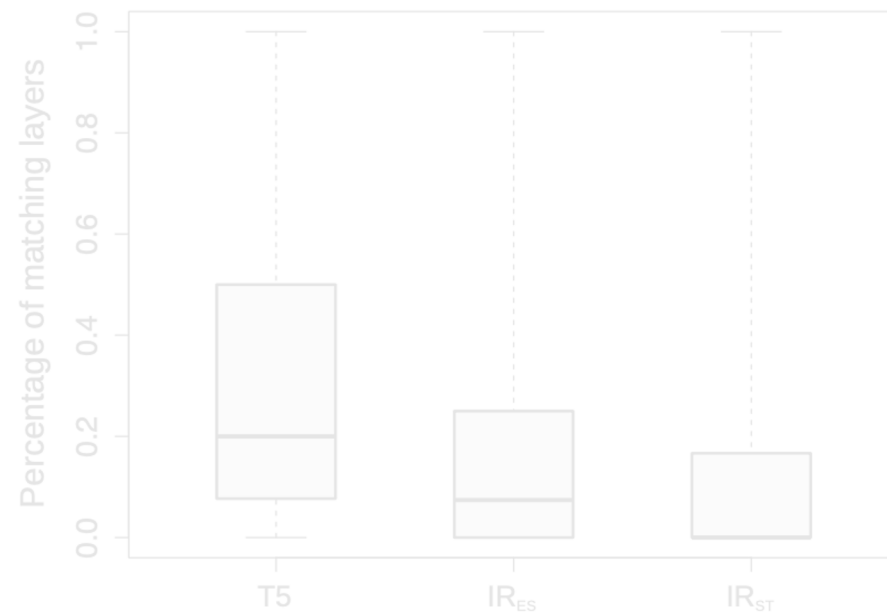
Results for RQ3



# Summary



T5 achieves slightly better results than IR ...

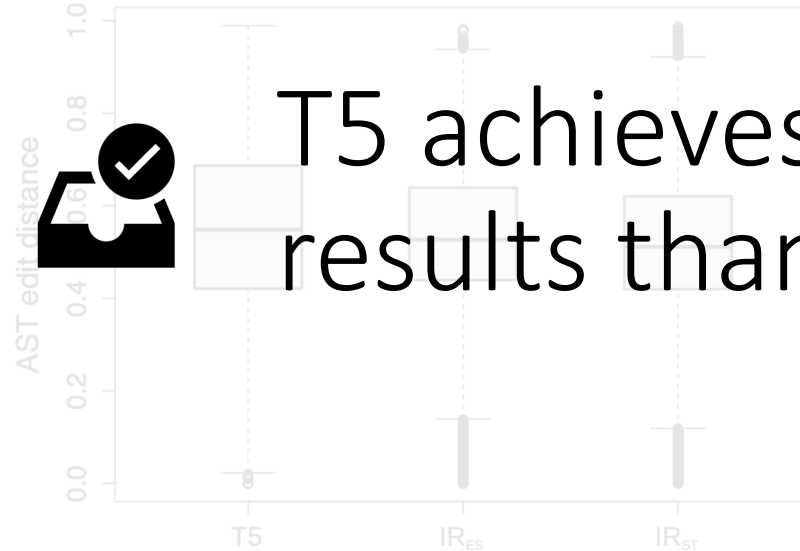




# Summary



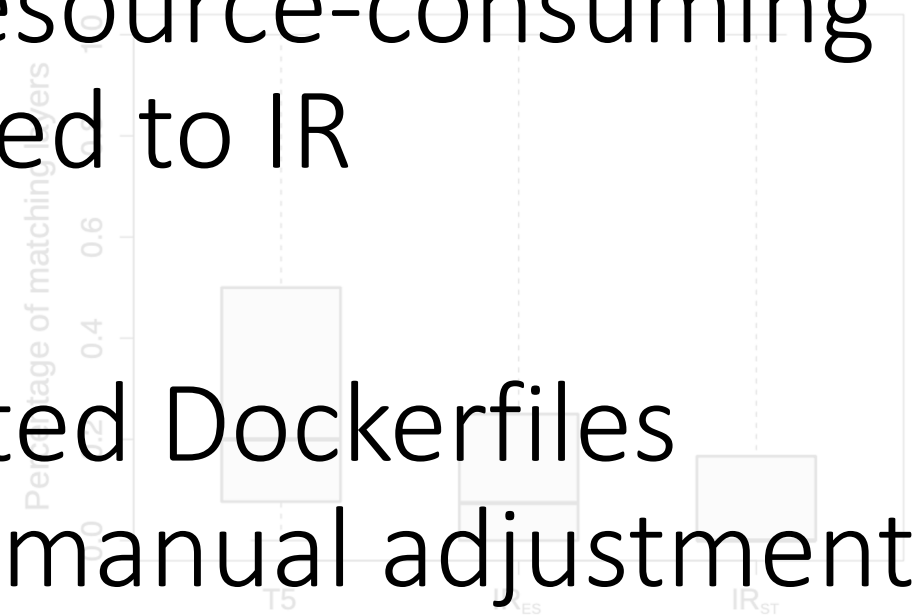
T5 achieves slightly better results than IR ...



