VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of de Ingenieurswetenschappen: Computerwetenschappen

FOLLOW YOUR NOSE

Consolidating and detecting Dockerfile smells with a focus on bloaters and layer-optimization

Ian Angillis

September 2023

Promotor: prof. dr. Coen De Roover
Advisors: dr. Ahmed Zerouali, dhr. Ruben Opdebeeck

sciences and bioengineering sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

VOLG UW NEUS

Het consolideren en detecteren van *Dockerfile smells* met een nadruk op *bloaters* en *layer-optimization*

Ian Angillis

September 2023

Promotor: prof. dr. Coen De Roover
Advisors: dr. Ahmed Zerouali, dhr. Ruben Opdebeeck

wetenschappen en bio-ingenieurswetenschappen

Abstract

My abstract goes here.

Keywords - Docker, empirical software engineering, software quality, code smells

0.0.1 Ideas - to delete

- First proper catalogue of Dockerfile smells

Acknowledgement

"*Vis et honor*"

My acknowledgement goes here.

Contents

Chapter 1

Introduction

Writing good Dockerfiles has become a speciality. It is not just about configuring the environment of your application, but about doing this in a proper manner. Bad Dockerfiles could increase maintenance of the application, increase the size of the application, slow down development, introduce bugs, increase the attack vector of the application and incur cost on various dimensions such as time and financial. This thesis sets out to consolidate these bad code practices with regards to Dockerfiles and, using a new tool to analyse Dockerfiles, attempts to hint at the prevalence of a subset of these bad code practices in the industry.

1.1 Context

1.1.1 Virtualization

Dockerfiles find themselves in the domain of virtualization technologies. These virtualization technologies are an important part of the software development lifecycle and form the basis of cloud computing (Dillon et al., 2010).

One of these virtualization technologies are virtual machines. Virtual machines allow for a better and more efficient allocation of physical resources by running multiple virtual machines on a single physical machine. These virtual machines are managed by an hypervisor that either runs on top of the hardware of the physical machine, meaning that these virtual machines abstract the hardware layer, or run on top of an operating system. Despite a better allocation of resources, virtual machines are slow to boot and take up quite a bit of space. As only the hardware is abstracted away, virtual machines still need to contain and boot an entire operating system. This is a slow process and takes up gigabytes in space.

Another virtualization technology, containers, packages an application and its dependencies together in a single container that is light-weight, portable, reproducible and fast to boot. Containers are abstracted away at the operating system layer and share kernel functionalities. These containers can run in parallel on a physical or virtual machine. Each container has their own filesystem, application and dependencies. This solves the problem of dependency hell, as the container should exhibit the same behaviour regardless of the machine that can work with those containers. “It works on my machine” is no longer an excuse, not even for academics.(Boettiger, 2015)

Using these technologies in tandem yields the most benefit. As one physical machine can have multiple instances of virtual machines, where each virtual machine can have multiple instances of containers. Where each containers runs a different application, with different dependencies. This lies at the basis of cloud computing as resources can be compartmentalised ad-hoc. Cloud service providers offer computing power and services such as Software as a Service (PaaS), Platform as a Service (Paas), Infrastructure as a Service (IaaS) and database as a Service (DBaas) to their clients (Dillon et al., 2010) (Pahl, 2015).

1.1.2 Docker

According to the 2023 StackOverflow Survey[1], Docker[2] is currently the industry standard and de facto containerization tool. Docker allows software practitioners to make highly-scalable, lightweight and portable applications which can run locally or in the cloud. Additionally, these containers are useful for researchers too as they allow for reproducible research(Boettiger, 2015). Docker is enabled by a Docker daemon which is running on either a remote server or on the local host which can be communicated with through a client. In the case of a remote server, communication is likely to happen through a web interface. Locally, it is likely to happen through a terminal. This Docker daemon is responsible for building, running and managing these Docker containers.

Docker containers themselves work roughly as described earlier in this introduction and are instantiated from Docker images. Such a Docker image can be described as a blueprint for a Docker container, and a container can be described as an instance of a Docker image. A Docker image can be described as a stack of layers, where each layer represents the a filesystem containing the differences with the previous layer. Whenever a layer is added, the Docker daemon has read/write permissions, but as soon as a new layer is added on top then the previous layer becomes read-only. Adopting this `COPY-ON-WRITE` technique, instantiating a Docker container is a light process as a new `read/write` layer will be added on top of the stack of the Docker image. Any write operation in this layer will not change the layers under it, and therefore breaking down the container comes down to popping the last layer from the image. This way, Docker images never change and keep their state constant. Docker images form the basis of other Docker images and are shared with other software practitioners in public and private registries such as DockerHub[3].

The Docker images are constructed from Dockerfiles. A Dockerfile, which is the core of this thesis, is an infrastructure-as-code (IaC) artifact that is comprised of a set of instructions. These instructions allow software practitioners to construct and set-up the environment of the container in which their application needs to run. Each Dockerfile starts from a base image specified by the `FROM` instruction. Each subsequent instruction adds another layer to the image which contains the differences of the previous layer. Other instructions such as the `RUN` instruction allows software practitioners to execute commands in a shell to, for example, install dependencies and software packages. The `COPY` instruction is a way to introduce files from the build context, the repository on the system in which the `docker build` command was executed, to the image. The `ADD` instruction also serves this purpose, but has extra functionalities such as unpacking compressed files and the ability to fetch resources from the internet. The `CMD` instruction specifies the

[1] https://survey.stackoverflow.co/2023/
[2] https://www.docker.com/
[3] https://hub.docker.com/

last command that should be invoked when the container boots. The whole set of instructions can be consulted in the Docker documentation[4].

Code listing 1.1 gives an example of a Dockerfile. This Dockerfile builds upon a base image with the Debian operating system in line one. In line two, using the `RUN` instruction, `NodeJS` is installed in the image using the `apt-get` API. The `COPY` instruction in line three copies the entire build context into the image on position `/usr/src/app`. In line four, by means of a `RUN` instruction, the current working directory of the image is changed to `/usr/src/app` and subsequently `npm install` is invoked on that location such that the dependencies are installed on the image. Line 5 exposes port 80000 which allows for communication between the container and the world wide web. Lastly, a `CMD` instruction will make sure the specified command is invoked when the container is initialised.

```
1 FROM debian
2 RUN apt-get update && apt-get -y install nodejs
3 COPY . /usr/src/app
4 RUN cd /usr/src/app && npm install
5 EXPOSE 80000
6 CMD ["node", "src/index.js"]
```

Listing 1.1: Dockerfile example

1.1.3 Code smells

As a Dockerfile is product of code, it is not exempt from code smells. Code smells are patterns that are a product of bad design or going against best practices(Fowler et al., 2003). Code smells are not a bug, as with code smells the code is syntactically correct, and thus the code will run, but the code is semantically unsound in the sense that it can and will cause negative effects. These negative effects range from maintenance to security to configuration ... In the book of Fowler et al., 2003, examples of code smells are given. One of those is code duplication. Code duplication causes smelly code because if the logic needs to change, then it has to change in all the places where that particular code is present. If the code modification was not done in all the places containing that duplicated code, unexpected behaviour might occur. Therefore, a general solution is to extract the duplicated code into its own method and call the method in those different places. If then the logic needs to change, it only need be done in one place.

Wu et al., 2020 defined Dockerfile smells as "the instructions of a Dockerfile that violate the recommended best practices and potentially affect the Dockerfile's quality in a negative way.". Dockerfiles are the basis of Docker images and thus Docker container. The quality of a Dockerfile will ripple throughout the entire process and thus it is important to write good Dockerfiles. The Dockerfile in code listing 1.1 is not a good one, as it contains a lot of Dockerfile smells:

- Line one has a base image of which the version is not pinned to a certain version. Additionally, the base image is too large for its purpose and thus takes up more space than is necessary.
- Line two uses the apt-get API to install NodeJS. As with the previous item, software packages need to specify their version. Furthermore, due to the light-weight philosophy of containers, it is imperative to clean any caching mechanism that is not necessary in order to run the application.

[4] https://docs.docker.com/engine/reference/builder/

- In line four, the current working directory of the image is changed through a `RUN` instruction whilst Docker offers the `WORKDIR` instruction. Not using the `WORKDIR` instruction negatively affects the maintenance of the file as it is easier to get confused where the current working directory is.
- Line five exposes a port that does not exist.
- Lastly, as this is an NodeJS application and dependencies need be installed, dependencies should be installed before introducing the application to the image as they take a lot of time to install and generally are less prone to code churn than the application itself. Due to Docker's layer caching mechanism which can reduce build time significantly, one single change could bust the cache and make the image build again from that point.

Code listing 1.2 represents the Dockerfile code listing in 1.1 without Dockerfile smells. This has a positive impact on readability, image size and subsequent build time.

```
1 FROM node:alpine
2 RUN apt-get update && apt-get -y --no-install-recommends install nodejs=15 \
3 && clean cache && rm -rf /var/apt/lists
4 COPY package*.json /usr/src/app
5 WORKDIR /usr/src/app
6 RUN npm install
7 COPY . /usr/src/app
8 EXPOSE 3000
9 CMD ["node", "src/index.js"]
```

Listing 1.2: Dockerfile example

Software practitioners are expected to write Dockerfiles with these best practices in mind. Fortunately, this need not be only manual work as open-source software tools such as Hadolint ("Haskell Dockerfile Linter", 2023) can aid developers by detecting these smells. Ultimately, writing good Dockerfiles can save developers a lot of time and financial hurdles. As containers are extensively used in cloud computing and companies pay for what they use in both space and time, good Dockerfiles can increase development speed by reducing build times and image size. Which in turn can bring down financial costs for companies.

1.2 Problem

Earlier research on large bodies of Dockerfiles point out that Dockerfile smells are prevalent and insist on more automatic tooling with regards to these Dockerfile smells (Cito et al., 2017)(Lin et al., 2020)(Eng & Hindle, 2021). These studies, however, make use of Hadolint ("Haskell Dockerfile Linter", 2023) which only uses a top-level syntax tree to parse the Dockerfile instructions, but not the nested language contained within the `RUN` instructions (Henkel et al., 2020a). Some research did not use Hadolint and went the other way around by gathering high-quality Dockerfiles which were used to derive best practices for writing Dockerfiles (Henkel et al., 2020a)(Zhou et al., 2023). Other research introduced more specific smells such as the temporary file smell (Lu et al., 2019) or defined code smells.

The problem that this thesis addresses is threefold:

1. **Dockerfile smells are scattered throughout the literature**
2. **Research studies the same small subset of Dockerfile smells using Hadolint**

3. Lack of tools covering other smells

With regards to the first problem, software practitioners are supposed to write Dockerfiles adhering to the best practices as described by Docker's offical best practices[5]. This is challenging, as there is a lack of awareness and attention to read these practices (Cito et al., 2017). On top of that, following Docker's best practices does not comprise all the Dockerfile smells out there. At the time of writing, all the code smells pertaining to Dockerfiles are spread out throughout the research literature, open-source tools and the official Docker documentation. There is no central place where these Dockerfile smells are consolidated and can be consulted by both software practitioners and researchers. Researchers would benefit from such a consolidation too as there is an increasing need for more automated tools with regards to writing good Dockerfiles (Cito et al., 2017) (Wu et al., 2020) (Wu et al., 2023). A consolidation of the known Dockerfile smells would allow researchers to create these news tools from a holistic point of view, or focus on a specific classification of Dockerfile smells.

With regards to the second problem, many of the empirical research has been performed using Hadolint and thus was done on the same smells. Therefore, there is analysis missing for other smells not covered by Hadolint. Additionally, the empirical studies have focused on large bodies of Dockerfiles mined from Github[6], Bitbucket[7] and Dockerhub[8]. To the best of our knowledge, no analysis is done on a dataset mined from StackOverflow [9].

1.2.1 Summary

To summarize, this thesis aims to answer the following research questions:

- RQ1: What is the current state of Dockerfile smells?
- RQ2: Does our tool perform well compared to the state-of-the-art?
- RQ3: How prevalent are the detected smells in the GitHub ecosystem?
- RQ4: How prevalent are the detected smells in the StackOverflow ecosystem?

1.3 Structure

We have reached the end of the introduction. In chapter 2 we discuss prior research that is relevant to the topic and goal of this thesis such as virtualization, Docker, code smells and the state of the art. In chapter 3 the Dockerfile smell catalogue is introduced. Next, the implementation of our tool is discussed in chapter 4. Lastly, we give an evaluation of our tool on the datasets we have gathered in chapter 5 and conclude the thesis in chapter 6.

[5] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
[6] https://github.com/
[7] https://bitbucket.org/
[8] https://hub.docker.com/
[9] https://stackoverflow.com/

Chapter 2

Background

The aim of this chapter is to provide the necessary background to inform the reader of the context of this thesis. We start with an overview of virtualisation in section 2.1. Then, we move on to explain Docker in section 2.2 and Code smells and how they relate to Docker in section 2.3. Section 2.4 dives into the literature with regards to Dockerfile smells whilst section 2.5 dives into the state of the art with regards to Dockerfile smell detection. Finally, we give an overview of the best practices that impact the size of Dockerfiles in section 2.6 and conclude the chapter in section 2.7.

2.1 Virtualization

Virtualization technologies can be used to create virtual representations of physical machines such as servers, storage, networks, etc... In this section, we zoom in on virtual machines in section 2.1.1 and containers in section 2.1.2. We finish this section with an important application of virtualization in section 2.1.3: cloud computing.

2.1.1 Virtual Machines

Virtual machines allows us to have multiple instances of a machine running in parallel a single physical machine. These virtual machines abstract away the hardware necessary in order to run and are managed by an hypervisor. An hypervisor is a piece of software that manages the the virtual machines by allocating resources such as CPU, memory and storage from the physical machine to each virtual machine.

There are two kinds of hypervisors:

1. **Bare-metal hypervisor**: this hypervisor is installed directly on the hardware.
2. **Hosted hypervisors**: this hypervisor is installed on an operating system.

The bare-metal hypervisor yields better performance than the hosted hypervisor as it interacts directly with the hardware resources. However, the hosted hypervisor, such as VMWare[1] allows end-users to install virtual machines on their personal devices. This is useful as a virtual machine can be used as a sandboxing tool.

[1] https://www.vmware.com/

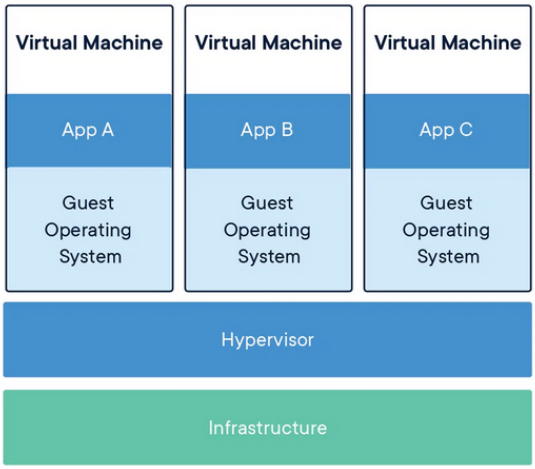

Figure 2.1: Virtual machines ("What is a Container?", 2021)

Figure 2.1 illustrates how this all fits together in a technology stack. The infrastructure in this image is either the hardware of a physical machine, or the hardware and operating system of a physical machines. The hypervisor, bare-metal or hosted depending on the infrastructure, hosts and manages three virtual machines. Each of these virtual machines have their own guest operating system and run their own applications.

Using virtual machine has some major upsides and they add up to each other:

- Better resource allocation
- Better security
- Better management

The better resource allocation is translated in the ability to have multiple virtual machines run in parallel on one physical machine. This means that one physical device can serve multiple roles and get companies or individuals more bang for their buck. A plausible effect is that because of the better allocation of resources, companies or individuals need to purchase less physical machines to serve their business needs, which drives infrastructure costs down. There are less physical machines to maintain, which improves maintenance. As there are less physical machines, it improves security as there are less devices to target, or access through side-channel attacks.

If a physical server breaks down or is the target of an attack, it can take a long time to regain access, fix, or replace the physical server. As virtual machines are virtualized isolated environments, the process of recovering can be done in minutes because the hypervisor can spin up another virtual machine. This increases resilience and drives costs down for companies. For example, cloud providers such as AWS and Azure have a Service Level Agreement which is a formal contract that specifies performance, up-time and quality expectations between the provider and the client.

Lastly, these virtual machines are virtual. And as such, they can be managed and automated by other software tools. This simplifies the workflow and improves maintenance and management. This is of interest to companies such as cloud providers who can exist because of virtualization.

Virtual machines also have some downsides:

- Take up a lot of space
- Dependency hell
- Slow to boot
- Single point of failure

A virtual machine, typically with the intent of running an application or service, requires the full copy of the operating system, the application binaries, the application dependencies, ... This means that a virtual machine takes up to gigabytes of space to run an application or service that might require a fraction of the space that the operating system needs.

A virtual, or physical machine, running different applications can run into a problem called dependency hell. This happens when multiple software applications require the same dependency, but each one needs a different version.

Whilst virtual machines allow for scalability, fast scalability and elasticity are a challenge. If a virtual machine provides a service, and the load is high enough such that another instance of that service is required, the hypervisor might spin up another instance of that virtual machine. As a virtual machine contains an entire operating system, this means that the entire operating system needs to boot before the service can be run on that virtual machine. This is a slow process that can take up to several minutes. This is not ideal in a situation where elasticity is required.

Lastly, virtual machines allow for a faster recovery from unfortunate accidents as they still run on a physical machine. When this physical machine fails, all the virtual machines on the physical machines will fail. Thus, the physical machine is a single point of failure.

2.1.2 Containers

Containers, unlike virtual machines, are abstracted over the operating system rather than the hardware. This is known as operating-system-level virtualization: a technique where the kernel can run multiple isolated processes, each process in user-space. This means that multiple containers can run on the same system and share some core kernel processes, without knowing of each others existence.

