FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Improving the Developer Experience of Dockerfiles

João Pereira da Silva Matos

WORKING VERSION

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Filipe Correia

January 2, 2023

Improving the Developer Experience of Dockerfiles

João Pereira da Silva Matos

Mestrado em Engenharia Informática e Computação

January 2, 2023

Contents

List of Tables

Chapter 1

Repairing and Generating Dockerfiles (title is WIP)

This section covers existing literature and tools that are related to several components of the Dockerfile development experience and Docker in general. Section 1.1 goes over the background that is required to understand this paper. Section 1.2 goes over ways to accelerate Docker builds. Section 1.3 goes over works that cover the generation of Dockerfiles. Section 1.4 covers smells in Dockerfiles. Section 1.5 talks about good practices that should be taken into account when writing Dockerfiles. Section 1.6 tackles the topic of security in Dockerfiles. Section 1.7 covers the detection and repair of faults in Dockerfiles. Section 1.8 covers the detection and removal of bloat in Dockerfiles. Section 1.9 is about testing Dockerfiles. Section 1.10 concludes with a general discussion.

1.1 Background

...

1.2 Speeding up Docker builds

Building Docker images can take a considerable amount of time, especially when a large amount of files have to be fetched from the internet [11]. Therefore, we looked for ways to reduce the amount of time consumed by this activity. Our findings are summarized in Table 1.1.

Name	**Speedup**	**Limitations**	**Implementation Complexity**
A Code Injection Method for Rapid Docker Image Building [28]	Up to 100000x faster	Can only be used with interpreted languages, limited to modifications in the source code	High
FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container [16]	Up to 10x faster	Limited to network activity	High
Slacker: Fast Distribution with Lazy Docker Containers [11]	Up to 20x faster	Limited to network activity	High
Docker Buildx [4]	Unknown	Unknown	Low-Medium

Table 1.1: Works about speeding up Docker builds

Wang et al. [28] propose a technique that bypasses typical building procedures by injecting the code modifications directly in an image. The results are very promising. However, due to the nature of approach, it can only be used with interpreted languages and will not accelerate builds related to modifications in development artifacts that are not source code.

Huang et al. [16] and Harter et al. [11] address the network bottleneck in different ways. The former caches files locally and intercepts Docker's network requests in order to serve files that have been stored locally. The latter proposes a new storage driver that lazily fetches files from the network. Both show promising results and do not address other inefficiencies in the Docker building process.

The works described so far interfere with the normal Docker build procedures, making them harder to implement. Another solution is offered by the Docker development team, Buildx [4]. Buildx makes use of a newer backend, BuildKit [3], which brings many features that can potentially accelerate docker builds. However, to our knowledge, an apples to apples time comparison has not been made. Implementing this in Dockerlive wouldn't be very hard but because the output of the buildx command is different, some modifications would still be required.

1.3 Dockerfile Generation

Writing Dockerfiles is not an easy process [26]. Therefore, having a way to generate a Dockerfile for a given project can be very useful. In this section, we looked for works that showcased ways to accomplish this. Our findings are summarized in Table 1.2.

Name	Successful generation rate	Limitations	Implementation Complexity
Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations [27]	Unknown	An Open API Spec is required	Medium
Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph [33]	Up to 80%	A vast knowledge graph is required, focused on Singularity	High
Container-Based Module Isolation for Cloud Services [17]	Unknown	Requires the use of templates to generate the files	Medium-High
DockerGen: A Knowledge Graph based Approach for Software Containerization [31]	Up to 73%	A vast knowledge graph is required	High
DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets [15]	Up to 30%	Limited to Python, requires a knowledge base	High
ExploitWP2Docker: a Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges [9]	Up to 39%	Limited to security testing scenarios	Low-Medium
MAKING CONTAINERS EASIER WITH HPC CONTAINER MAKER [22]	Unknown	Requires Python code to generate the files	Medium-High

Table 1.2: Works about generating Dockerfiles

Zhong et al. [33], Ye et al. [31] and Horton et al. [15] present solutions that require pre-existing knowledge bases in order to generate the files, making them hard to implement in a project like ours, which is not completely focused on file generation. DockerizeMe is also limited to Python environments, while Burner is more focused on Singularity [2], a containerization tool similar to Docker but focused on HPC (High Performance Computing).