More resource-consuming compared to IR

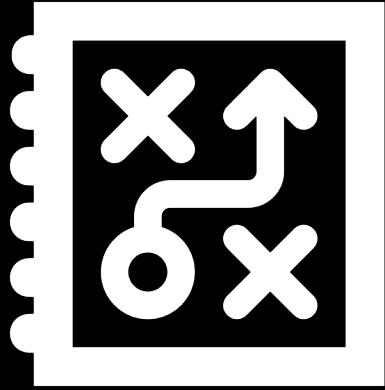


Generated Dockerfiles require manual adjustments



**What we have learned?**

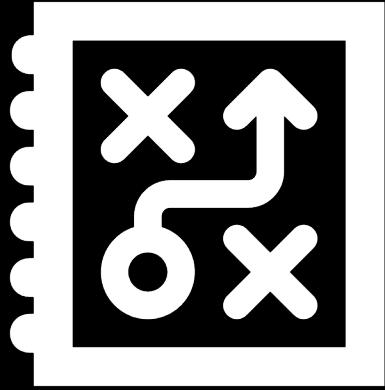




# Challenge #1



Not enough training instances



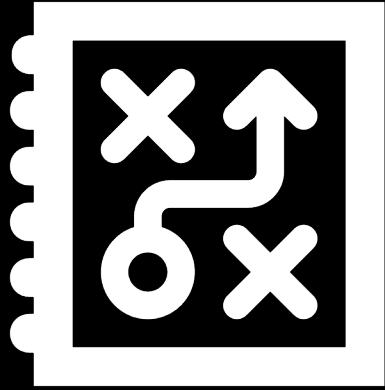
# Challenge #1



Not enough training instances



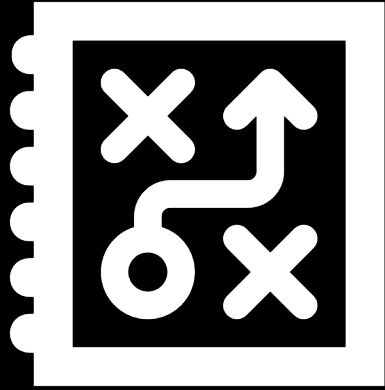
**Data augmentation**



## Challenge #2



A different training procedure  
must be used



## Challenge #2



A different training procedure  
must be used

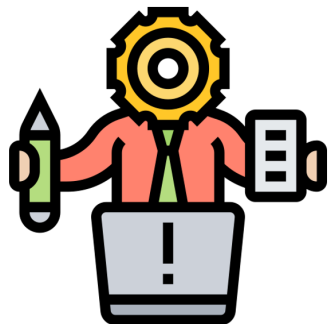


**Different stopping  
criterion**

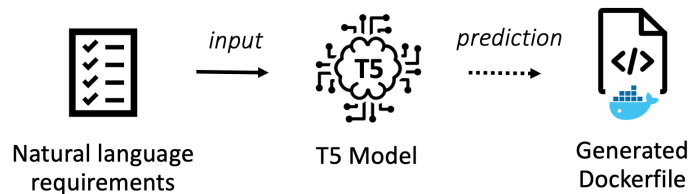


**Dockerfile abstractions**

# Summary



Writing Dockerfiles is **challenging**



Dockerfile Generation via T5



**3** dimensions

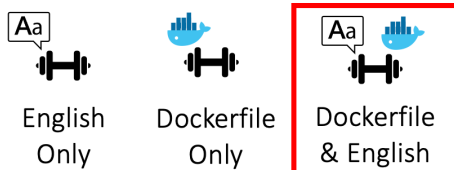
**11k** test

Evaluation



T5 model construction

Fine Tuning



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!**