Containers package the application and its dependencies together. These containers can run in parallel on a physical or virtual machine that supports these containers. By packaging the application and its dependencies together and running them as an isolated process, means that containers overcome the issue of dependency hell (Merkel, 2014). As container `A` with an application that uses version `X` of a specific software package, will run just fine alongside a container `B` with an application that uses version `Y` of that same software package. Additionally, this also means that a container will behave in exactly the same way on any machine, physical or virtual,

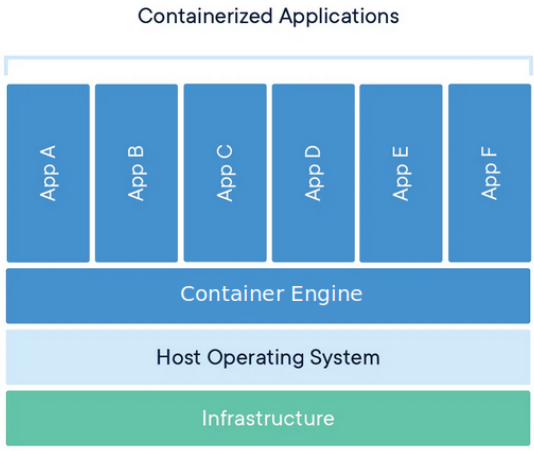

Figure 2.2: Containers ("What is a Container?", 2021)

that supports these containers. No more ""But it works on my machine" excuses.

Figure 2.2, just like figure 2.1 on virtual machines, gives us a proper overview of how this all fits in a technology stack. The infrastructure here can be three things:

1. Physical hardware
2. Physical hardware and a bare-metal hypervisor running an instance of virtual machine.
3. Physical hardware with an operating system on top that runs a hosted hypervisor running an instance of a virtual machine

The operating system is either the operating system running directly on the hardware, or running in a virtual machine. The container engine is a piece of software that manages these containers. The container engine can manage multiple containers who each run in their own isolated process and can run in parallel.

Containers have multiple upsides:

- Portability
- Scalability
- Reproducible

Containers bundle the application code and its dependencies together. As they are abstracted on the operating system layer, they need not carry around a full operating system like virtual machines. This results in containers being light-weight and thus being a highly portable entity which can easily be shared and deployed on different devices. Sometimes even over the network. This also means that they do not need to boot an entire operating system. Containers are scalable and, unlike virtual machines, can quickly be deployed or removed, depending on the workload.

With the application code and its dependencies bundled together, the behaviour of the application is the same on every machine that supports managing these containers. This is extremely

useful for software practitioners in the industry. Again, "But, it works on my machine" is not a valid excuse anymore. Additionally, reproducibility is important for researchers too. Where the laws of physics generally do not change, computing environments do. As more and more research fields rely on computational experiments, it is important to make sure that these experiments are reproducible for future work and verification. Containerization tools, like Docker, solve this problem (Boettiger, 2015).

Lastly, containers allow software practitioners to improve their development lifecycle as these containers can be automated. Even more, they can work together and enable developers to create microservices instead of approaching development from a holistic angle.

Containers have multiple downsides:

- Security
- Single point of failure
- Setup complexity

As multiple containers share the same resources of a machine and underlying operating system, a detected vulnerability of a container could also endanger other containers. Additionally, just like virtual machines, there is a single point of failure that also endangers the containers. If the containerization engine or host machine fails, all the running containers will fail.

Applications can be build from a holistic approach where one application contains all the services that that application provides. This means that when developers modify one line of code in the application, that the entire applications needs to be rebuild and deployed. Even the parts that were not affected. Containers offer the ability to work with microservices where instead of one big application, the application is a collection of micro applications each responsible for a part of the application. These parts of the application communicate with each other over the network. While this is a strength of containers, it is also a weakness as setting up this kind of application is complex and in general requires a container orchestration tool such as Kubernetes[2] or multi-container tools such as Docker Compose[3] to handle the deployment, scaling and management of these applications.

2.1.3 Application of virtualization

Whilst in some cases virtual machines are preferred over containers and vice-versa, it would be amiss to consider them as substitutes. The real power of virtualization lies in combining these ideas together. A physical machine can run multiple instances of virtual machines, where each virtual machine contains a container engine and potential container orchestration tool to manage containerized applications, services or other tools. This way, the resources of a single physical device can be allocated very efficiently.

This lies at the basis of cloud computing (Pahl, 2015). Cloud computing is a service provided by cloud providers such as AWS[4], Microsoft Azure[5], Google Cloud[6], etc. Whilst these cloud

[2] https://kubernetes.io/
[3] https://docs.docker.com/compose/
[4] https://aws.amazon.com/?nc2=h_lg
[5] https://azure.microsoft.com/en-us/
[6] https://cloud.google.com/

providers have different structures and targets, the business model is the same. They offer computing services such as computing power, storage, databases, applications, analysis, AI-solutions, etc.

When companies and institutions decide to move to the cloud, they do not have to invest anymore in the required infrastructure such as physical machines, maintenance personnel, in-house knowledge, storage, electricity, internet, security and all other related costs. Instead, the cloud providers take care of all of that. They buy, maintain, secure and upgrade computing infrastructure and can that way leverage economies of scale. Their clients can use this computing infrastructure via different models such as:

- Software as a Service (SaaS)
- Platform as a Service (Paas)
- Infrastructure as a Service (IaaS)
- Database as a Service (DBaas)

This way, clients only pay for the computing power that they use, coined "finops[7]". And by leveraging the virtualization tools mentioned earlier, cloud providers can serve many clients in a very efficient way as the resources of their infrastructure are efficiently allocated.

2.2 Docker

This section documents on the Docker technology. The overal architecture is discussed in section 2.2.1. Followed up by explaining Docker images in section 2.2.2. Lastly, we go over Dockerfile, which stands at the center this thesis, in chapter 2.2.3.

Docker[8] is a tool to build, deploy and maintain Docker containers. It was first released to the public in 2013 by dotCloud, which later became Docker Inc. At the time of writing, Docker is the most popular tool with regards to containerization and the de-facto industry standard. According to the StackOverflow survey[9] of 2022, Docker is becoming a fundamental tool for professional developers on top of being the most loved tool. In the survey results[10] for 2023 published in May, Docker is the most popular tool.

2.2.1 Docker architecture

In this section we give an overview of the Docker architecture. Figure 2.3 shows that the Docker architecture can be divided up in three different components:

- Docker host
- Docker Client
- Docker Registry

[7] https://cloud.google.com/learn/what-is-finops
[8] https://www.docker.com/
[9] https://survey.stackoverflow.co/2022/
[10] https://survey.stackoverflow.co/2023/

The Docker host running the Docker daemon is central to how Docker works. The Docker daemon runs on a physical or virtual machine and is responsible for building images, running containers, maintaining images and containers, ... The Docker daemon listens to API requests either from the local machine or from the network.

The client interacts with the Docker daemon through requests. When the client is on the same device as the Docker daemon, requests can be sent using the terminal on the device or using applications such as Docker Desktop[11]. When the client is a web-based application, such as cloud providers, the requests are sent over the network to the device which has the Docker daemon running.

The registry is a central place to store and share Docker images. Dockerhub[12] allows anyone to store Docker images and use these as a basis to build other Docker images.

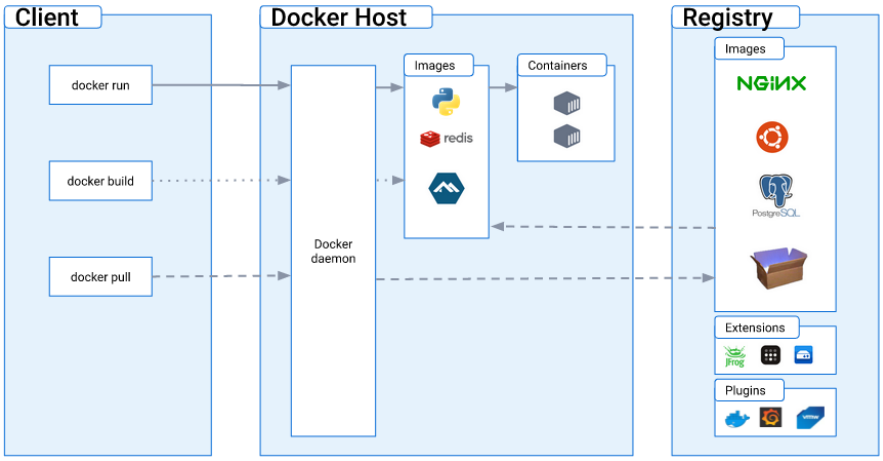

Figure 2.3: Docker Architecture [13]

2.2.2 Docker Image

A Docker image can be seen as the blueprint for a Docker container, and a Docker container can be seen as an instance of a Docker image. Concretely, a Docker image is a stack of read-only layers. Where each new layer contains the differences in the filesystem with regards to the older layer.

These read-only layers are what makes Docker containers light-weight and portable. Whenever a Docker container is instantiated, a new ephemeral read-write layer is created on top of the stack of layers. When this read-write layer wants to modify the content of a file in the lower layers, this file is copied to that read-write layer and then kept there, also known as `COPY-ON-WRITE`. This means that the same image can be used to instantiate multiple containers as the content of the image will not be altered by changes made of an active container using that image. The only

[11] https://www.docker.com/products/docker-desktop/
[12] https://hub.docker.com/
[13] https://docs.docker.com/get-started/overview/

changes that are happening, happen in the read-write ephemeral layer on top of the read-only layers. This also means that when a container is taken down, only this top layer is removed. This enables these Docker containers to boot quickly, and also to be killed quickly.

These read-only layers also have the benefit of leveraging cache when building Dockerimages. Each layer has a hash, and when that hash is changed it means that the layer has changed. Therefore, so long no layer is changed Docker will use layers from the cache. But as soon as a change is detected, the Docker image will be rebuild from that layer on.

2.2.3 Dockerfile

A Dockerfile is an Infrastructure-as-Code (IaC) artifact which the Docker daemon uses to build a Docker image. A Dockerfile contains of a set of instructions[14] that allows software practitioners to configure the environment, the file system, of the container. Docker has specified different instructions that can be used in a Dockerfile. We list most of the instructions:

- The `FROM` instruction sits on top of every Dockerfile and specifies the base image on which a new image will be build. This base image can be any valid Docker image, whether it be an official image from Docker Hub or a private image from a private registry.
- The `RUN` instruction will execute a command in a shell. This can be any command specific to the operating system. The shell also depends on the operating system.
- The `COPY` instruction copies the specified files from the build context, the directory in which the build request is sent to the Docker daemon, to the image.
- The `ADD` instruction adds the specified files form the build context to the image. The ADD instruction has some other functionalities such as decompressing a file before adding it to the container, or fetching a resource from the internet.
- The `WORKDIR` instruction changes the current working directory of the container.
- The `ENV` instruction allows the author of the Dockerfile to set environment variables within the container.
- The `ARG` instruction defines a variable that is passed to the builder at build-time. This means that it is not available anymore when the container is up-and-running. When this is required, use `ENV` instead
- The `EXPOSE` instruction exposes specified ports to allow for network communication with the container.
- The `CMD` instruction executes the binary or executes a command that is required in order to launch the application or some code when the container is built.

Listing 2.1 is an example of a Dockerfile. In line 1 the `node:18-alpine` image is specified as the base image. In line 2 the current working directory, default `/`, is set to `/app`. In line 3, the entire build context is copied into the current working directory of the image. In line 4, the `npm install` command is executed to install the dependencies of the application. In line 5, the port 3000 is exposed which allows for network traffic via that specific port. In line 6, the node application is started.

[14] https://docs.docker.com/engine/reference/builder/

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Listing 2.1: Dockerfile example

2.3 Code Smells

This section goes over code smells and in particular explains the impact Dockerfile smells can have in section 2.3.1.

Fowler et al., 2003 coined the term code smells as code patterns that suggest for the possibility of refactoring. These patterns are a result of bad design and going against best practices. This means that the code is syntactically correct, and thus runs, but is semantically not sound. Whilst this is a vague description, it is worth pointing out that these smells are different for different programming languages and technologies. Every programming language has a different syntax, a different implementation under the hood, a different philosophy and different best practices. This means that these smells can appear in many different forms, in all kinds of software targeting all kinds of different domains, and have negative effects on multiple aspects such as security(Rahman et al., 2019), maintainability (Yamashita, 2014), testability(Sharma & Spinellis, 2018), performance(Sharma & Spinellis, 2018), effort(Sharma & Spinellis, 2018), cost(Sharma & Spinellis, 2018), configuration(Sharma et al., 2016), ...

In an insightful survey on software smells by Sharma and Spinellis, 2018, the authors synthesized five defining characteristics of a software smell:

1. Indicator or symptom of a deeper design problem
2. Poor or suboptimal solution
3. Violation of best practices (of the domain)
4. Impacts quality negatively
5. Recurrence

The authors also provided a meta-classification of the smells such as (i) effect-based smells, (ii) principle-based smells, (iii) artifact characteristics-based smells, (iv) granularity based smells. Each meta-classification has their own classifications. E.g. some of the classifications identified for effect-based smells include couplers and bloaters. These classifications are an insightful tool to categorize a long list of code smells. This can be useful for research when building automated smell-detection tools for a specific kind of code smell.

As a concrete example of a code smell, imagine the violation of the “Do not repeat yourself” principle(Hunt & Thomas, 2000). This principle means that duplication is not desired and should be avoided. As having the same logic in different places is bound to introduce bugs as changes to that logic needs to happen in different places. Whilst this is a relatively simple example of a code smell, code smells and their detection ranges from relatively simple to non-trivial endeavours.

2.3.1 Code Smells in Dockerfiles

Code smells pertaining to Dockerfiles can have different effects on both the Docker file themselves and the Docker images that are build from the Dockerfile. In a blogpost[15] on the Docker website, Tibor Vass lays out some of the best practices for developing Docker files and the kind of effects that following these practices have. In general, code patterns going against these best practices can be considered Dockerfile smells. We lay out the effects in the subsequent subchapters.

Incremental build time

Incremental build time leverages the cache of layers that is being kept when the Docker daemon builds Docker images. When the daemon builds a docker image, it will only rebuild the image when it reaches a layer that is changed. Each layer has a hash, thus even the slightest change will yield a completely different hash for that layer. Making it possible for the daemon to detect when it can leverage the cache or not. When a change is noticed, the image will from that point on be completely rebuild, as effects from the changed layer could propagate into the other layers. Therefore, it is recommended to write your Dockerfile in such a way that that your layers are ordered from least likely to change to most likely to change. The more layers can be taken from the cache, the faster the build time of the Docker image. It is important to note that minimizing the amount of layers will also improve build time.

Image size

Having Docker images being larger than they should be goes against the philosophy of light-weight, portable containers (Boettiger, 2015). Therefore, it is important to prevent any kind of bloating. In the case of Dockerfiles, this means that software practitioners are challenged by only importing and installing the software that is absolutely necessary for the container to function. As Docker images are a stack of layers, and each layer becomes read-only after it is processed, it is paramount to make sure that every layer is clean. Clean in the sense that nothing that is not required is left behind in the layer. Because when the layer becomes read-only, it stays there. Concretely this translates in making sure to delete the cache of package managers, avoid installing recommended dependencies and avoid introducing files that should not be there.

Maintainability

A maintainable Dockerfile is a Dockerfile that allows for easy upgrading and debugging. When a Dockerfile has low maintainability, it can take a lot of time to get things up and running.

Security

Dockerfiles should not expose any vulnerabilities, under any circumstances. Software practitioners are advised to only use official base images and check their images and software dependencies for vulnerabilities.

2.4 Dockerfile smells

In this section we discuss the prevelance of Dockerfiles with sprinkles of their impact in section 2.4.1. Some research employ different techniques such as rule mining to discover new smells as can be read in section 2.4.3. We then proceed discussing the temporary file smells in section

[15] https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/

2.4.4 and security smells in section 2.4.5. Lastly, we present configuration smells in section 2.4.6.

It is important to note that research on this topic makes extensive use of the Haskell Docker linter called Hadolint[16]. These smells, rules in Hadolint, are referenced by an identifier `DLXXXX` which will also be adopted by this thesis.

2.4.1 Prevalence of Dockerfile smells

As Dockerfiles are infrastructure-as-code artifacts, just like any other code, they are not exempt from code smells. Cito et al., 2017 conducted the first large-scale empirical study on Dockerfiles and analysed 70,197 unique Dockerfiles found on Github. This analysis showed that most Dockerfiles use an operating system as base image that is too large for the purpose of the image. This defeats the purpose of using light-weight containers to reduce the virtualization footprint(Boettiger, 2015). Additionally, using the open-source Dockerfile linter Hadolint, they found that 28.6% of the quality issues arise from missing version pinning. Other prevelant quality issues were also found such as using `COPY` and `ADD` in the wrong context, not using the `WORKDIR` instruction to change the current working directory of the image that is being built and not deleting the cache of package managers.

Version pinning is not only done for software packages that are added to the image by package managers such as `apt-get` or `npm`, but also for specifying the base image. By specifying the versions of the image and software packages that the application is guaranteed to run on, only then it is guaranteed that the behaviour is the same every single time and that the application does not run the risk of breaking down the line. It is worth noting that using the `:latest` tag is discouraged, as this would always use the most recent version of the package or image which could increase the risk of the application breaking in the future.

Whilst version pinning remains a problem (Lin et al., 2020) (Eng & Hindle, 2021), research is being done into taking away this mainly manual work from the developers.Zhang et al., 2022 propose a novel approach called DCCimagerec with the purpose of recommending potential base-images for docker images. As finding the proper base-image is a non-trivial task that can take up a lot of time. With regards to dependencies, (Hassan et al., 2018) developed RUDSEA, a novel approach which leverages context from the source code to recommend updates for Dockerfiles.