Caturano et al. [9] propose a tool that uses Docker to generate security testing environments from exploit descriptions. Sorgalla et al. [27]'s work can generate Dockerfiles from models, which are generated from Open API Specifications (like Swagger [5]). Kehrer et al. [17] use Apache FreeMarker [1] to generate Dockerfiles from templates. McMillan et al. [22] offer a tool that allows developer to use Python code to define the information required to generate Dockerfiles.

All these works show varying degrees of success and some of them are not even focused on the Dockerfile generation aspect.

1.4 Dockerfile Smells

Smells are commonly found in Dockerfiles [29], making it important to create ways of detecting and, if possible, remove them. This section covers works related to this. Our findings are summarized in Table 1.3.

Name	**Smells**	**Findings**	**Possible repair implementation complexity**
An Empirical Case Study on the Temporary File Smell in Dockerfiles [21]	Temporary File	The smell is quite common and can be divided into 4 different types, 3 of which can be detected through the proposed methods	Medium
An empirical study on self-admitted technical debt in Dockerfiles [7]	SATD	Shows that a type of smell appears in Dockerfiles and can be divided into several classes and subclasses	High
Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study [29]	All	Smells appear commonly in projects and the frequency with which they appear varies according to several metrics	High
Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods [30]	Temporary File	It's possible to detect the smell with high accuracy using static and dynamic analysis	Medium

Table 1.3: Works about Dockerfile smells

Lu et al. [21] and Xu et al. [30] have focused on the temporary file smell and propose ways to detect this smell. A repair to deal with this smell could be implemented using the information provided by these works.

Azuma et al. [7] focus on a variation of smells they call SATD (self-admitted technical debt) which can be detected in comments written in the Dockerfiles. Due to the nature of these SATDs, implementing repairs to eliminate them could be very complicated.

Wu et al. [29] analyzed a large amount of open-source projects and found that Dockerfile smells are very common and their frequency changes according to several factors like the programming language used by the project or the project's age. Due to the variety of smells covered by this study implementing repairs to deal with all of them would be difficult.

1.5 Dockerfile Good Practices

To prevent the creation of smells like the ones mentioned in Section 1.4, a developer should follow good practices. This section goes over works that cover these practices. Our findings are summarized in Table 1.4.

Name	**Practices**	**Limitations**	**Practices that could be implemented**
Learning from, Understanding, and Supporting DevOps Artifacts for Docker [13]	Related to usage of package managers and command line utilities (gold rules)	No inherent repair functionality	All of them
Security Misconfigurations Detection and Repair in Dockerfile [24]	Practices that make an image more secure	Limited to improving security (although these practices have other benefits)	Most of them
Ten simple rules for writing Dockerfiles for reproducible data science [23]	Rules 3,4,5 and 9 are applicable to every scenario	Focused on data science	Rules 5 and 9

Table 1.4: Works about Dockerfile good practices

Henkel et al. [13] mined rules from Dockerfiles created by experts, allowing them to create a set of "gold rules", a set of patterns that often appear in Dockerfiles written by these experts. All of these "gold rules" could be implemented as repairs.

Prinetto et al. [24] looked for flaws in Dockerfiles that could lead to vulnerabilities in a system. As part of that work they list a set of practices developers should follow to improve a Docker image's security. Most of the practices listed could be implemented as repair, although some of them would be too complex to implement.

Nust et al. [23] propose a list of 10 rules developers should follow when writing Dockerfiles for data science environments. Some of these rules are applicable to other scenarios and 2 of those could be implemented as repairs.

1.6 Dockerfile Security

Nowadays, security is a topic that is heavily discussed and deserves a great amount of attention from developers. However, security problems are still commonly found in Dockerfiles [10] and many developers do not have the knowledge required to evaluate how vulnerable their containers are [32]. For these reasons, it's important to study Docker containers from a security perspective, which is what this section focuses on. Our findings are summarized in Table 1.5.

