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 8, 2023

Improving the Developer Experience of Dockerfiles

João Pereira da Silva Matos

Mestrado em Engenharia Informática e Computação

January 8, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Repairing and Generating Dockerfiles (title is WIP)

Now that we've introduced the problem this thesis revolves around it's time to go over the state of the art. This section covers existing literature and tools that are related to several components of the Dockerfile development experience and Docker in general.

Estes parágrafos deviam sobretudo fazer uma introdução ao capítulo, e ligação com o capítulo anterior. Listar as subsecções e dizer o que elas contêem não tem muito interesse, estas secções podem é explicar melhor de que forma é que as várias secções se relacionam. Mesmo assim, neste caso, como são muitas, fica um pouco maçudo, e acho que não acrescenta muito para o leitor.

1.1 Background

...

1.2 Goals and methodology

Our end goal is to help developers write Dockerfiles. To do this, we first need to know what challenges they face when writing Dockerfiles and how the quality of the Dockerfiles is affected by these challenges. Furthermore, we need to analyze current solutions that address these issues in order to build upon them. With this in mind, we came up with the following research questions:

- **RQ1**: What are the challenges that developers face in the development of Dockerfiles?
- **RQ2**: How have these challenges been addressed so far?

To answer these questions we tried to find as much information as possible about Docker and Dockerfiles. Furthermore, we also tried to find works that analyzed a developer's subjective

experience when performing development tasks. We ended up using the following queries to look for information in Google Scholar [1]:

- dockerfile generation
- distroless
- docker build
- dockerfile
- dockerfile creation
- dockerfile generator
- dockerfile repair
- docker repair
- dockerfile readability
- dockerfile evaluation
- dockerfile analysis
- docker bloat
- docker build time
- dockerfile practices
- programming experience

These queries gave us plenty of results, which had to be filtered to reach an amount that could be reasonably analyzed in the amount of time that we were given. We focused on works from the last few years that focused (at least partially) on Docker or Dockerfiles (instead of merely using Docker as a tool due to its convenience). We looked at the title and abstracts of the results to perform our selection and ended up with around 50 works to analyze. Most of these were analyzed, although some had to be excluded due to either time constraints or difficulty accessing the work.

1.3 Challenges in the Development of Dockerfiles

After looking at the literature, we identified several aspects of the Dockerfile development experience that could be considered challenging. This includes the following aspects:

[1] Google Scholar, https://scholar.google.com/

- The amount of time required to build an image from a Dockerfile
- The amount of vulnerabilities present in Dockerfiles (and the corresponding images)
- The amount of bloat present in Docker images
- The reliance on trial and error to test Dockerfiles
- The amount of smells contained in Dockerfiles
- The amount of Dockerfiles that do not follow best practices

With these issues in mind, this chapter contains sections that address each of these problems. Additionally, there are also sections dedicated to repairing and generating Dockerfiles since those are features that our tool will provide and could improve the Dockerfile development experience.

Sugiro que incluas aqui um pouco sobre os objetivos desta revisão de literatura. As RQs da revisão de literatura apareceriam aqui. Além dessas perguntas, fala também um pouco do processo que seguiste para lhes tentar dar resposta, incluindo onde pesquisaste, que queries usaste, quais foram os teus critérios de inclusão/exclusão de resultados, etc.

Sobre as secções que se seguem: cada uma delas endereça um "challenge" diferente no desenvolvimento de dockerfiles, certo? Antes de falares de cada um deste tópicos seria bom ficar mais claro de onde/porque é que aqui aparecem. Talvez a melhor abordagem seja incluires uma nova secção antes das que vêm a seguir, chamada "Challenges in the development of Dockerfiles", ou algo semelhante, que olhe para a literatura e identifique o top dos challenges mais relevantes.

1.4 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 [6]. Therefore, we looked for approaches to reduce the amount of time consumed by this activity. Our findings are summarized in Table 1.1.

ways -> approaches

Dependendo do que disseres acima sobre quais foram as queries que usaste, pode não fazer sentido aqui dizeres que procuraste especificamente por abordagens de aceleração do build. A ver...