Wu et al., 2020 also used Hadolint to characterize the occurrence of Dockerfile smells in Open-Source software using three regression models. The dataset consisted of 6,334 projects gathered from Google's BigQuery. They consider both DL-smells, referring to Dockerfile smells, and SC-smells. SC-smells are code smells belonging to Shell. The `RUN` instruction allows the execution of Shell commands and thus an important part of writing good Dockerfiles is dealing with the nested shell language within the Docker specification. We refer back to listing 2.1 where in line 4 `npm install` is executed through a `RUN` instruction. Whilst the `RUN` instruction is part of the Dockerfile specification, so is `npm install`, but it is a Shell command and thus could contain code smells pertaining to the Shell language.

On their entire dataset, the authors detect 29,843 smells of which 88,9% are DL-smells and 11,1% are SC-smells. 83,8% of the projects contains at least one smell and 16,2% do not contain any smells and are considered healthy. They also found that more recent and newer Dockerfiles are likely to contain less Dockerfile smells and that projects owned by organisations are also

[16] https://github.com/hadolint/hadolint

likely to have fewer Dockerfile smells. Interestingly, Dockerfiles using Shell, Makefile tend to have more Dockerfile smells. This is because of smells with regards to using package managers such as `apt-get` in the `RUN` instruction or changing the current working directory using `cd`.

Lin et al., 2020 revisits the work of Cito et al., 2017 using a much larger and more recent dataset assembled from Docker Hub[17], GitHub[18] and BitBucket[19] consisting of more than 3,000,000 Docker images. The results of their analysis finds that over the years, software practitioners have been choosing more specific and light-weight base images. These more specific images already have a language run-time or specific application configured and the authors recommend companies to create their own specific pre-configured images to keep the resulting Docker images as small as possible. The only OS image that saw an increase in use is the `Alpine`[20] base image which is a light-weight Linux distribution.

Rule	Percentage	Definition
DL3008	13.60%	miss version pinning in apt install
DL3015	9.51%	miss –no-install-recommends
DL3020	9.04%	incorrectly use ADD instead of COPY
DL4006	8.31%	not using -o pipefail before RUN
DL4000	7.65%	deprecated MAINTAINER
DL3003	6.97%	incorrectly use cd command instead of WORKDIR
DL3009	5.64%	not cleaning apt cache
DL3006	3.30%	untagged version of the image
DL3025	2.23%	not using JSON notation for CMD
DL3007	1.90%	using the error-prone latest tag

Table 2.1: The definiton and overall proportion of the most frequent smells in Dockerfiles of influential images.(Lin et al., 2020)

The authors also studied some evolution trends with regards to Dockerfile smells between the years of 2015 to 2020. A first evolution is the distribution of smell count in Dockerfiles over time, in figure 2.4 the authors show that the smell count in Dockerfiles has a downward trend. As of 2020, the average smell count is 5.13 per Dockerfile, with the median being 4.0 for every Dockerfile. Only a small portion, 7.78%, of their entire dataset did not contain smells.

A second evolution is about the prevalence of Dockerfile smells. They authors only did this on popular Dockerfiles, which they defined as the top 50 most popular images on Docker hub. In table 2.4 the authors give an overview of the smells, using Hadolint, they checked for and the percentage of the popular Dockerfiles that contains the specified smell. Using figure 2.5 the authors show that smells such as incorrectly using ADD instead of COPY (DL3020), not cleaning the apt-cache (DL3009) and installing recommended packages (DL3015) are becoming less prevalent. Whilst highlighting that smells such as not using the WORKDIR instruction to change the current working directory (DL3003) and not using `-o pipefail` before executing a command using the RUN instruction are becoming more prevalent. It is also clear that version pinning is still a prevalent quality issue.

[17] https://hub.docker.com/
[18] https://github.com/
[19] https://bitbucket.org/product/
[20] https://www.alpinelinux.org/

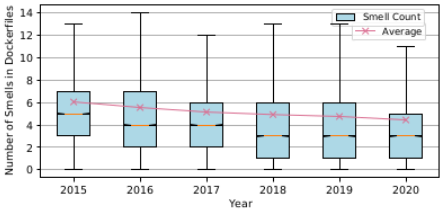

Figure 2.4: The distribution of smell count in Dockerfiles over time (Lin et al., 2020)

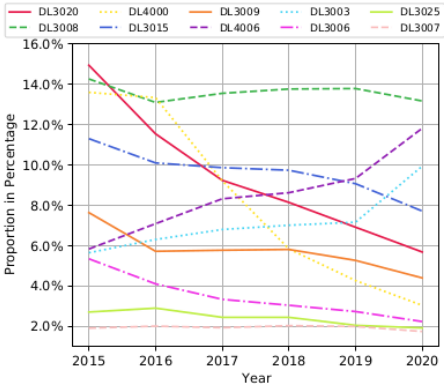

Figure 2.5: Evolution of prevalent Dockerfile smells in popular images (Lin et al., 2020)

Lastly, they found that over time the size of Docker images are shrinking. In figure 2.6 it is shown that the median image size shrinks from roughly 180 MB in 2015 to roughly 110 MB in 2020. The authors attribute this evolution to software practitioners choosing smaller base images and following best practices on writing Dockerfiles. This is in line with research from Ksontini et al., 2021 which found that reducing the image size is among the quality issues that are addressed in Dockerfile changes and refactoring. Other Docker-specific refactorings that they have identified correspond to smells found in the literature and Docker best practices. Durieux, 2023a also found that Dockerfile smells have a significant impact on the size of a Docker image, in particular smells pertaining to package managers.

The latest empirical study performed by (Eng & Hindle, 2021) revisited the Dockerfile ecosystem and set out to reconfirm or debunk previous findings presented by Cito et al., 2017 and Lin et al., 2020 using a dataset consisting of 9,455,938 Dockerfiles. One of these reconfirmed findings is that the RUN instruction is the top used instruction (Cito et al., 2017) and the authors find that 44.94% of these run instructions are dependency related which plays in hand with version pinning issues. 24.44% of the RUN instructions are related with the filesystem such as creating and removing files, as well as navigating to other directories using `cd`, which is a Dockerfile smell.

Whilst Ubuntu is still the most-used image, a continued down-ward trend was observed for pure operating system base images in favour for more light-weight base images with a preconfigured language-runtime or application such as Python, Node, etc... A continued downtrend was also observed for Dockerfile smells overall.

Figure 2.7 from the research paper shows the prevalence of the top 10 Dockerfile smells on the dataset. In (a) it is shown that the incorrect use of the cd command instead foof the `WORKDIR` instruction (DL3003), not using `-o pipefail` (DL4006), and version pinning smells such as DL3018 and DL3013 are becoming more prevalent. Which is in line with the findings of Lin et al., 2020. In (b) it is shown that smells such as version pinning for the `apt` manager (DL3008) has a downward trend but is stagnating. Whilst smells such as incorrectly using ADD instead

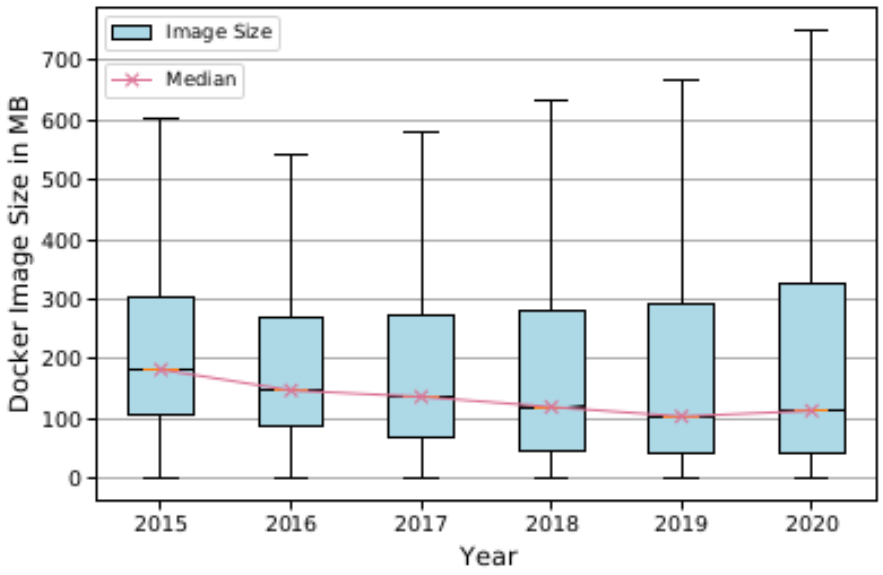

Figure 2.6: The distribution of Docker image sizes in MB over time (Lin et al., 2020)

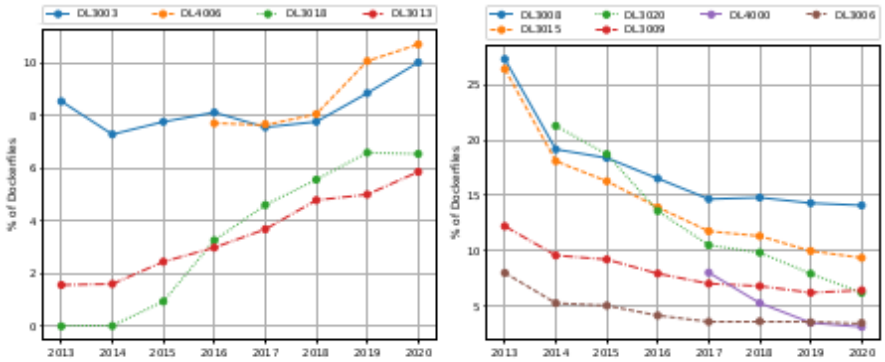

Figure 2.7: Percentage of the top 10 Dockerfile smells:(a) increasing and (b) decreasing (Eng & Hindle, 2021)

of COPY (DL3020), using the deprecated MAINTAINER[21] instruction (DL4000), not version pinning the base image (DL3006), installed recommended dependencies (DL3015) and deleting the `apt-get` cache (DL3009) all show a downward trend.

Despite seeing a downward trend in the prevalence of Dockerfile smells over the years, there is still a growing need for automated tooling that supports developers in writing better Dockerfiles (Cito et al., 2017) (Lin et al., 2020) (Eng & Hindle, 2021) (Wu et al., 2020). This is also reïnforced by Henkel et al., 2020a who found that Dockerfiles on Github voilate best practices

[21] https://docs.docker.com/engine/deprecated/

and rules on average five times more than Dockerfiles written by experts. They did this by mining rules from the Gold Set, which contains 400 Dockerfiles taken from the official Docker Github. These rules were then unleashed on a larger corpus of 178,000 Dockerfiles.

2.4.2 Effect of Docker smells on build time

Earlier work suggests that build times should be less than ten minutes (Laukkanen & Mantyla, 2015). The recent work of Wu et al., 2023 gives us insight how Dockerfiles can attribute to longer build times. Using a filtered subset of the database used in Lin et al., 2020 comprised of 5,833 projects, they found that 48,3% of the builds take more than ten minutes. Figure 2.8 shows the distribution of the build duration for the projects. They also found an upward trend in build time duration evolution over time as shown in figure 2.9 with the medium for 2020 being 11,1 minutes, exceeding the 10 minutes.

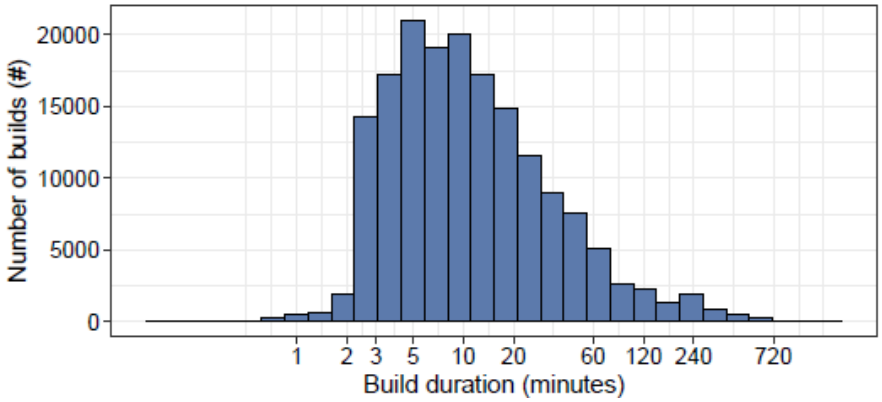

Figure 2.8: Distribution of overall Docker build duration (Wu et al., 2023)

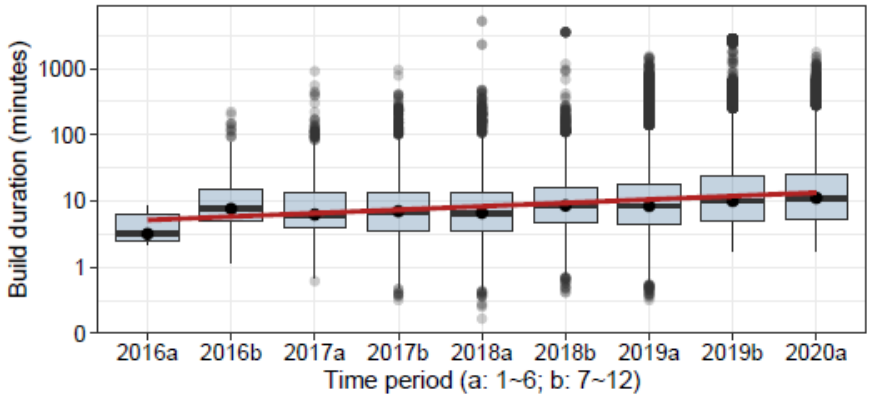

Figure 2.9: Distribution of build durations across time (Wu et al., 2023)

By means of a survey and literature review, the authors derived 27 features across three perspectives (project context, trigger context and Dockerfile context) to build a model to predict the Docker build time duration of a project. Figure 2.10 shows the most important features. 7 out of 10 of those features are related to Dockerfiles. Most notably, the base image (D_3), url tag (D_6) and the `ADD` / `COPY` ratio (D_4), which all have to do with importing resources. Downloading and setting up these large resources can take more time. In earlier work, Huang et al., 2019 created a tool called FastBuild which maintains a local cache of downloaded resources such that subsequent builds need not download them again. Two more other features are of particular interest. Which are the amount of instructions (D_1) and smell density (D_9) that could have an effect on build time. The authors used Hadolint to calculate the smell density.

Group	ID	Feature	Importance*
1	T_5	build_interval_pc	100.000±0.000
2	D_3	base_image	82.167±3.030
3	D_6	url_tag	71.545±2.632
4	P_1	language	54.565±3.787
5	D_4	add_copy_ratio	51.028±3.442
6	D_7	maintainer_tag	45.795±3.066
7	P_2	owner_type	36.669±1.921
8	D_9	smell_density	33.197±3.076
8	D_5	run_ratio	32.706±2.447
9	D_1	instructions	30.430±2.999

*mean and standard deviation

Figure 2.10: Top 10 important features in RF model (Wu et al., 2023)[sic]

2.4.3 Rule mining

Whilst the research in the previous section made use of Hadolint and the Docker best practices, Henkel et al., 2020a and Zhou et al., 2023 both created their own tools, Binnacle and DRIVE, that mine syntactical and semantic Dockerfile rules from a dataset of high quality Dockerfiles. We discuss Binnacle in section 2.5.2 and DRIVE in section 2.5.3. Each tools uses their different data files and have a different approach for defining the proper rules. For example, Henkel et al., 2020a manually went through their gold set of 400 Dockerfiles and by looking at changes made to these files they came up with their own set of rules. Whilst Zhou et al., 2023 hired Docker experts to investigate the rules that were mined, omitting the process of having to double check themselves.

Given the context of the thesis we will show a result with regards to semantic rules mined by DRIVE. In table figure 2.11 shows the 34 rules mined by DRIVE. As these rules are mined, it is possible that news rules can be found which have not previously been described by the literature. The rules with a gray background are these new rules and a voilation of them could have negative effects on the Docker image.