Name	**Findings**	**Implementation notes**
DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning [10]	DAVS can detect more vulnerabilities than competing scanners	It should be possible to repair some of the mentioned vulnerabilities, although it would be easier to use existing scanners
Investigating the inner workings of container image vulnerability scanners [32]	Many scanners use the same methods to detect vulnerabilities, which have limitations	Using one of these scanners could be useful
Outdated software in container images [20]	Having outdated software in containers brings security problems and there are limitations to what current scanners can detect, new detection method is proposed	It should be possible to implement some repair to try to address this situation
Security Analysis of Code Bloat in Machine Learning Systems [6]	Removing bloat from containers used in machine learning environments can considerably improve security	It should be possible to implement some repairs that reduce bloat
Security Misconfigurations Detection and Repair in Dockerfile [24]	Security problems are common in containers, a way to repair them is proposed	It might be possible to implement the proposed technique to repair the problems

Table 1.5: Works about Dockerfile security

Doan et al. [10] propose DAVS (Dockerfile analysis-based vulnerability scanning), a tool that can detect potentially vulnerable files in containers. This approach allows them to detect more vulnerabilities than current scanners, which, according to Zarei et al. [32], rely on information provided by distribution package managers. This information can be manipulated and, in some cases, may not even be available, which prevents scanners from detecting vulnerabilities.

Ahmed et al. [6] used Cimplifier [25] to debloat containers used in machine learning environments and found the amount of vulnerabilities present in those containers was significantly reduced.

Linnalampi et al. [20] found that having outdated software introduces vulnerabilities in containers and propose a new method to detect vulnerabilities by analyzing the binaries present in containers to detect the software versions that are in use. This approach would address some of the limitations of current scanning techniques.

Prinetto et al. [24] found that security problems are common and propose a way to repair them by processing the Dockerfile to obtain the abstract syntax tree, find the vulnerabilities and modify the tree before reconverting into a file that is no longer vulnerable.

Implementing repairs that address most of the problems and vulnerabilities found by these works should be possible. It may even be possible to use some of the proposed approaches.

1.7 Dockerfile Repair

Like the previous sections have shown, the average Dockerfile has several problems and it can be difficult for a developer to figure out how to deal with those issues in an optimal way. This makes it important to create tools that can assist developers in the repair process. This section goes over works that do that (although other sections also discuss works that perform repairs that are related to more specific scenarios). Our findings are summarized in Table 1.6.

Name	Performed repair	Limitations	Implementation Complexity
Latest Image Recommendation Method for Automatic Base Image Update in Dockerfile [18]	Base image update	Does not cover other parts of the Dockerfile	Medium
Learning from, Understanding, and Supporting DevOps Artifacts for Docker [13]	Enforcing the gold rules by detecting violations	Does not perform the repair, only helps detect places where they should be performed	Medium
RUDSEA: recommending updates of Dockerfiles via software environment analysis [12]	Updates portions of the source code which are tied to values in the source code	Does not cover parts other parts of the Dockerfile	Medium-High
Shipwright: A Human-in-the-Loop System for Dockerfile Repair [14]	Repairs that deal with some functional problems	Some of the repairs listed can only be applied to some projects	Medium
Supporting microservices deployment in a safer way: a static analysis and automated rewriting approach [8]	Reducing the number of layers in the image to take advantage of layer caching	Small number of repairs	Low-Medium

Table 1.6: Works about Dockerfile repair

Kitajima et al. [18] focused on updating a container's base image by analyzing the available tags, while Hassan et al. [12] focused on portions of the Dockerfile which are tied to values in the source code.

Henkel et al. [13] offers a way to detect violations of the gold rules they obtained but don't automate the repair of said violations. Henkel et al. [14] also proposes a different approach for automating repairs, although most of the repairs listed here are specific to certain programming languages or package managers.