A não ser que já o tenhas dito antes (a secção de "Goals ..." acima pode ser um sítio melhor até!) era importante começar esta secção explicando qual é o interesse/importância/relevância para o teu trabalho em fazer esta análise.

Name	**Speedup**	**Limitations**	**Interferes with normal Docker procedures?**	**Lines of code**	**Language used**	**Is the source code available?**
A Code Injection Method for Rapid Docker Image Building [23]	Up to 100000x faster	Can only be used with interpreted languages, limited to modifications in the source code	Yes	Unknown	Unknown	No
FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container [11]	Up to 10x faster	Limited to network activity	Yes	2600	Go	No
Slacker: Fast Distribution with Lazy Docker Containers [6]	Up to 20x faster	Limited to network activity	Yes	Unknown	Unknown	No
Docker Buildx [2]	Unknown	Unknown	No	Unknown	Go	Yes

Table 1.1: Works about speeding up Docker builds

Na tabela tens uma coluna sobre complexidade de implementação que é muito subjetiva. Melhor do que isso seria usares colunas com coisas objetivas mas que te permitam concluir sobre a complexidade no texto que acompanha a tabela.

Wang et al. [23] 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.

[2]Docker Buildx, https://docs.docker.com/engine/reference/commandline/buildx/

Huang et al. [11] and Harter et al. [6] 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. FastBuild was implemented using around 2600 lines of Go code.

The works described so far interfere with the normal Docker build procedures and do not have public source code for their tools. This makes them harder to implement. Another solution is offered by the Docker development team, Buildx. 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.

1.5 Dockerfile Generation

Many projects out there contain Dockerfiles that can not even be used to build an image [25, 15]. Therefore, having a way to generate a functinoal 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.

No parágrafo anterior, "easy" não parece ser o melhor adjetivo — fácil comparaticamente com quê? No artigo que citas estávamos interessados sobretudo em aferir quão mais trabalhosas eram umas atividades **comparaticamente com outras**. Este "easy" não tem uma base de comparação tão clara.

As duas primeiras frases que aqui tens estão a servir essencialmente para motivar a importancia do que vais falar nesta secção. Algo que pode ajudar mais a fazer isso é definires bem os objetivos de todo este capítulo (com as perguntas, secção sobre os challenges, etc. Ver o que escrevi acima).

[3] Docker BuildKit, https://docs.docker.com/build/buildkit/

Name	**Successful generation rate**	**Limitations**
Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations [22]	Unknown	An Open API Spec is required
Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph [29]	Up to 80%	A vast knowledge graph (2832 nodes and 62614 edges) is required, focused on Singularity
Container-Based Module Isolation for Cloud Services [12]	Unknown	Requires the use of templates to generate the files
DockerGen: A Knowledge Graph based Approach for Software Containerization [27]	Up to 73%	A vast knowledge graph (900000 nodes and 2900000 edges) is required
DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets [10]	Up to 30%	Limited to Python snippets, requires a knowledge base
ExploitWP2Docker: a Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges [4]	Up to 39%	Limited to security testing scenarios, requires an exploit description
MAKING CONTAINERS EASIER WITH HPC CONTAINER MAKER [18]	Unknown	Requires Python code to generate the files

Table 1.2: Works about generating Dockerfiles

Se a memória não me falha, o DockerizeMe só fuciona para snippets, e não para um sistema completo.

A coluna sobre a complexidade é um pouco subjetiva...

Zhong et al. [29], Ye et al. [27] and Horton et al. [10] 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 [4], a containerization tool similar to Docker but focused on HPC (High Performance Computing).

Caturano et al. [4] propose a tool that uses Docker to generate security testing environments from exploit descriptions. Sorgalla et al. [22]'s work can generate Dockerfiles from models, which

[4] Singularity, https://apptainer.org/

are generated from Open API Specifications (like Swagger [5]). Kehrer et al. [12] use Apache FreeMarker [6] to generate Dockerfiles from templates. McMillan et al. [18] 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.

1.6 Dockerfile Smells

Smells are commonly found in Dockerfiles [24], making it important to create ways of detecting and, if possible, remove them. This section covers works related to this. Table 1.3 contains a list of smells found in literature.