Note that rule 8 states that that compressed files should be removed after unzipping. Whilst this technically is a new rule, or smell, it is important to underline that this smell can also be categorized as a temporary file smell, which will be discussed in the next section

Id	Rule Description	Rule Type	Level	Confidence	Lift
1	apk add using arg. –no-cache	$P \Rightarrow Q$	M	86%	4.43
2	pip install using arg. –no-cache-dir	$P \Rightarrow Q$	M	55%	1.68
3	pip install using requirement.txt	$P \Rightarrow Q$	E	66%	3.48
4	curl using arg. -f	$P \Rightarrow Q$	E	77%	1.39
5	curl with url type https	$P \Rightarrow Q$	M	89%	1.58
6	wget with url type https	$P \Rightarrow Q$	M	82%	1.49
7	git clone with url type https	$P \Rightarrow Q$	E	96%	1.72
8	removing compressed files after unzipping	$P \Rightarrow Q$	M	70%	1.51
9	tar something then remove	$P \Rightarrow Q$	M	64%	1.43
10	gpg using arg. –batch	$P \Rightarrow Q$	E	45%	9.31
11	gpg using arg. –keyserver	$P \Rightarrow Q$	E	45%	9.31
12	gpg using .asc file then remove the .asc file	$P \Rightarrow Q$	E	60%	9.12
13	dnf install using arg. -y	$P \Rightarrow Q$	M	76%	1.57
14	mkdir using arg. -p	$P \Rightarrow Q$	E	61%	1.02
15	chown using arg. -r	$P \Rightarrow Q$	E	61%	0.89
16	rm using arg. -rf	$P \Rightarrow Q$	E	77%	1.63
17	yum install using arg. -y	$P \Rightarrow Q$	M	84%	1.78
18	zypper install using arg. -y	$P \Rightarrow Q$	M	81%	1.72
19	apt-get install using arg. -y	$P \Rightarrow Q$	M	72%	1.53
20	apt-get install using arg. –no-install-recommends	$P \Rightarrow Q$	M	77%	1.63
21	configure using arg. -build	$P \Rightarrow Q$	M	85%	7.83
22	apt-get update prefix apt-get install	$P \Leftarrow Q \Rightarrow R$	M	76%	2.09
23	go build using multi-stage	$P \Leftarrow Q \Rightarrow R$	E	91%	4.47
24	java build using multi-stage	$P \Leftarrow Q \Rightarrow R$	E	72%	6.67
25	clean cache after using conda to install packages	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	M	72%	7.21
26	clean cache after using apt-get/dpkg to install packages	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	M	68%	2.81
27	clean cache after using zypper to install packages	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	M	75%	8.82
28	clean cache after using dnf to install packages	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	M	61%	9.77
29	clean cache after using yum/rpm to install packages	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	M	71%	5.73
30	using sha to verify the downloaded file	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	E	56%	1.54
31	using gpg to verify the downloaded file	$(P_1\|...\|P_n) \Rightarrow (Q_1\|...\|Q_n)$	E	42%	1.32
32	set -eux to print command and quick fail in shell script	*Special*	E	N/A	N/A
33	using useradd to avoid last user to be root	*Special*	E	N/A	N/A
34	using groupadd/addgroup to avoid last user to be root	*Special*	E	N/A	N/A

Figure 2.11: Table with semantic rules mined by DRIVE (Zhou et al., 2023)

2.4.4 Temporary file smells

Lu et al., 2019 were the first to underscore the existence of temporary file smells. They describe these smells as an increase of the image size through human error. A temporary file smells happens when a developer introduces a temporary file in the building process of an image and does not delete it after it is not required anymore.

```
1 FROM centos:7
2 COPY jdk-8u171-linux-x64.tar.gz .
3 RUN tar zxvf jdk-8u171-linux-x64.tar.gz
```

Listing 2.2: Dockerfile without temporary file smell Lu et al

```
1 FROM centos:7
2 COPY jdk-8u171-linux-x64.tar.gz .
3 RUN tar zxvf jdk-8u171-linux-x64.tar.gz
4 RUN rm -f jdk-8u171-linux-x64.tar.gz
```

Listing 2.3: Dockerfile with temporary file smell Lu et al

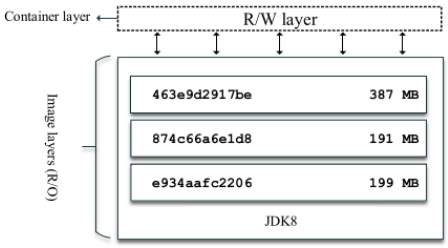

Figure 2.12: Dockerfile layers without temporary file smell (Lu et al., 2019)

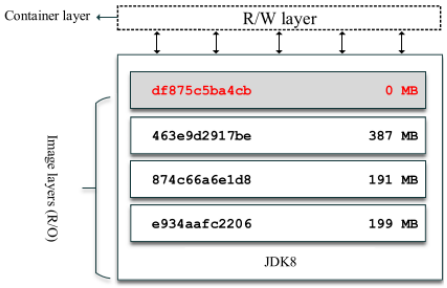

Figure 2.13: Dockerfile layers with temporary file smell (Lu et al., 2019)

The code in listing 2.2 shows when a Dockerfile smell is not present. The Dockerfile starts with the base image `centos:7` and copies a compressed file to image. Then a `RUN` instruction command extracts the file. When we look at figure 2.12 we see the corresponding layers and their sizes. Layer `e934aafc2206` corresponds to the `FROM` instruction, layer `874c66a6e1d8` corresponds to the `COPY` instruction and `463e9d2917be` corresponds to the `RUN` instruction. When we add the sizes of these layers together, the image is 777 MB in size.

The code in listing 2.3 introduces a temporary file smell. The code the same as in listing 2.2 but with the addition of another `RUN` instruction that deletes the file that was introduced by the `COPY` instruction. In figure 2.13 we see that the corresponding layer is 0 MB. Indeed, the file can not be seen in this layer, but when adding the sizes of all the layers together the image still ends up with a size of 777MB. Just like the Dockerfile that did not explicitly delete the file.

This happens because of how Docker builds images, as explained in section 2.2.2. When a new instruction starts, the previous one become read-only and stay that way. Thus removing a file in a later layer than it is introduced, does make the file not visible in the top-level layer, but the file is still there in the low-level layers as shown in picture 2.13 and thus enlarges the Docker image more than necessary.

The authors have identified four temporary file smell patterns:

- `COPY/rm`: introduce a file through the `COPY` instruction and delete it with a `RUN` instruction.
- `ADD/rm`: introduce a file through the `ADD` instruction and delete it with a `RUN` instruction.
- `built-in-cdm/rm`: introduce a file using a built in command such as wget or curl using a `RUN` instruction and delete it later with a `RUN` instruction.
- `no rm`: introduce a file without deletion.

Testing for these smells using a state-dependent static analysis tool, the authors found that 4.69% of 3242 Dockerfiles contains a Dockersmell and thus describe the temporary file smell as an urgent problem in the real world. They also not that this can be taken further as installing tools such as wget can also be considered as enlarging the image unnecessarily if the application within the container does not need it to run.

One way to get around temporary files smells which cannot be avoided is to leverage multi-stage building [22]. Multi-stage building allows software practitioners to use multiple `FROM` instructions. Each new `FROM` instruction starts a new Docker image and this new Docker image will be able to pull resources out of the previous image. This way, a common a pattern is to build an executable in a Docker image by installing dependencies and all other side-requirements and to then start a new image, only install the absolute necessary resources and copy over the executable (or a `.jar` file, for example). One downside is multi-stage building increases the complexity of the Dockerfile and might have a negative effect on the build-time of Docker images.

Listing 2.4 is a modified example found on Alibaba Cloud[23]. In the first stage, an image that comes with Maven is used as the base image. The `pom.xml` and source code is added to the image and then, using the `mvn` command in the `RUN` instruction, it is all packaged into a `.jar` file. The first stage is concluded as the Java executable is built. Then, a new image that comes with Java indicates the start of a new image and the Java executable from the previous image is copied unto the new image, which is then invoked by the `CMD` instruction. This concludes the second stage. This second image is a minimal environment as it will not contain tools like Maven which was required to build the `.jar` file from the source code and the `pom.xml` file containing the dependencies.

```
# First stage: complete build environment
FROM maven:3.5.0-jdk-8-alpine AS builder

# add pom.xml and source code
COPY ./pom.xml pom.xml
COPY ./src src/

# package jar
RUN mvn clean package

# Second stage: minimal runtime environment
From openjdk:8-jre-alpine

# copy jar from the first stage
COPY --from=builder target/my-app-1.0-SNAPSHOT.jar my-app-1.0-SNAPSHOT.jar

EXPOSE 8080

CMD ["java", "-jar", "my-app-1.0-SNAPSHOT.jar"]
```

Listing 2.4: Multi-stage building a Java application

2.4.5 Security smells

Pertaining to the aspect of security in IaC scripts, Rahman et al., 2019 identified seven security smells by analysing 1,726 Puppet scripts:

1. Admin by default
2. Empty password
3. Hard-coded secret

[22] https://docs.docker.com/build/building/multi-stage/
[23] https://www.alibabacloud.com/help/en/acr/use-cases/build-an-image-for-a-java-application-by-using-a-dockerfile-with-multi-stage-build

4. Invalid IP address binding
5. Suspicious comments such as `TODO` or `FIXME`
6. Use of HTTP without TLS
7. Use of weak cryptographic algorithms

These smells can be applied to Dockerfiles, as was done in the recent thesis of Prinetto et al., 2022 in which a tool was developed that detects and fixes certain Dockerfile security smells on the Gold and Standard datasets (Henkel et al., 2020c). This tool managed to fix up to 100% of the security misconfigurations in most cases and clearly shows that security smells are prevalent in Dockerfiles and not covered well by existing detection tools such as Hadolint or Binnacle(Henkel et al., 2020a) for example. Other research classifies suspicious comments such as `TODO` or `FIXME` as self-admitted technical debt (SATD). Azuma et al., 2022 found that the proportion of SATD's in Dockerfiles is 3,4%. The authors identified different identified five categories of SATDS and eleven subcategories of SATD's in Dockerfiles.

The security smells mentioned do not paint the full picture, containers can become vulnerable in other ways too. Related to version pinning and using software packages in general, Zerouali et al., 2019 found that all containers using a `Debian` distribution have high severity vulnerabilities. And if they do get fixed, it takes a long time to do so. As security is a never ending story and vulnerabilities cannot be avoided, they call for more inclusion of vulnerability data in automated tools with regards to Docker.

2.4.6 Configuration smells

Sharma et al., 2016 set out to evaluate configuration code quality and proposed a catalogue of 13 implementation smells, pertaining to formatting and style, and 11 design configuration smells, pertaining to structure or design of the configuration code. To this end, they analysed 4,621 Puppet repositories using a Puppet Linter and a tool called Puppeteer which was mainly used to detect design configuration smells. As Docker is an IaC artifact, we list the implementation smells from the catalogue that are applicable to Dockerfiles.

Implementation smells:

- Complex Expression (ICE)
- Duplicate Entity (IDE)
- Improper Alignment (IIA)
- Incomplete Tasks (IIT)
- Deprecated Statement Usage (IDS)
- Long Statement (ILS)

Note that whilst these smells are applicable to Dockerfiles, some of them have already been implemented or are categorised differently. Hadolint DL4000[24] for example specifies that the `MAINTAINER` instruction is deprecated, which corresponds to the implementation smell of deprecated statement usage. Incomplete Tasks on the other hand, is a smell that could be seen

[24] https://github.com/hadolint/hadolint/wiki/DL4000

as a security smell discussed in section 2.4.5. Incomplete Tasks are described as having tags or comments containing `TODO` or `FIXME`, which can also be considered as vulnerabilities.

2.4.7 Evolutionary trajects of Dockerfiles

Zhang et al., 2018 studied the evolutionary trajectories of Dockerfiles themselves and investigated the impact of those trajectories on the quality of the Dockerfiles. An evolutionary trajectory is the way in which the Dockerfiles evolves over time. The trajectories are as follows:

- C-1: Increasing and holding.
- C-2: Constantly growing.
- C-3: Holding and increasing.
- C-4: Increasing and Decreasing.
- C-5: Holding and decreasing.
- C-6: Gradually Reducing.

Holding means that the size of the Dockerfile stays the same and that changes to the Dockerfiles are comprised of updating environment variables. Increasing means that the Dockerfile size is increased by adding new instructions and more services. Decreasing means reducing the size of the Dockerfile by removing redundant or deprecated instructions and by moving configuration to files. For example, moving `pip` dependencies to a `requirements.txt` file. With these definitions, the *C-1* evolutionary trajectory is that at first the size of the Dockerfile is increased by adding new instructions, configuration and services to the file and after a period of time it is holding as only environment variables are updated (to pin another version of a software package, etc ...). The *C-6* evolutionary traject is that this decrease of the Dockerfile size goes at a slow pace.

With knowing the evolutionary trajectories, Ksontini et al., 2021 investigated reasons as to why Dockerfiles are changed and created a refactorings taxonomy for Dockerfiles. Figure 2.14 show the refactorings taxonomy for Dockerfiles and some of the refactorings have a one on one relation with Dockerfile smells. For example: updating the base image such that size in that regard is kept to a minimum. Replacing the `ADD` instruction with the `COPY` instruction, etc ...

2.5 Dockerfile smell detection approaches

This section discusses smell detection approaches with regards to Dockerfiles found in the literature and discusses Hadolint in section 2.5.1, Binnacle in section 2.5.2, DRIVE in section 2.5.3, temporary file smells in section 2.5.4 and Parfum in section 2.5.5.

Software quality analysis aims to evaluate, improve and assure the quality of software through automated techniques. One of these techniques is program querying in which a query can be asked to a body of code. This query can be formulated in such a way that it describes a bug, or a smell. One of the applications of EKEKO, a meta-programming for Clojure, is to do program and corpus querying (De Roover & Stevens, 2014). Another example is static symbolic execution where the input values are represented by symbols and as such values during the execution can be expressed as symbolic formulas, when for example encountering a branching statement, which

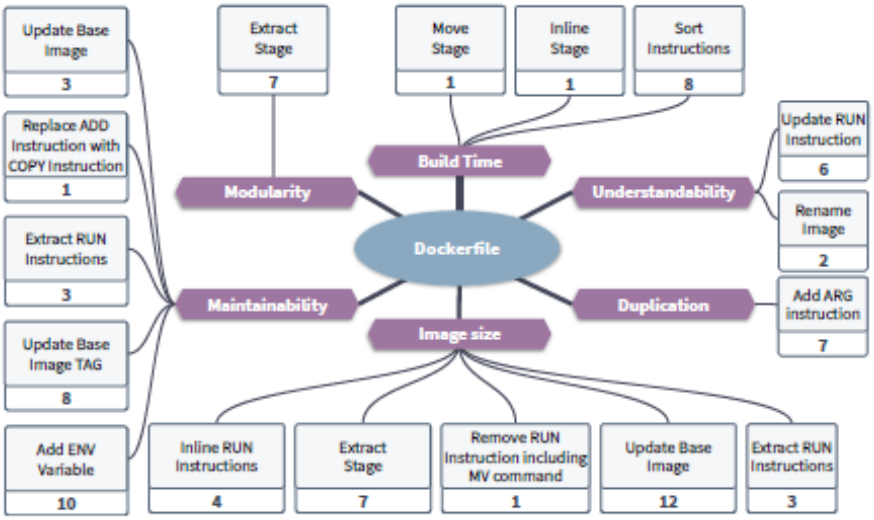

Figure 2.14: Refactorings taxonomy for Dockerfiles (Ksontini et al., 2021)

results in a specific path condition(King, 1976). These path conditions represent constraints and can be solved to adjust inputs for consecutive runs such that each run covers a new path through the application. Sometimes a constraint cannot be solved and for this it is useful to have the application execute and replace the constraint by the actual value of an input at that certain point. This is also known as dynamic symbolic execution or concolic testing.

Static analysis comprises of analysing the source code of an application without running it. Examining this source code can be done in different ways as the source code can be represented differently. The source code can be represented as text, an abstract syntax tree, an intermediate representation, etc. Note that static analysis walks on the edges of the limits of computation, a problem can be decidable when the answer can be computed or undecidable when the answer cannot be computed (Turing, 1937).

Fortunately, static analysis can be used to detect, or compute, some code smells and Sharma and Spinellis, 2018 analysed a large body of research with regards to the code smells and common detection strategies. They identified five categories of smell detection tools:

- **Metrics-based smell detection** takes as input the source code of an application and transforms it to an abstract syntax tree or another source code model. Using specific metrics with a threshold, the detection of a smell happens when a threshold for a certain metric is satisfied.
- **Rules/Heuristic-based smell detection** takes as input the source code of an application and transforms it to an abstract syntax tree or another source code model. Using heuristics or rules, a smell is detected when it satisfies the heuristic or a specific rule.
- **History-based smell detection** is based on the evolution and changes in source code over time.
- **Machine learning-based smell detection** used machine learning algorithms such as

Support Vector Machines to detect code smells. The source code is used to populate a model.

- **Optimization-based smell detection** which uses optimization algorithms and existing examples to detect smells.

2.5.1 Hadolint

Hadolint[25] is an open-source Dockerfile linter and can be categorized under heuristic/rule-based smell detection. It aims to help software practitioners in writing better Dockerfiles by implementing rules over an abstract syntax tree of the Dockerfile. This syntax tree, however, is limited to Docker and does not treat the Shell code inside the `RUN` instruction. It delegates this to ShellCheck[26].

2.5.2 Binnacle

Binnacle is a tool developed by Henkel et al., 2020a whom set to create a semantic, rule-based analyser for Docker after identifying three challenges:

- nested languages
- rule mining
- lack of rule-based analysis

The binnacle toolset also falls under the rule-based smell detectors and has three main functionalities. The first one is to enrich the Dockerfile abstract syntax tree. This can be done because Docker contains a nested language. Indeed, Docker allows Shell code to be written in the `RUN` instruction. The top-level syntax of Docker comprises of a sequence of instructions with their arguments, but for some of these instructions, such as the `RUN` instruction, the arguments can be Shell code and thus have their own syntax. Sometimes, the Shell code itself can contain nested langauges too. It is important to take these nested languages into consideration as they might be able to help detect smells, or contain smells themselves.

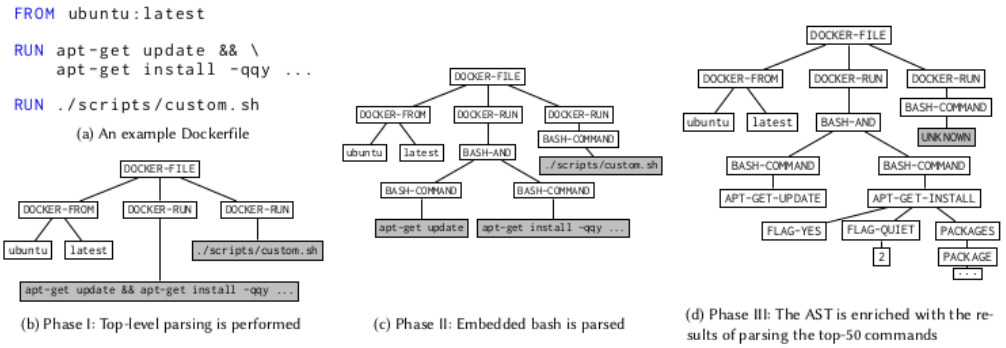

Figure 2.15: Example of a Dockerfile at each of the three phases of phased parsing (Henkel et al., 2020a)

[25] https://github.com/hadolint/hadolint
[26] https://github.com/koalaman/shellcheck

Figure 2.15 shows what this phased parsing looks like. In (a) the Dockerfile is parsed and the abstract-syntax tree for the top-level syntax is visible. The Dockerfile instructions and their arguments. The arguments of the `RUN` instructions are gray as these represent string literals. The authors call these nodes effectively uninterpretable. In (b), the AST is enriched by parsing the Shell code into commands. This, again, leaves the AST with effectively uninterpretable nodes as the commands themselves are string literals. In (c), the authors enrich the AST again with the result of parsing 50 of the most popular commands used in Shell. Almost no effectively uninterpretable nodes are left. Note that these nodes on a deeper level are abstracted and generalised into more general tokens. This allows the AST to encompass and bundle the similarities that exist between package managers for example.