Benni et al. [8] describe a way to reduce the number of layers in Dockerfiles in order to take advantage of layer caching.

Implementing the repairs mentioned in this section should be possible, although these implementations would have varying degrees of complexity.

1.8 Dockerfile Bloat

1.9 Dockerfile Testing

1.10 General Discussion

Ksontini et al. [19] focused on refactorings (a concept which is closely tied to smells) and found that developers' main motivations for performing refactorings were tied to maintainability and image size among others. Implementing some of these refactorings as repairs would be useful, although implementing all of them would be challenging.

References

[1] Apache FreeMarker.

[2] Apptainer.

[3] BuildKit.

[4] Docker buildx.

[5] Swagger.

[6] Fahmi Abdulqadir Ahmed and Dyako Fatih. Security Analysis of Code Bloat in Machine Learning Systems.

[7] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. An empirical study on self-admitted technical debt in Dockerfiles. 27(2):49.

[8] Benjamin Benni, Sébastien Mosser, Philippe Collet, and Michel Riveill. Supporting microservices deployment in a safer way: A static analysis and automated rewriting approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1706–1715. Association for Computing Machinery.

[9] Francesco Caturano, Nicola d' Ambrosio, Gaetano Perrone, Luigi Previdente, and Simon Pietro Romano. ExploitWP2Docker: A Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges. In *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–7.

[10] Thien-Phuc Doan and Souhwan Jung. DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning. 72(1):1699–1711.

[11] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers.

[12] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. RUDSEA: Recommending updates of Dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 796–801. Association for Computing Machinery.

[13] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 38–49.

[14] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d' Amorim, and Thomas Reps. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1148–1160.

[15] Eric Horton and Chris Parnin. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 328–338.

[16] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 28–37.

[17] Stefan Kehrer, Florian Riebandt, and Wolfgang Blochinger. Container-Based Module Isolation for Cloud Services. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 177–17709.

[18] Shinya Kitajima and Atsuji Sekiguchi. Latest Image Recommendation Method for Automatic Base Image Update in Dockerfile. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari, editors, *Service-Oriented Computing*, Lecture Notes in Computer Science, pages 547–562. Springer International Publishing.

[19] Emna Ksontini and Marouane Kessentini. Refactorings and Technical Debt for Docker Projects.

[20] Markus Linnalampi. Outdated software in container images.

[21] Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. An Empirical Case Study on the Temporary File Smell in Dockerfiles. 7:63650–63659.

[22] Scott McMillan. MAKING CONTAINERS EASIER WITH HPC CONTAINER MAKER. page 47.

[23] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J. Eglen, Tim Head, Tony Hirst, and Benjamin D. Evans. Ten simple rules for writing Dockerfiles for reproducible data science. 16(11):e1008316.

[24] Paolo Ernesto Prinetto, Dott Riccardo Bortolameotti, and Giuseppe Massaro. Security Misconfigurations Detection and Repair in Dockerfile. page 78.

[25] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 476–486. Association for Computing Machinery.

[26] David Reis, Bruno Piedade, Filipe F. Correia, João Pedro Dias, and Ademar Aguiar. Developing Docker and Docker-Compose Specifications: A Developers' Survey. 10:2318–2329.

[27] Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations: The Case for Microservice Architecture. 2(6):459.

[28] Yujing Wang and Qinyang Bao. A Code Injection Method for Rapid Docker Image Building.

[29] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. 8:34127–34139.

[30] Jiwei Xu, Yuewen Wu, Zhigang Lu, and Tao Wang. Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 185–190.

[31] Hongjie Ye, Jiahong Zhou, Wei Chen, Jiaxin Zhu, Guoquan Wu, and Jun Wei. DockerGen: A Knowledge Graph based Approach for Software Containerization. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 986–991.

[32] Mehdi Zarei. Investigating the inner workings of container image vulnerability scanners. page 96.

[33] Shuaihao Zhong, Duoqiang Wang, Wei Li, Feng Lu, and Hai Jin. Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph. 2022:e4592428.