Smell	**Related works**	**Related findings**
Temporary File	[17] and [26]	The smell is quite common and can be divided into 4 different types; can be detected through static and dynamic analysis
SATD	[2]	This type of smell appears in Dockerfiles and can be divided into several classes and subclasses
Lack of version tagging/pinning	[24]	N/A
Use of the MAINTAINER instruction	[24]	N/A
Use of *cd* to switch directories instead of WORKDIR	[24]	N/A
The parameter *–no-install-recommends* is not used when installing packages with APT	[24]	N/A

Table 1.3: Dockerfile smells

[5] Swagger, https://swagger.io/
[6] Apache FreeMarker, https://freemarker.apache.org/

Mais do que listar/caracterizar artigos, idealmente estas tabelas seriam sobre as coisas que te interessam encontrar. Por exemplo, nesta secção estás interessado é nos smells, logo, acho que faria mais sentido ser uma tabela de smells, em que indicavas o artigo em que o encontraste (em vez de uma tabela de artigos em que indicas os smells, estou a fazer sentido?). A mesma ideia é aplicável às outras secções.

Lu et al. [17] and Xu et al. [26] 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. [2] 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. [24] 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. For the same reasons, only some of the analyzed smells were listed here.

1.7 Dockerfile Good Practices

To prevent the creation of smells like the ones mentioned in Section 1.6, a developer should follow good practices. This section goes over works that cover these practices. Table **??** contains a list of good practices found in literature.

Practice	**Related works**
Format for clarity	[19]
Document within the Dockerfile	[19]
Specify software versions	[19]
Order the instructions	[19]
Run the container in rootless mode	[20]
Use tagged minimal images and multistage builds	[20]
Use COPY with specific parameters	[20]
Update and install packages in the same RUN instruction	[20]
Use COPY instead of ADD	[20]
Do not leak sensitive information to an image	[20]
Remove unnecessary dependencies	[20]
Only expose ports that are needed	[20]
Use official images when possible	[20]
Remove temporary directories	[8]
Use flag *-f* with curl	[8]
Remove tarballs after extraction	[8]
Do not use APK's cache	[8]
Do not install dependencies recommended by APT	[8]
Use HTTPS urls with curl	[8]
Use batch flag with gpg	[8]
Use HTTPS urls with wget	[8]
Use flag *-y* with *apt-get install*	[8]
Remove APT lists after package installation	[8]
Run *apt-get update* before *apt-get install*	[8]

Table 1.4: Dockerfile good practices

Henkel et al. [8] 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. Some of these "gold rules" are not listed above because it's not clear what they refer to. It should be possible to implement the listed ones as repairs.

Prinetto et al. [20] 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. [19] 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.

1.8 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 [5] and many developers do not have the knowledge required to evaluate how vulnerable their containers are [28]. 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 [5]	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 [28]	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 [16]	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 [1]	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 [20]	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. [5] 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. [28], 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. [1] used Cimplifier [21] to debloat containers used in machine learning environments and found the amount of vulnerabilities present in those containers was significantly reduced.

Linnalampi et al. [16] 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. [20] 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.

1.9 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 [13]	Base image update	Does not cover other parts of the Dockerfile	Medium
Learning from, Understanding, and Supporting DevOps Artifacts for Docker [8]	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 [7]	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 [9]	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 [3]	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. [13] focused on updating a container's base image by analyzing the available tags, while Hassan et al. [7] focused on portions of the Dockerfile which are tied to values in the source code.

Henkel et al. [8] offers a way to detect violations of the gold rules they obtained but don't automate the repair of said violations. Henkel et al. [9] 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. [3] 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.

1.10 Dockerfile Bloat

1.11 Dockerfile Testing

1.12 General Discussion

Ksontini et al. [14] 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.

Seria aqui um bom sítio onde responder de forma mais explícita às perguntas que colocamos no início do capítulo.

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Mingjie Li, Xiaoying Bai, Minghua Ma, and Dan Pei. DockerMock: Pre-Build Detection of Dockerfile Faults through Mocking Instruction Execution.

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

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An Empirical Study of Build Failures in the Docker Context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 76–80. ACM.

[26] 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.

[27] 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.

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

[29] 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.