The other two parts comprise of mining rules and enforcing these rules. The rules are mined from a dataset comprised of Dockerfiles that are assumed to be of high quality named the Gold Set and then compared to manually checked rules called the Golden Rules. These rules are then enforced by the enforcer. The power of the Binnacle toolset lies in leveraging the enriched AST to discover and enforce rules that work with these nodes that tools such as Hadolint do not see.

Other work of Henkel et al., 2021 gathers build logs and clusters broken Dockerfiles to find repair rules for broken Dockerfiles based on the build logs.

2.5.3 DRIVE

Dockerfiles **R**ule m**I**ning and **V**iolation d**E**tection, DRIVE, is an novel approach proposed by Zhou et al., 2023. Just like Binnacle(Henkel et al., 2020a) it transforms the source code to an intermediate representation and then using a gold set, DRIVE applies a sequential pattern mining algorithm to this intermediate interpretation in order to discover patterns in these Dockerfiles. The gold set used by the authors consists out of 1,761 Dockerfiles, which is four times larger than the gold set of Binnacle. After applying the pattern, the authors were mainly interested in enforcing the mined semantic rules using their own detection algorithm.

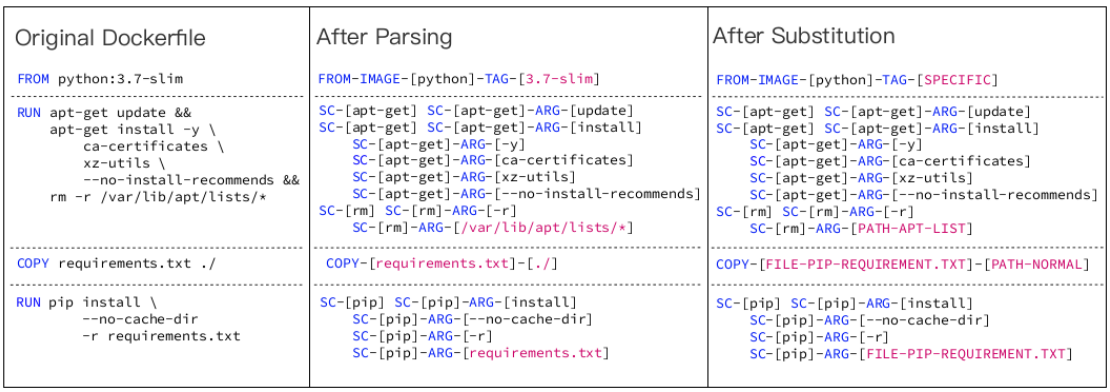

Figure 2.16: Before/After parsing and substitution (Zhou et al., 2023)

Figure 2.16 gives us insight in the intermediate representation of DRIVE. After parsing, the Dockerfile is transformed into an AST which corresponds not only to the top-level syntax of the Dockerfile, but also Shell commands and their arguments. Then, the substitution phase happens

in which some nodes of the AST are substituted for more general, pre-defined tokens. For example, the last `RUN` instruction installs Python libraries using a file called `requirements.txt` is replaced by a token `FILE-PIP-REQUIREMENT.TXT`.

2.5.4 Temporary File smell detection

In 2.6.2 we have discussed the temporary file smells. In order to show the prevalence of temporary file smells in Dockerfiles, Lu et al., 2019 approached the problem with a state-dependent static analysis.

This state-dependent static analysis traverses over the Dockerfile AST. Before the analysis, the AST is pruned. This is done by marking Dockerfile nodes with a colour. In figure 2.17 we see a Dockerfile and in figure 2.18 we see its corresponding AST. The colours classify the following nodes:

- Red: operator nodes
- Blue: resource nodes
- Green: cmd nodes
- Purple: parameter nodes

The authors assume that the Shell scripts, which comprimise of resource nodes and parameters nodes, run as the developers intended and thus the AST is pruned by removing all the parameter, purple, nodes.

The analysis itself utilizes a state table to prevent ambiguity. There can be ambiguity when a resource or its path need to be determined. This state table keeps track of the current directory of the container, which can be changed by either using a `WORKDIR` instruction or by using the `cd` command in a `RUN` instruction. Note that the latter is a Dockerfile smell on multiple levels. It goes against best practices and in general makes the Dockerfile less readable and maintainable as the current directory will not be clear. On top of the current file directory, the state table also keeps track of filepaths and directory paths, indicated by F and D respectively. For paths that are unclear, U is used. The state table is used by the semantic analyzer that analyses the AST. The result is then used to detect temporary file smells. Note that this tool was node made available for verification or to reproduce the results.

Xu et al., 2019 expanded this research by applying a dynamic analysis of the system when building a dockerfile to detect temporary file smells. The authors injected logging code in the Overlay filesystem and XFS filesystem of CentOS 7. This way, the system will generate a logs during building Docker images from Dockerfiles and specify exactly which filepath in which layer was added and deleted. If the layers of an identical filepath are different it classifies as a temporary file smell.

Whilst the authors point out that this can theoretically detect all the temporary file smells with the exception of `no rm`, injecting logging code into kernel yields a lot of logs and could hurt performance up to 37.1%, which is not practical.

Dockerfile 5: JDK1.8

```
1: FROM centos:7
2: RUN wget http://download.oracle.com/otn-pub/
   java/jdk/8u171-b11/512cd62ec5174c3487ac17c
   61aaa89e8/ jdk-8u171-linux-x64.rpm
3: RUN rpm -ivh jdk-8u171-linux-x64.rpm
4: RUN rm -f jdk-8u171-linux-x64.rpm
```

Figure 2.17: Dockerfile 5 (Lu et al., 2019)

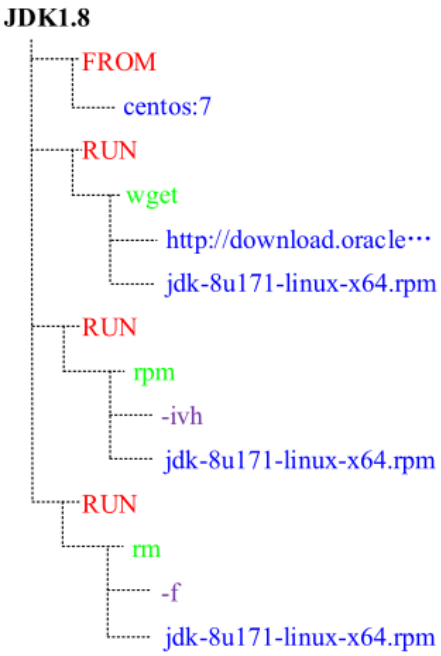

Figure 2.18: Dockerfile 5 coloured AST (Lu et al., 2019)

2.5.5 Parfum

The recent work of Durieux, 2023a is the closest to this thesis. Durieux, 2023a set out to fill the gap that exists with regards to tools detecting and repairing Dockerfile smells. Parfum, a heuristic/rule-based smell identifier and program repair tool, is build on a new library called Dhingy [27]. Dhingy uses the `dockerfile-ast` and `mvdan-sh` libraries to create a unified abstract syntax tree that encompasses both the top-level syntax of Docker (`dockerfile-ast`) and the low-level syntax of the embedded Shell code (`mvdan-sh`). This unified AST also provides functionality such as navigating and modifying the tree, annotating nodes, queries and printing. Printing in the sense of converting the, potentially modified, AST back into a Dockerfile.

The Parfum tool relies on this Dhingy library and leverages its query and annotation capabilities. Before running detection rules defined by queries, Parfum enriches the AST by annotating extra information about the shell code. Figure 2.19 shows us the annotated AST of a Dockerfile for that has a `RUN` instruction. It has an `Invocation` node that annotated with `apt install` and a `Arg` node with the string literal `wget` which is annotated with package. This example thus shows how these commands can be generalised and follows the same idea of enriching the AST as introduced by Henkel et al., 2020a.

With the AST enriched, parfum runs predefined queries on the AST. If a smell is detected, the tool will look for a repair. If this repair is available, the repair will modify the AST tree. When this is done for all the predefined queries, the AST is converted to the repaired Dockerfile.

[27] https://github.com/tdurieux/Dinghy

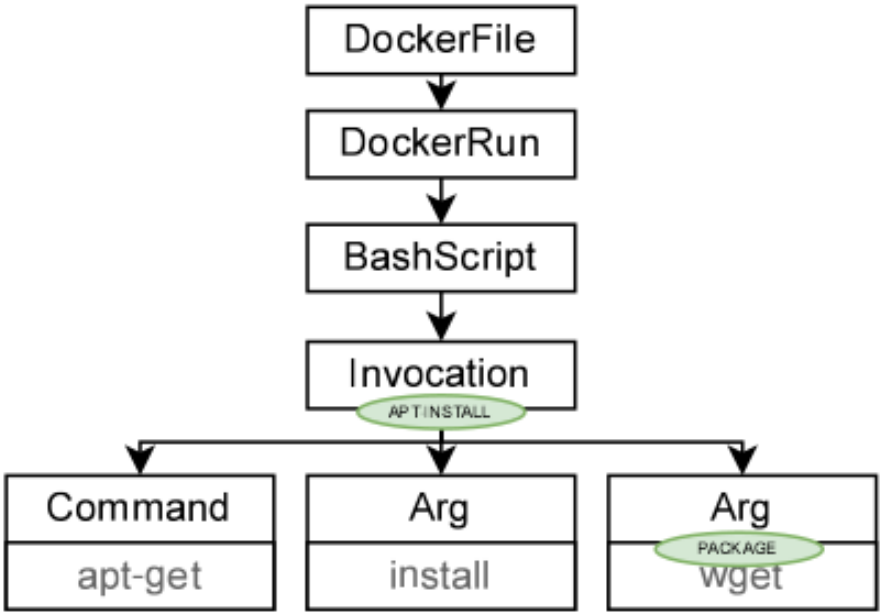

Figure 2.19: Dhingy AST annotated with information of the `apt` command (Durieux, 2023a)

2.6 Best practices for minimizing image size

In this section we present the Dockerfile smells gathered from the research in earlier sections that bloat the Docker image as we focus on these kind of smells in this thesis. The smells are formatted in the same as the smells catalogue which is presented in chapter 3. Whenever a fix is possible, it is included. All the Dockerfile smells identified during this thesis can be consulted in the Dockerfile smells catalogue which can be found in appendix A of this thesis. Specifically, Dockerfile smells pertaining to caching issues are discussed in section 2.6.1, Dockerfile smells pertaining to temporary files are discussed in section 2.6.2 whilst section 2.6.3 showcases some of the Dockerfile smells that stand out on their own.

2.6.1 Keeping cache

In this section, we present the Dockerfile smells that occur when cache features of tools such as package managers are leveraged. Whilst in generally cache systems are useful, they are particularly nasty for containers as these containers should be as light as possible. These smells are repetitive in nature as package managers use specific commands to clear their cache, therefore we refrain from writing them all out. All these smells can be consulted in appendix A.

DL3009

Code: DL3009
Message: Delete the apt-get lists after installing something using apt-get
Category: BLOATER
Rationale: Removing `/var/lib/apt/lists` helps keeping the image size down. Removing this folder must be performed in the same RUN step, otherwise it will affect image size negatively as

files that should not be in the image are included in the image.
Problematic Code:

```
RUN apt-get update && apt-get install --no-install-recommends -y python
```

Correct Code:

```
RUN apt-get update && apt-get install --no-install-recommends -y python \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Sources: Hadolint Wiki

DL3019

Code: DL3019
Message: Use the `--no-cache` switch for `apk`
Category: BLOATER
Rationale: In Alpine Linux 3.3 there exists a new `--no-cache` option for `apk`. It allows users to install packages with an index that is updated and used on-the-fly and not cached locally. Avoids the need to use `--update` and remove `/var/cache/apk/*` when done installing packages.
Problematic Code:

```
FROM alpine:3.7
RUN apk update \
    && apk add foo=1.0 \
    && rm -rf /var/cache/apk/*
```

Correct Code:

```
FROM alpine:3.7
RUN apk --no-cache add foo=1.0
```

Sources: Hadolint Wiki, docker-alpine disabling cache

DL3032

Code: DL3032
Message: `yum clean all` missing after yum command.
Category: BLOATER
Rationale: Clean cached package data after installation to reduce image size. Cleaning the cache must be performed in the same RUN instruction, otherwise it will not affect image size.
Problematic Code:

```
RUN yum install -y httpd-2.24.2
```

Correct Code:

```
RUN yum install -y httpd-2.24.2 && yum clean all
```

Sources: Hadolint Wiki, Dockerfile best practices

DL3036

Code: DL3036
Message: `zypper clean` missing after zypper use
Severity: WARNING
Category: BLOATER
Rationale: Reduce layer and image size by deleting unneeded caches after running zypper.
Problematic Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd=2.4.46
```

Correct Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd=2.4.46 && zypper clean
```

Sources: Hadolint Wiki

2.6.2 Temporary file smells

In this section we present Dockerfile smells pertaining to temporary file smells as discussed in section 2.4.4. Whilst these look similar, the differences are subtle and warrant them being their own smell. As mentioned, we do not give the full list here. All the smells can be consulted in append A.

DL3010

Code: DL3010
Message: Use ADD for extracting archives into an image.
Category: BLOATER
Rationale: The best use for ADD is local tar file auto-extraction into the image. This will also prevent possible TF code smells.
Problematic Code:

```
COPY rootfs.tar.xz <path>
RUN <extract rootfs.tar.xz>
```

Correct Code:

```
ADD rootfs.tar.xz <path>
```

Sources: Hadolint Wiki

DL9014

Code: DL9014
Message: COPY/rm pattern
Category: BLOATER
Rationale: Temporary file smells are always accompanied by a form of transforming operation such as uncompressing. Copying an archive file and then extracting it introduces temporary files in the image that cannot be deleted, even if attempted to do so in later layers.The ADD instruction can directly uncompress them in the importing process and is a good alternative that

alleviates this problem.
Problematic Code:

```
FROM centos:7
COPY jdk-8u171-linux_x64.tar.gz .
RUN tar x jdk-8u171-linux_x64.tar.gz
RUN rm -f jdk-8u171-linux_x64.tar.gz
```

Correct Code:

```
FROM centos:7
ADD jdk-8u171-linux_x64.tar.gz .
```

Sources: Xu et al., 2019

DL9015

Code: DL9015
Message: build-in-cmd/rm pattern
Category: BLOATER
Rationale: Commands such as wget can introduce temporary files which are required to be deleted within the same layer. There are many variations since there are many different ways to import temporary files into an image. This pattern has a common feature such as the import and deletion of temporary files in different image layers. The defined smell is shown with a wget example.
Problematic Code:

```
FROM centos:7
RUN wget http://download.oracle.com/.../jdk-8u171-linux-x64.rpm
RUN rpm -ivh jdk-8u171-linux-x64.rpm
RUN rm -f jdk-8u171-linux-x64.rpm
```

Correct Code:

```
FROM centos:7
RUN wget http://download.oracle.com/.../jdk-8u171-linux-x64.rpm \
    && rpm -ivh jdk-8u171-linux-x64.rpm \
    && rm -f jdk-8u171-linux-x64.rpm
```

Sources: Xu et al., 2019

2.6.3 Other

This section contains some examples of other smells that pertain to bloating images. These smells are not as repitive in nature as the smells shown in section 2.6.1 and section 2.6.2 .As mentioned in the previous sections, all the smells can be found in appendix A of this thesis.

DL3015: Avoid recommended packages

Code: DL3015
Message: Avoid additional packages by specifying `--no-install-recommends`.
Category: BLOATER
Rationale: Avoid installing additional packages that you did not explicitly want. If they are needed, add them explicitly.
Problematic Code:

```
FROM busybox:<tag>
RUN apt-get install -y python=2.7
```

Correct Code:

```
FROM busybox:<tag>
RUN apt-get install -y --no-install-recommends python=2.7
```

Sources: Hadolint Wiki, Dockerfile reference

DL3059: Consolidate multiple consecutive RUN instructions

Code: DL3059
Message: Multiple consecutive `RUN` instructions.
Severity: INFO
Category: BLOATER
Rationale: Each `RUN` instruction will create a new layer in the resulting image. Therefore, consider consolidation as it this will reduce the layer count and reduce the size of the image
Problematic Code:

```
# big files stored in image layer!
RUN command_creating_big_files
RUN command_deleting_these_files
```

Correct Code:

```
RUN command_creating_big_files \
 && command_deleting_these_files
```

Sources: Hadolint Wiki, Docker development best practices

DL4001: Either use Wget or Curl

Code: DL4001
Message: Either use Wget or Curl, but not both.
Category: BLOATER
Rationale: Do not install two tools that have the same effect and keep your image smaller than it would be.
Problematic Code:

```
FROM debian:<tag>
RUN wget http://google.com
RUN curl http://bing.com
```

Correct Code:

```
FROM debian:<tag>
RUN curl http://google.com
RUN curl http://bing.com
```

Sources: Hadolint Wiki

2.7 Background Conclusion

This section concludes this chapter and discusses the gaps in the literature in section 2.7.1 and provides a summary of the chapter in section 2.7.2.

2.7.1 Gap in the literature

In this section, we identify several gaps that we have found in the literature with regards to Dockerfile smells.

The first gap is that there is no consolidation of the Dockerfile smells. The research on Dockerfile smells is definitely an ongoing trend in the literature. Researchers such as Cito et al., 2017, Lin et al., 2020 and Eng and Hindle, 2021 have used Hadolint to map the prevalence of Dockerfile smells and highlight the impact of these smells on Dockerfile quality, build time and size of the Docker image. Other research such as Henkel et al., 2020a and Zhou et al., 2023 use high quality Dockerfile to mine rules and violations of these rules can be considered as going against best practices and thus can be described as a Dockerfile smell. Lu et al., 2019 identifies a specific kind of smell called the temporary file smell. In other research, security and configuration smells were identified which are applicable to Dockerfiles as well. However, to the best of our knowledge, there no consolidation of all these Dockerfile smells which could in turn both help researchers and academics writing better tools and examine Dockerfiles.

The second gap we identify is that the empirical research on the prevalence and the effect of Dockerfiles has mostly been done by Hadolint. Whilst we encourage software practitioners to use Hadolint as much as possible while writing Dockerfiles, Hadolint itself gives an incomplete overview when used to describe the Dockerfile smells landscape. One of the limitations of Hadolint is explained in by Henkel et al., 2020a, in which is pointed out that Hadolint does only parse the top-level Docker syntax, but not the underlying nested Shell language. Additionally, it does not have the tools in place to detect more complex smells such as temporary file smells which requires some form of static analysis. We argue it is useful to build on top of this research and shine some light on smells that have not been discussed yet.

The third gap we identify is that, to the best of our knowledge, it is known that layer-optimization is useful and can bring benefits to the Docker image, no specific and concrete layer-optimization smells have been identified so far.

The fourth gap is that none of the Dockerfile research has a focus on the Dockerfiles that are found on StackOverflow. It has been shown that StackOverflow is extensively used by developers and that the quality of code snippets is understudied which calls developers to use caution when using these code snippets for their solutions (Meldrum et al., 2020). We argue that it is not unreasonable to investigate the quality of Dockerfiles is on StackOverflow as bad practices might seep from StackOverflow into Dockerfiles.

The last gap is that research calls out for more automated tools pertaining to Dockerfiles and Docker in general. Whilst there definitely are tools out there, there is no accessible tool that detects layer-optimisation smells and temporary file smells. Whilst Xu et al., 2019 have created a tool with regards to temporary file smells, we were not able to find it.

2.7.2 Summary

We started this chapter of in section 2.1 by explaining virtualization techniques such as virtual machines and containers. Giving a general overview of how they work and how they should not be seen as substitutes but as complementary technologies that form the basis of cloud computing when used in tandem.

In section 2.2 we presented the most popular containerization technology and tool at the time of writing: Docker. The client, daemon and registry are essential components of its architecture with the Docker image being central in the ecosystem. This Dockerimage can be build from a Dockerimage which is a set of instructions that allows developers to model the environment for their containers using this declarative language.

Continuing on, we explained what a code smell is in section 2.3 and how this relates to Dockerfiles, as well as the possible effects on aspects such as incremental build time, image size, maintainability and security.

Subsequently, section 2.4 dove into literature on Dockerfile smells which revealed that code smells are still prevelant in Dockerfiles with some smell becomes less prevelant over time, others becoming more prevalent over time. It also shown that some of these prevelant smells definitely have an impact om image size, such as package mananger cache functionality. Some smells were discovered using rule mining techniques, others were found by exploring aspects of other IaC langauges pertaining to configuration and security. Lastly, temporary file smells occur when a file is introduced in a layer and deleted in a later layer. How these smells were found and thus detected was discussed in section 2.5.

Lastly, we present some of the smells found in the literature with an effect on image size in section 2.6. These smells have to do with the caching functionality of package managers, temporary file smells and some other smells such as not using different tools for the same purpose.

Chapter 3

Catalogue

In this chapter we discuss the catalogue compiled with, to the best of our knowledge, all the Dockerfile smells found in the literature. First, we summarize why this catalogue was necessary in section 3.1. Then we present the the format which is used to present the Dockerfile smells in the catalogue in section 3.2 followed by explaining the structure of the catalogue in section 3.3 with examples. We summarize the chapter in section 3.4. The entire catalogue can be consulted in the appendix A.

3.1 Motivation

Code smells that are applicable to Dockerfiles are scattered throughout the literature. There are open-source tools such as Hadolint("Haskell Dockerfile Linter", 2023) which includes Dockerfile smells identified by reading best practices on writing Dockerfiles("Best practices for writing Dockerfiles", 2023) and software practitioners by experience. Much of the empirical research on the prevalence of Dockerfiles such as Cito et al., 2017, Wu et al., 2020 and Eng and Hindle, 2021 all make extensive use of Hadolint. Henkel et al., 2020a and Zhou et al., 2023 discover new smells by mining rules from Dockerfiles that are considered to be high quality. Lu et al., 2019 identified the four different kinds of temporary smell files which has to do with the copy-on-write mechanism. Prinetto et al., 2022 applied IaC security smells identified in Rahman et al., 2019 to Dockerfiles and Sharma et al., 2016 identified configuration smells in configuration code which are also applicable to Dockerfiles.

In addition to these code smells being scattered throughout the literature, there are several Dockerfile smells that are an instance of the same, more general, smell. For example, Not pinning the version of software packages and base images is one of the most prevalent Dockerfile smells. Version pinning is important with regards to Dockerfiles as you specify which exact version of a software package allows your application to run without issues. Not pinning the version can and will introduce bugs or in the worst case break the application over time. Throughout the literature, we identified several smells that are an instance of this more general smell such as version pinning for base images and several package managers such as `apt-get`, `npm`, `pip`, etc. To the best of our knowledge, this kind of categorisation is not made explicitly yet and could help identify code smells of this nature as they are inherently not a fixed set. New tools and package managers are released frequently and thus the set of version pinning smells will only grow as essentially every package manager, or command-line tool, that enables version pinning can be added as a concrete instance of the version pinning smell. This line of thought can also

be expanded to things such as cache management, yes-flags, recommended dependencies, etc.

A consolidation of all the identified Dockersmells (both general and concrete), to the best of our ability and time, could assist and save time for both software practitioners and researchers in writing better Dockerfiles, recognising undocumented smells, building better tools and performing research on Dockerfile smells.

3.2 Dockerfile smell format

The Dockerfile smells in the catalogues are described using a format inspired by the Hadolint wiki[1]. We describe the attributes:

- **Code**: This is a code which serves as an identifier. This thesis adopts the same format as used by Hadolint but will use DL9000 as basis for smells that are not covered by Hadolint.
- **Message**: This is a message which gives a top-level description of the code smells.
- **Category**: This is the category to which the smells belong to as descirbed in section 3.3.
- **Rationale**: The rationale explains why this is considered a smell.
- **Problematic Code**: The problematic code is a Dockerfile code snippet which contains the Dockerfile smell.
- **Correct Code**: The correct code is a Dockerfile which exerts the same behaviour as the problematic code but without the possible negative side-effects.
- **Source(s)**: Cites the sources where the smell was identified.

Code: DL3008
Message: Pin versions when installing software packages using apt-get.
Category: MAINTAINABILITY
Rationale: Version pinning forces the build to retrieve a particular version regardless of what is in the cache. This reduces failure due to unanticipated changes in required packages.
Problematic Code:

```
FROM busybox
RUN apt-get install python
```

Correct Code:

```
FROM busybox
RUN apt-get install python=2.7
```

Sources: Hadolint Wiki, [1]

Figure 3.1: DL3008 from the smell catalogue

Figure 3.1 shows a concrete example from the catalogue. Note that the correct code does not necessarily contain pristine Dockerfile code. Whilst the version pinning for apt-get is indeed

[1] https://github.com/hadolint/hadolint/wiki

correct now, there is still various other smells present such as other version pinning smells, cache management, etc. For clarity, these are omitted such that the focus is just on the smell being discussed.

3.3 Structure

The Dockerfiles smells are divided into four different categories which comprises the structure of the catalogue. Some of these categories have subcategories which are a general smell, a smell of a certain kind, which has multiple concrete implementations. For each category, we discuss one to two examples.

3.3.1 Maintainability

These are the Docker smells that go against the best practices of Dockerfiles. In general, these are smells that make Dockerfiles less maintainable and could even cause applications to introduce bugs or break down over time. We discuss two examples to make sure the range and extend of these kind of smells is known.

Code: DL3003
Message: Use WORKDIR to change working directory.
Category: MAINTAINABILITY
Rationale: Docker provides the WORKDIR instruction if you need to change the current working directory
Problematic Code:

```
1 FROM busybox:<tag>
2 RUN cd /usr/src/app && git clone git@github.com:
    lukasmartinelli/hadolint.git
```

Correct Code:

```
1 FROM busybox:<tag>
2 WORKDIR /usr/scr/app
3 RUN git clone git@github.com:lukasmartinelli/hadolint.git
```

Sources: Hadolint Wiki,[1]

Figure 3.2: DL3003 from the smell catalogue

In figure 3.2 we describe a smell where the `cd` command in the `RUN` instruction is used to change the working directory instead of the `WORKDIR` instruction. The sole purpose `WORKDIR` instruction is to make it explicitly clear when the working directory is changed. Doing it through the `cd` instruction could introduce bugs as it is less readable and could throw off other software practitioners who do not see files appearing in the path which was specified by the `WORKDIR` instruction.

In figure 3.3 we describe a smell in which we restrict the ports that can be used within a Dockerfile using the `EXPOSE` instruction as it does not make much sense to use numbers that are not in the range of valid ports.

This category also contains two more general smells which have concrete implementations:

- Version pinning

Code: DL3011
Message: Valid UNIX ports range from 0 to 65535.
Category: MAINTAINABILITY
Rationale: Any port outside of this range does not make any sense.
Problematic Code:

```
1 FROM busybox:<tag>
2 EXPOSE 80000
```

Correct Code:

```
1 FROM busybox:<tag>
2 EXPOSE 65535
```

Sources: Hadolint Wiki, List of TCP and UDP port numbers

Figure 3.3: DL3011 from the smell catalogue

- Interaction prevention

We have explained version pinning in section 3.1. Interaction prevention is another general smell related to command-line tools which demand user input, which cannot be given when building a container. Some command-line tools allow users to assume input using a flag (most commonly the `-y` flag). Therefore, it is important to always include this flag when possible such that the container does not wait for user input which it will never receive.

3.3.2 Bloaters

These are Dockerfile smells that go against the best practices of Dockerfiles and/or have a nature of increasing the size of the image. As mentioned, containers and images should be as light-weight as possible and thus anything that makes the Docker image larger than it could, and should, be can be considered a bloater. This category contains two general smells which have concrete implementations:

- Keeping cache
- Temporary files

Keeping cache has to do with the cache that is kept by package managers. Whilst caching mechanisms are useful in traditional software development, they are an extra burden to Dockerfiles because there is no need to keep these files around and therefore they should be deleted from the image in the correct layer. The second general smell, temporary files, have been discussed in section 2.6.2.

Figure 3.4 is an example of a bloater in which there are multiple RUN instructions in sequence. Both `RUN` instructions contain a single shell command. In order to reduce the amount of layers, and thus the image size, we can join these two single `RUN` instructions into one single `RUN` instruction by chaining the commands using the `&&` operator.

Figure 3.5 is a smell that is not easy to solve computationally but regardless is an important one to keep in mind. A full fledged OS image is larger than a minimal one and thus the effects on the image size can be significantly reduced by aiming for the smallest base image possible.

Code: DL3059
Message: Multiple consecutive RUN instructions.
Category: BLOATER
Rationale: Each RUN instruction will create a new layer in the resulting image. Therefore, consider consolidation as it this will reduce the layer count and reduce the size of the image
Problematic Code:

```
# big files stored in image layer!
RUN command_creating_big_files
RUN command_deleting_these_files
```

Correct Code:

```
RUN command_creating_big_files \
 && command_deleting_these_files
```

Sources: Hadolint Wiki, Docker development best practices

Figure 3.4: DL3059 from the smell catalogue

Code: DL4001
Message: Use the smallest base image possible with FROM
Category: BLOATER
Rationale: Smaller base image results in a smaller container. For example: use Alpine instead of Debian.
Problematic Code:

```
FROM debian:<tag>
```

Correct Code:

```
FROM alpine:<tag>
```

Sources: [1]

Figure 3.5: DL4001 from the smell catalogue

3.3.3 Security

These are Dockerfile smells that go against the best practices of Dockerfiles and/or have a nature of exposing or leaving the container vulnerable. This ranges from user-permissions in the container to hard-coding secrets in environment variables.

Figure 3.6 describes a smell which states that the last user should not be a root user as this would leave the container vulnerable. It adheres to the principle of least privilege (Saltzer & Schroeder, 1975) in order to minimize the impact from an attack.

3.3.4 Layer-optimization

These are Dockerfile smells that go against the best practices of Dockerfiles and/or have a nature of not optimizing the cache mechanism of the Docker layers. Note that that this cache mechanism of Docker layers is not to be confused with the cache in bloater smells. That is cache related to the image which we would like to keep as small as possible. The idea behind layer optimization

Code: DL3002
Message: Last user should not be root
Category: SECURITY
Rationale: Switching to the root USER opens up certain security risks if an attacker gets access to the container. In order to mitigate this, switch back to a non-privileged user after running the commands you need as root.
Problematic Code:

```
FROM busybox:<tag>
USER root
RUN ...
```

Correct Code:

```
FROM busybox:<tag>
USER root
RUN ...
USER guest
```

Sources: Hadolint Wiki,Dockerfile best practices

Figure 3.6: DL3002 from the smell catalogue

is that we only bust the (layer) cache and thus rebuild the image from a point where it is only logical and required to do so.

It is worth pointing out that possible solutions for these kind of smells usually increase the amount of layers. At the surface, this is in conflict with the general advise of keeping the amount of layers to a minimum. However, we think that these are an exception to that advice as faster subsequent build times is more valuable than two or three extra layers.

Figure 3.7 lists as problematic code a Dockerfile which has a base image with node installed, copies the entire build context using the `COPY` instruction and then installs the dependencies with a `RUN` instruction to finally run the NodeJS application using the `CMD` instruction. This is a problem as the source code is likely to change a lot more than the dependencies, therefore it is advised to copy and install the dependencies before copying the source code. As such, subsequent builds will leverage the cache and thus the image need not install the dependencies so long they are not modified and yield faster build times for these subsequent images.

3.4 Chapter summary

In this chapter we introduced the catalogue which was proposed in the approach. In section 3.1 we motivated again why this catalogue fills a gap: Dockerfile smells scattered throughout the literature and a distinction between more general smells and concrete smells. Then we presented the format of a Dockerfile smell which the catalogue adopts in section 3.2. It lists the code, category, a brief description, rationale, problematic code, correct code and cites the source where it was originally found. Then we described the structure of the catalogue in section 3.3 which is comprised of the categories: maintenance, bloaters, security, layer-optimisation.

Code: DL9000
Message: Using npm, install dependencies before copying source code
Category: LAYER-OPTIMIZATION
Rationale: The `npm install` command can take up a significant amount of time when building a Dockerfile. Whilst this is unavoidable when building a Docker image for the first time, you can significantly decrease incremental build times by leveraging the Docker cache by copying the source code from the build context to the images after installing the dependencies. This way, changes to the source code will not cause the image to install the dependencies in package.json.
Problematic Code:

```
FROM node:<tag>
COPY . .
RUN npm install
CMD ["node", "index.js"]
```

Correct Code:

```
FROM node:<tag>
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "index.js"]
```

Sources:

Figure 3.7: DL9000 from the smell catalogue

Chapter 4

Implementation

Here comes my Implementation

4.1 Motiviation

4.2 Overview

4.3 Detecting Dockerfile smells

4.3.1 Detecting Package managers

Here comes my representation for bashcommands

4.3.2 Detecting temporary file smells

Here comes my analyser for temporary files

4.3.3 Detecting layer-optimization smells

Here comes my analyser for layer-optimization smells

4.3.4 Reporting

Logs

SmellBox

4.4 Limitations

Here comes my Limitations

Chapter 5

Evaluation

In this chapter we evaluate the result of our conducted experiments In section 5.1 we reiterate the research questions and objectives of the thesis. The data used in the conducted experiments is explained in section 5.2. In section 5.3 we describe the results of our experiments related to each research question. The results of the experiments are then discussed and coupled back to the research questions in section 5.4. Lastly, we discuss both the internal and external validity of the thesis in section 5.5.

5.1 Research Questions

Our thesis sets out to answer the following research questions, which were also given in section 1.2.1:

- **RQ1**: What is the current state of Dockerfile smells?
- **RQ2**: Does our tool perform well compared to the state-of-the-art?
- **RQ3**: How prevalent are the detected smells in the GitHub ecosystem?
- **RQ4**: How prevalent are the detected smells in the StackOverflow ecosystem?

5.1.1 Experimental set-up

The experiments were conducted on a custom build machine, of which the details can be found in table 5.1. Note that Intel's Turbo Boost Technology is enabled. This means that the base clock speed increases when the CPU notices that tasks are demanding. Which means that the CPU could have been running at 3.70GHz for some experiments, and at 4.30GHz, for example, for other experiments.

5.2 Data

This section describes the data that was gathered and used for conducting the experiments of which the results are described in section 5.3.

Machine	
Model	Custom build
CPU	Intel® Core™ i7-8700K Processor
RAM	16 GB
Processor Specs	
Cores	6
Logical Processors	12
Base Clock Speed	3.70 GHz
HyperTreading Technology	Enabled
Intel Turbo Boost Technology	Enabled
L1 cache	384 KB
L2 cache	1,5 MB
L3 cache	12,0 MB
Erlang specs	
Version	12.3
OTP	24
HiPE	Yes
OS Specs	
Edition	Windows 10 Education
Version	20H2
OS build	19042.1586

Table 5.1: Specification

(Schermann et al., 2018)
Whilst there are

5.2.1 Ground truth dataset

Here comes my ground truth dataset

5.2.2 Merged dataset

The merged dataset is comprised of two already existing datasets: the Binnacle dataset(Henkel et al., 2020b) introduced by Henkel et al., 2020c and the Parfum dataset(Durieux, 2023b) introduced by Durieux, 2023a. The Binnacle dataset from Henkel et al., 2020c contains 178,505 unique Dockerfiles which were mined from GitHub from repositories with at least ten stars between 2007 and 2019 which could be parsed by a Dockerfile parser. The Parfum dataset from Durieux, 2023a was also mined from GitHub. Akin to the Binnacle dataset it gathered repositories from 2022 with at least ten stars, but with the addition of having at least 50 commits. Of this dataset, we were able to extract 145,478 Dockerfiles of the supposed 193,948 Dockerfiles.

The two datasets were merged by hashing the contents of the deduplicated Binnacle dataset using SHA256. Then, for each Dockerfile in the deduplicated Parfum dataset, the content was hashed using SHA256 and compared to hashes of the Binnacle dataset. If any hashes match, the Dockerfile was omitted. This way, our merged dataset comprises of 323,347 unique Dockerfiles.

5.2.3 StackOverflow dataset

Here comes the description of the StackOverflow dataset, ask advisors on how Camilo scraped this so that I can describe it properly.

5.3 Results

In this section we present our findings and use them to address our research questions in the order which were given in section 5.1. In the results we often mention the code of the smell. A smell code just by itself does not contain much information. Therefore, table 5.2 gives an overview of the detected smells with their smell code which can be used as legend when reading the results. The legend is divided to mimic the order of the smells in the Dockerfile smell catalogue.

5.3.1 What is the current state of Dockerfile smells

In order to answer RQ1 we created a catalogue which was discussed in chapter 3. In order to gather all the Dockerfile smells which comprise the current catalogue we started by deriving smells from the Dockerfile best practices ("Best practices for writing Dockerfiles", 2023) page and a blog-post ("Intro Guide to Dockerfile Best Practices - Docker", 2019) by Tibor Vass on the official Docker[1] website. This baseline of smells was expanded by smells found in state-of-the-art tools like Hadolint and by reading the literature on Dockerfile smells and quality of Dockerfiles. As well reading literature on other the quality of other IaC artifacts and transforming those smells that can be applied to Dockerfiles as well.

In total, 95 individual Dockerfile smells were gathered from aforementioned resources and categorized in four categories. 54 Dockerfile smells belong to maintenance (56,8%), 28 Dockerfile smells belong to bloaters (29,5%), 11 Dockerfile smells belong to security (11,6%) and 2 Dockerfile smells belong to layer-optimization (2,1 %). Figure 5.1 visualises these proportions in a pie-chart.

In both the maintenance and bloaters category there are more general smells which include more concrete implementations. Figure 5.2 shows the proportions of the individual identified Dockerfile smells that were categorized under maintenance. 22,2% of these smells have to do with version pinning, 14,8% of these smells have to do with interaction prevention. The remaining 63% stand on their own. Figure 5.3 shows the proportions of the individual identified Dockerfile smells that were categorized under bloaters. 42,9% of these are comprised of smells related to keeping cache unnecessarily. 28,6% of these smells are related to temporary file smells and 10,7% to installing recommended packages.

Findings: We gathered 95 individual Dockerfile smells categorized in four categories: maintenance, bloaters, security and layer-optimization. Maintenance is the largest category, followed up by bloaters, then security and as last layer-optimization. Both maintenance and bloaters contain Dockerfile smells that have many specific implementations.

5.3.2 Does our tool perform well compared to the state-of-the-art

//TODO manuel verification, precision and recall of both Hadolint and the tool. Both their data is available in the same format. Just have to do the manual verification ... In terms of

[1] https://www.docker.com/

Smell Code	Explanation
MAINTENANCE	
DL3006	Always tag the version of an image explicitly
DL3007	Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag
DL3008	Pin the version of the installed software package using `apt-get`
DL3013	Pin the version of the installed software package using `pip`
DL3016	Pin the version of the installed software package using `npm`
DL3028	Pin the version of the installed software package using `gem`
DL3033	Pin the version of the installed software package using `yum`
DL3037	Pin the version of the installed software package using `zypper`
DL3041	Pin the version of the installed software package using `dnf`
DL9001	Pin the version of the installed software package using `apt`
DL9002	Pin the version of the installed software package using `pip3`
DL3014	Use the -y (assumed yes) flag when using `apt-get`
DL3030	Use the -y (assumed yes) flag when using `yum`
DL3034	Use the -y (assumed yes) flag when using `zypper`
DL3038	Use the -y (assumed yes) flag when using `dnf`
DL9003	Use the -y (assumed yes) flag when using `apt`
DL9004	Use the –no-input (assumed yes) flag when using `pip3`
DL9005	Use the –no-input (assumed yes) flag when using pip
BLOATERS	
DL3009	Delete the apt-get lists after installing something using apt-get
DL3019	Use the –no-cache switch
DL3032	yum clean all missing after yum command
DL3036	zypper clean missing after zypper use
DL3040	dnf clean all missing after dnf command
DL3042	Avoid cache directory with pip install –no-cache-dir
DL3060	yarn cache clean missing after yarn install was run.
DL9008	Make sure to clean the cache when using npm
DL9009	Make sure to clean the cache when using pip3
DL9010	Make sure to clean the cache when using conda
DL9011	Make sure to clean the cache when using apt-get
DL9012	Make sure to clean the cache when using apt
DL3010	Use ADD for extracting archives into an image.
DL9013	COPY/rm pattern causing temporary file
DL9014	COPY/rm pattern causing temporary file
DL9015	build-in-cmd/rm pattern
DL9016	ADD introduced compressed file through an URL which was decompressed but not deleted.
DL9017	COPY introduced compressed file which was decompressed but not deleted
DL9018	BUILT-IN introduced compressed file which was decompressed but not deleted
DL9019	Compressed file, introduced from URL through ADD, was deleted in a later layer
DL3015	Using apt-get, avoid additional packages by specifying –no-install-recommends
DL9006	Using apt, avoid additional packages by specifying –no-install-recommends
DL9007	Using zypper, avoid additional packages by specifying –no-recommends
DL3059	Multiple consecutive RUN instructions
LAYER-OPTIMISATION	
DL9000	Using npm, install dependencies before copying source code
DL9020	Using pip, install dependencies before copying source code

Table 5.2: Legend for smell codes

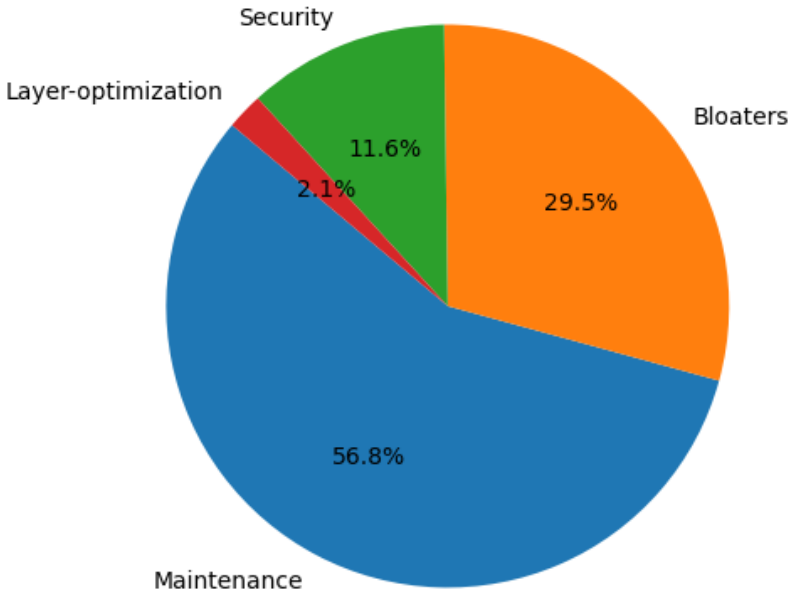

Figure 5.1: Pie chart of category proportions of the Dockerfile smells

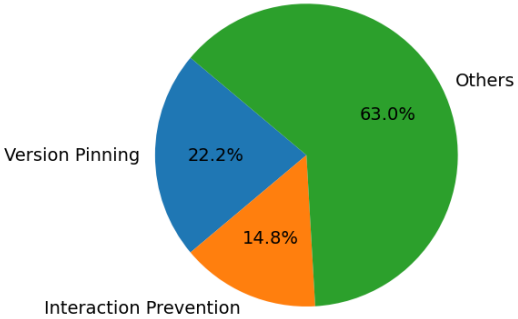

Figure 5.2: Pie chart of maintenance proportions

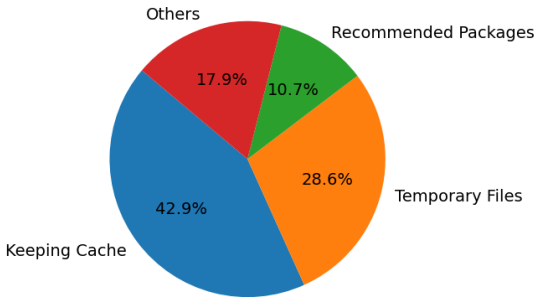

Figure 5.3: Pie chart of bloater proportions

milliseconds, might have to redo the experiments. Will have to ask.

Findings:

5.3.3 How prevalent are the detected smells in the GitHub ecosystem

We conducted the experiment by running the tool over the merged dataset described in section 5.2.2 which is comprised of 323,347 Dockerfiles. During the experiment, 1,535 Dockerfiles were only able to run partially, which is 0.5% of the entire merged dataset. Of the 323,347 analysed Dockerfiles, 205,188 (63%) contains at least Dockerfile smell and 118,159 (37%) do not contain

any smell detected by the tool. The smell density is 7.6 smells per Dockerfile.

On the smelly Dockerfiles, the tool was able to detect 2,485,993 cases of individual smells. Table 5.3 shows the proportion of each smell attributing to all the detected smells. With 41%, DL3008, not pinning the version of software packages using `apt-get`, has the largest proportion with more than 1,000,000 occurrences in the dataset. The second largest proportion, 11%, also has to do with not pinning the version of software packages when using `apk` (DL3018). When looking at the smells concerned with layer-optimization (DL9000, DL9020) which are highlighted in grey, it is clear that their proportion is less than one percent of the detected smells. When looking at the smells concerning temporary file smells which are highlighted in cyan, all but DL9018 have a proportion of < 0.01 and DL9018 itself has a proportion of 1% of the detected Dockerfile smells.

Table 5.4 shows the prevalence of the detected Dockerfile smells on the merged dataset. DL3008 has the highest prevalence: 32%. Followed up by related smells also have a high prevalence of 27% and 23%. The smells related to layer-optimization, highlighted in grey, both have a prevalence of 1%. The smells related to temporary file smells, highlighted in cyan, have a prevalence of 1% or less with the exception of DL9018. This smell, introducing a compressed file through built-in tools and not deleting it, has a prevalence of 5%.

Findings: In the context of the merged dataset we find that 63% of the Dockerfiles contain at least one Dockerfile smell detected by tool. Smells relating to the command `apt-get` and version pinning are most prevalent. All the identified smells related to layer-optimization (DL9000, DL9020) have a prevalence of 1%. All the Dockerfile smells relating to temporary files (DL3010, DL9013, DL9014, DL9015, DL9016, DL9017, DL9019) except DL9018 have a prevalence of one percent or less. DL9018, introducing a compressed file through a built-in tool and not deleting it, has a prevalence of 5%.

5.3.4 How prevalent are the detected smells in the StackOverflow ecosystem

We conducted the experiment by running the tool over the StackOverflow dataset described in section 5.2.3 which is comprised of 24,881 Dockerfiles. Of these Dockerfiles, 447 were able to only run partially. This represents 2% of the entire StackOverflow dataset. Of the 24,881 Dockerfiles, 7,858 (32%) contained Dockerfile smells whilst 17,023 (68%) did not contain any smell detected by the tool. The smell density for the entire dataset is 2.3 smells per Dockerfile.

On these 7,858 infected Dockerfiles, the tool was able to detect 57,292 cases of individual Dockerfile smells. Table 5.5 shows the proportions of the detected Dockerfile smells. 26% of all the detected smells have to do with not pinning the version of software packages when using `apt-get` (DL3008). Not pinning versions of the software packages using `apk` (DL3018), not cleaning cache using `apt-get` (DL9011), not deleting the cache using `apt-get` (DL3009), installing recommended packages using `apt-get` (DL3015) and using multiple `RUN` instructions (DL3059) all have an equal share of 9% and amount to 45% of the detected smells. The layer optimization smells (DL9000, DL9020), highlighted in grey, account for 2% of the detected smells. All the Dockerfile smells pertaining to temporary files, highlighted in cyan, each account for less than 1% of the detected smells.

Table 5.6 shows the prevalence of a smell on the dataset. The most prevalent smell, DL3008,

Smell proportion		
Smell	# Detections	proportion (%)
DL3008	1,012,020	0.41
DL3018	271,522	0.11
DL9011	187,811	0.08
DL3009	187,811	0.08
DL3015	162,548	0.07
DL3059	122,396	0.05
DL3013	71,379	0.03
DL9005	63,992	0.03
DL9001	53,464	0.02
DL3042	52,424	0.02
DL3033	48,090	0.02
DL9008	28,122	0.01
DL9002	24,824	0.01
DL9018	23,300	0.01
DL3019	21,062	0.01
DL9004	20,172	0.01
DL9012	20,024	0.01
DL9009	17,023	0.01
DL3016	16,830	0.01
DL3041	16,060	0.01
DL3032	14,710	0.01
DL9006	9,453	$< 0,01$
DL3014	7,053	$< 0,01$
DL3037	6,162	$< 0,01$
DL9020	4,495	$< 0,01$
DL9000	4,458	$< 0,01$
DL3060	4,213	$< 0,01$
DL9010	3,555	$< 0,01$
DL3040	2,432	$< 0,01$
DL9014	2,177	$< 0,01$
DL9015	1,708	$< 0,01$
DL9013	1,046	$< 0,01$
DL9007	682	$< 0,01$
DL3036	659	$< 0,01$
DL3034	549	$< 0,01$
DL9016	480	$< 0,01$
DL3010	455	$< 0,01$
DL9003	330	$< 0,01$
DL9017	275	$< 0,01$
DL9019	180	$< 0,01$
DL3030	36	$< 0,01$
DL3038	11	$< 0,01$

Table 5.3: Smell proportion on the merged dataset

was found at least once in 3128 files of the StackOverlow dataset. The prevalence of this smell is 13%. When looking at the top 10 smells from table 5.5 containing the smell proportion, we

Smell prevalence		
Smell	# Files	Proportion (%)
DL3008	102,805	0.32
DL3009	88,203	0.27
DL9011	88,203	0.27
DL3015	74,133	0.23
DL3018	45,859	0.14
DL3059	36,694	0.11
DL9005	35,086	0.11
DL3042	29,883	0.09
DL3013	22,287	0.07
DL9008	21,590	0.07
DL9018	16,343	0.05
DL3019	16,047	0.05
DL9004	11,511	0.04
DL3033	10,108	0.03
DL3016	10,042	0.03
DL9009	9,858	0.03
DL9002	8,213	0.03
DL9001	7,059	0.02
DL3032	7,033	0.02
DL9012	6,724	0.02
DL9006	6,441	0.02
DL3014	4,756	0.01
DL9020	4,313	0.01
DL9000	4,270	0.01
DL3060	3,695	0.01
DL3041	3,321	0.01
DL9010	1,830	0.01
DL9014	1,712	0.01
DL3040	1,397	< 0.01
DL9015	1,268	< 0.01
DL9013	898	< 0.01
DL3037	427	< 0.01
DL9016	397	< 0.01
DL9007	389	< 0.01
DL3010	375	< 0.01
DL3036	355	< 0.01
DL3034	313	< 0.01
DL9003	299	< 0.01
DL9017	229	< 0.01
DL9019	135	< 0.01
DL3030	31	< 0.01
DL3038	10	< 0.01

Table 5.4: Smell prevalence on the merged dataset

see that they contain the same smells albeit not in the same sequence. Each smell with regards to layer-optimization (DL9000, DL9020), highlighted with grey, occurs in 1% of the files of the

Smell Proportion		
Smell	# Detections	Proportion (%)
DL3008	15,078	0.26
DL3018	5,206	0.09
DL9011	5,082	0.09
DL3009	5,082	0.09
DL3015	4,993	0.09
DL3059	4,919	0.09
DL9005	2,367	0.04
DL3042	2,141	0.04
DL9008	2,018	0.04
DL3013	1,536	0.03
DL9012	896	0.02
DL9001	864	0.02
DL3019	862	0.02
DL3033	843	0.01
DL3016	727	0.01
DL9004	686	0.01
DL3032	649	0.01
DL9009	617	0.01
DL9002	589	0.01
DL9006	437	0.01
DL9020	370	0.01
DL3014	345	0.01
DL9000	315	0.01
DL3060	184	< 0.01
DL9018	152	< 0.01
DL3041	65	< 0.01
DL9014	61	< 0.01
DL9010	47	< 0.01
DL9003	42	< 0.01
DL9015	33	< 0.01
DL3030	29	< 0.01
DL3010	14	< 0.01
DL9013	10	< 0.01
DL3040	10	< 0.01
DL9016	9	< 0.01
DL9017	6	< 0.01
DL3037	3	< 0.01
DL9007	3	< 0.01
DL9019	2	< 0.01

Table 5.5: Smell proportion on the StackOverflow dataset

dataset. Each smell with regards to temporary files, highlighted in cyan, occurs in less than 1% of the files of the dataset.

TODO - Discuss with advisors what else can be done. I have a couple of ideas myself but I am not sure if they are anything good. These are the results and for now I have posted the

Smell prevalence		
Smell	# Files	Proportion (%)
DL3008	3,128	0.13
DL3009	3,003	0.12
DL9011	3,003	0.12
DL3015	2,910	0.12
DL3059	1,835	0.07
DL9005	1,610	0.06
DL9008	1,588	0.06
DL3042	1,470	0.06
DL3018	1,128	0.05
DL3013	744	0.03
DL3019	621	0.02
DL3016	460	0.02
DL9004	437	0.02
DL9009	398	0.02
DL9020	362	0.01
DL9000	300	0.01
DL3032	297	0.01
DL9012	288	0.01
DL9001	284	0.01
DL3014	284	0.01
DL9006	282	0.01
DL3033	271	0.01
DL9002	253	0.01
DL3060	174	0.01
DL9018	133	0.01
DL9014	48	< 0.01
DL9003	41	< 0.01
DL9010	35	< 0.01
DL9015	29	< 0.01
DL3030	25	< 0.01
DL3010	14	< 0.01
DL3041	11	< 0.01
DL9013	10	< 0.01
DL3040	7	< 0.01
DL9016	7	< 0.01
DL9017	6	< 0.01
DL9019	2	< 0.01
DL3037	1	< 0.01
DL9007	1	< 0.01

Table 5.6: Smell prevalence on the StackOverflow dataset

”hard” results. (and ofc, interpretation of these are reserved for the discussion section)

Findings: In the context of StackOverflow, we find that 32% of the Dockerfiles contain at least one smell detected by the tool. Smells relating to the `apt-get` command (DL3008, DL3009, DL9011, DL9015) are more prevalent than any other smell. Smells relating to layer optimization (DL9000, DL9020) have a prevalence of 1% each and smells relating to temporary files (DL3010, DL9013, DL9014, DL9015, DL9016, DL9017, DL9018, DL9019) have a prevalence of $< 1\%$.

5.4 Discussion

This section discusses the relevance of our results given in the previous sections.

TODO Discuss smells (first "proper" academic catalogue of Dockerfile smells With regards to our first research question, underlines and supports previous research.

Stackoverflow thoughts:

- General advise, pseudocode, ... goes undetected. These are patterns which could maybe be detected by other systems trained on that. Many mistakes in actual, concrete feedback of Dockerfiles and not in statements explaining how like 1 instruction works. Will need to double check
- Top 10 smells the same - Lots of smells detected, probably in lots of files (not unreasonable to argue)

5.5 Threats to validity

Here comes my threats to validity

Ideas for validity:

- Smells might manifest itself in a way not covered by the tool
- There are more smells out there than gathered by the study
-

5.5.1 Internal threats to validity

Here comes my internal threats to validity

5.5.2 External threats to validity

Here comes my external threats to validity

Chapter 6

Conclusion

Here comes my conclusion

- Catalogue: researchers can take a more structured approach to detecting and fixing Dockerfile smells.

Bibliography

Azuma, H., Matsumoto, S., Kamei, Y., & Kusumoto, S. (2022). An empirical study on self-admitted technical debt in Dockerfiles [Place: Dordrecht Publisher: Springer WOS:000750874900002]. *Empirical Software Engineering*, *27*(2), 49. https://doi.org/10.1007/s10664-021-10081-7

Best practices for writing Dockerfiles. (2023). Retrieved July 17, 2023, from https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

Cito, J., Schermann, G., Witternt, J. E., Leitner, P., Zumberi, S., & Gall, H. C. (2017). An Empirical Analysis of the Docker Container Ecosystem on GitHub [ISSN: 2160-1852 WOS:000425917100034]. *2017 Ieee/Acm 14th International Conference on Mining Software Repositories (msr 2017)*, 323–333. https://doi.org/10.1109/MSR.2017.67

De Roover, C., & Stevens, R. (2014). Building Development Tools Interactively using the EKEKO Meta-Programming Library [WOS:000349240500059]. In S. Demeyer, D. Binkley, & F. Ricca (Eds.), *2014 Software Evolution Week - Ieee Conference on Software Maintenance, Reengineering, and Reverse Engineering (csmr-Wcre)* (pp. 429–433). Ieee. Retrieved June 25, 2023, from https://www.webofscience.com/wos/alldb/full-record/WOS:000349240500059

Dillon, T., Wu, C., & Chang, E. (2010). Cloud Computing: Issues and Challenges [ISSN: 1550-445X Num Pages: 7 Series Title: International Conference on Advanced Information Networking and Applications Web of Science ID: WOS:000299433200007]. *2010 24TH IEEE INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS (AINA)*, 27–33. https://doi.org/10.1109/AINA.2010.187

Dockerfile reference — Docker Documentation. (n.d.). Retrieved November 17, 2022, from https://docs.docker.com/engine/reference/builder/

Durieux, T. (2023a). Parfum: Detection and Automatic Repair of Dockerfile Smells [arXiv:2302.01707 [cs]]. https://doi.org/10.48550/arXiv.2302.01707

Durieux, T. (2023b). Parfum Experiment [original-date: 2023-01-26T10:24:24Z]. Retrieved July 30, 2023, from https://github.com/tdurieux/docker-parfum-experiment

Eng, K., & Hindle, A. (2021). Revisiting Dockerfiles in Open Source Software Over Time [ISSN: 2160-1852 WOS:000693399500045]. *2021 Ieee/Acm 18th International Conference on Mining Software Repositories (msr 2021)*, 449–459. https://doi.org/10.1109/MSR52588.2021.00057

Fowler, M., et al. (2003). Refactoring: Improving the design of existing code. 2000. *DOI= http://www. martinfowler. com/books. html/refactoring*.

Haskell Dockerfile Linter [original-date: 2015-11-15T20:20:58Z]. (2023). Retrieved July 17, 2023, from https://github.com/hadolint/hadolint

Hassan, F., Rodriguez, R., & Wang, X. (2018). RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis [ISSN: 1527-1366 WOS:000553784500075]. In

M. Huchard, C. Kastner, & G. Fraser (Eds.), *Proceedings of the 2018 33rd Ieee/Acm International Conference on Automted Software Engineering (ase' 18)* (pp. 796–801). Ieee. https://doi.org/10.1145/3238147.3240470

Henkel, J., Bird, C., Lahiri, S. K., & Reps, T. (2020a). Learning from, Understanding, and Supporting DevOps Artifacts for Docker [ISSN: 0270-5257 WOS:000652529800004]. *2020 Acm/Ieee 42nd International Conference on Software Engineering (icse 2020)*, 38–49. https://doi.org/10.1145/3377811.3380406

Henkel, J., Bird, C., Lahiri, S. K., & Reps, T. (2020b). ICSE 2020 Artifact for: Learning from, Understanding, and Supporting DevOps Artifacts for Docker [Language: eng]. https://doi.org/10.5281/zenodo.3628771

Henkel, J., Bird, C., Lahiri, S. K., & Reps, T. (2020c). A Dataset of Dockerfiles. *Proceedings of the 17th International Conference on Mining Software Repositories*, 528–532. https://doi.org/10.1145/3379597.3387498

Henkel, J., Silva, D., Teixeira, L., d'Amorim, M., & Reps, T. (2021). Shipwright: A Human-in-the-Loop System for Dockerfile Repair [ISSN: 1558-1225]. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 1148–1160. https://doi.org/10.1109/ICSE43902.2021.00106

Huang, Z., Wu, S., Jiang, S., & Jin, H. (2019). FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container [ISSN: 2160-195X WOS:000852685500004]. *2019 35th Symposium on Mass Storage Systems and Technologies (msst 2019)*, 28–37. https://doi.org/10.1109/MSST.2019.00-18

Hunt, A., & Thomas, D. (2000). The pragmatic programmer: From journeyman to master.

Intro Guide to Dockerfile Best Practices - Docker [Section: Products]. (2019). Retrieved November 17, 2022, from https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394. https://doi.org/10.1145/360248.360252

Ksontini, E., Kessentini, M., Ferreira, T. d. N., & Hassan, F. (2021). Refactorings and Technical Debt in Docker Projects: An Empirical Study [WOS:000779309000066]. *2021 36th Ieee/Acm International Conference on Automated Software Engineering Ase 2021*, 781–791. https://doi.org/10.1109/ASE51524.2021.9678585

Laukkanen, E., & Mantyla, M. V. (2015). Build Waiting Time in Continuous Integration - An Initial Interdisciplinary Literature Review [Num Pages: 4 Web of Science ID: WOS:000380530700001]. *2015 IEEE/ACM 2ND INTERNATIONAL WORKSHOP ON RAPID CONTINUOUS SOFTWARE ENGINEERING (RCOSE)*, 1–4. https://doi.org/10.1109/RCoSE.2015.8

Lin, C., Nadi, S., & Khazaei, H. (2020). A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub [ISSN: 1063-6773 WOS:000613139300033]. *2020 Ieee International Conference on Software Maintenance and Evolution (icsme 2020)*, 371–381. https://doi.org/10.1109/ICSME46990.2020.00043

Lu, Z., Xu, J., Wu, Y., Wang, T., & Huang, T. (2019). An Empirical Case Study on the Temporary File Smell in Dockerfiles [Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc WOS:000469942900001]. *Ieee Access*, *7*, 63650–63659. https://doi.org/10.1109/ACCESS.2019.2905424

Meldrum, S., Licorish, S. A., Owen, C. A., & Savarimuthu, B. T. R. (2020). Understanding stack overflow code quality: A recommendation of caution [Num Pages: 50 Place: Amsterdam Publisher: Elsevier Web of Science ID: WOS:000577344100002]. *SCIENCE OF COMPUTER PROGRAMMING*, *199*, 102516. https://doi.org/10.1016/j.scico.2020.102516

Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment, 5.

Pahl, C. (2015). Containerization and the PaaS Cloud [Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc WOS:000366899300005]. *Ieee Cloud Computing*, *2*(3), 24–31. https://doi.org/10.1109/MCC.2015.51

Prinetto, P. E., Bortolameotti, D. R., & Massaro, G. (2022). Security Misconfigurations Detection and Repair in Dockerfile. *2022*, 78.

Rahman, A., Parnin, C., & Williams, L. (2019). The Seven Sins: Security Smells in Infrastructure as Code Scripts [ISSN: 0270-5257 WOS:000560373200015]. *2019 Ieee/Acm 41st International Conference on Software Engineering (icse 2019)*, 164–175. https://doi.org/10.1109/ICSE.2019.00033

Saltzer, J., & Schroeder, M. (1975). Protection of Information in Computer Systems [Num Pages: 31 Place: New York Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:A1975AP33300002]. *PROCEEDINGS OF THE IEEE*, *63*(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

Schermann, G., Zumberi, S., & Cito, J. (2018). Structured Information on State and Evolution of Dockerfiles on GitHub [ISSN: 2160-1852 Num Pages: 4 Series Title: IEEE International Working Conference on Mining Software Repositories Web of Science ID: WOS:000458687200007]. *2018 IEEE/ACM 15TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR)*, 26–29. https://doi.org/10.1145/3196398.3196456

Sharma, T., Fragkoulis, M., & Spinellis, D. (2016). Does Your Configuration Code Smell? [WOS:000391614700018]. *13th Working Conference on Mining Software Repositories (msr 2016)*, 189–200. https://doi.org/10.1145/2901739.2901761

Sharma, T., & Spinellis, D. (2018). A survey on software smells [Place: New York Publisher: Elsevier Science Inc WOS:000426233300009]. *Journal of Systems and Software*, *138*, 158–173. https://doi.org/10.1016/j.jss.2017.12.034

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

What is a Container? — Docker. (2021). Retrieved June 10, 2023, from https://www.docker.com/resources/what-container/

Wu, Y., Zhang, Y., Wang, T., & Wang, H. (2020). Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study [Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc WOS:000567609700008]. *Ieee Access*, *8*, 34127–34139. https://doi.org/10.1109/ACCESS.2020.2973750

Wu, Y., Zhang, Y., Xu, K., Wang, T., & Wang, H. (2023). Understanding and Predicting Docker Build Duration: An Empirical Study of Containerized Workflow of OSS Projects. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13. https://doi.org/10.1145/3551349.3556940

Xu, J., Wu, Y., Lu, Z., & Wang, T. (2019). Dockerfile TF smell detection based on dynamic and static analysis methods [ISSN: 0730-3157 WOS:000538791700023]. In V. Getov, J. L. Gaudiot, N. Yamai, S. Cimato, M. Chang, Y. Teranishi, J. J. Yang, H. V. Leong, H. Shahriar, M. Takemoto, D. Towey, H. Takakura, A. Elci, Susumu, & S. Puri (Eds.), *2019 Ieee 43rd Annual Computer Software and Applications Conference (compsac), Vol 1* (pp. 185–190). Ieee Computer Soc. https://doi.org/10.1109/COMPSAC.2019.00033

Yamashita, A. (2014). Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, *19*(4), 1111–1143. https://doi.org/10.1007/s10664-013-9250-3

Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019). On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs [WOS:000469754100047].

In X. Wang, D. Lo, & E. Shihab (Eds.), *2019 Ieee 26th International Conference on Software Analysis, Evolution and Reengineering (saner)* (pp. 491–501). Ieee. Retrieved October 23, 2022, from http://arxiv.org/pdf/1811.12874

Zhang, Y., Yin, G., Wang, T., Yu, Y., & Wang, H. (2018). An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 138–143. https://doi.org/10.1109/COMPSAC.2018.00026

Zhang, Y., Zhang, Y., Mao, X., Wu, Y., Lin, B., & Wang, S. (2022). Recommending Base Image for Docker Containers based on Deep Configuration Comprehension [ISSN: 1944-2793 WOS:000855050800048]. *2022 Ieee International Conference on Software Analysis, Evolution and Reengineering (saner 2022)*, 449–453. https://doi.org/10.1109/SANER53432.2022.00060

Zhou, Y., Zhan, W., Li, Z., Han, T., Chen, T., & Gall, H. (2023). DRIVE: Dockerfile Rule Mining and Violation Detection [arXiv:2212.05648 [cs]]. https://doi.org/10.48550/arXiv.2212.05648

Appendices

Appendix A

A. Dockerfile Code Smells Catalogue

A.1 Maintainability

A.1.1 Version Pinning

DL3006

Code: DL3006
Message: Always tag the version of an image explicitly
Category: MAINTAINABILITY
Rationale: You can never rely that the latest tag is a specific version.
Problematic Code:

```
1 FROM debian
```

Correct Code:

```
1 FROM debian:jessie
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3007

Code: DL3007
Message: Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag.
Category: MAINTAINABILITY
Rationale: Cannot rely on the latest tag being a specific version.
Problematic Code:

```
1 FROM debian:latest
```

Correct Code:

```
1 FROM debian:jessie
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3008

Code: DL3008
Message: Pin the version of the installed software package using `apt-get`
Category: MAINTAINABILITY
Rationale: Version pinning forces the build to retrieve a particular version regardless of what is in the cache. This reduces failure due to unanticipated changes in required packages.
Problematic Code:

```
FROM busybox
RUN apt-get install python
```

Correct Code:

```
FROM busybox
RUN apt-get install python=2.7
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3013

Code: DL3013
Message: Pin the version of the installed software package using `pip`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM python:3.4
RUN pip install django
```

Correct Code:

```
FROM python:3.4
RUN pip install django==1.9
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3016

Code: DL3016
Message: Pin the version of the installed software package using `npm`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM node:8.9.1
RUN npm install express
```

Correct Code:

```
FROM node:8.9.1
RUN npm install express@4.1.1
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3018

Code: DL3018
Message: Pin the version of the installed software package using `apk`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM alpine:3.7
RUN apk --no-cache add foo
```

Correct Code:

```
FROM alpine:3.7
RUN apk --no-cache add foo=1.2.3
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3028

Code: DL3028
Message: Pin the version of the installed software package using `gem`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM ruby:2
RUN gem install bundler
```

Correct Code:

```
FROM ruby:2
RUN gem install bundler:1.1
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3033

Code: DL3033
Message: Pin the version of the installed software package using `yum`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
RUN yum install -y httpd && yum clean all
```

Correct Code:

```
RUN yum install -y httpd-2.24.2 && yum clean all
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3037

Code: DL3037
Message:Pin the version of the installed software package using `zypper`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd && zypper clean
```

Correct Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd=2.4.46 && zypper clean
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL3041

Code: DL3041
Message: Pin the version of the installed software package using `dnf`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
RUN dnf install -y httpd && dnf clean all
```

Correct Code:

```
RUN dnf install -y httpd-2.24.2 && dnf clean all
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL9001

Code: DL9001
Message: Pin the version of the installed software package using `apt`
Category: MAINTAINABILITY
Rationale: Version pinning forces the build to retrieve a particular version regardless of what is in the cache. This reduces failure due to unanticipated changes in required packages.
Problematic Code:

```
FROM busybox
RUN apt install python
```

Correct Code:

```
FROM busybox
RUN apt install python=2.7
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

DL9002

Code: DL9002
Message: Pin the version of the installed software package using `pip3`
Category: MAINTAINABILITY
Rationale: Version pinning reduces failures due to unanticipated changes in required packages.
Problematic Code:

```
FROM python:3.4
RUN pip3 install django
```

Correct Code:

```
FROM python:3.4
RUN pip3 install django==1.9
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.2 Interaction Prevention

DL3014

Code: DL3014
Message: Use the -y (assumed yes) flag when using `apt-get`
Category: MAINTAINABILITY
Rationale: Without the –assume-yes option it might be possible that the build breaks without human intervention.
Problematic Code:

```
FROM debian:<tag>
RUN apt-get install python=2.7
```

Correct Code:

```
FROM debian:<tag>
RUN apt-get install -y python=2.7
```

Sources: Hadolint Wiki, (Henkel et al., 2020a)

DL3030

Code: DL3030
Message: Use the -y (assumed yes) flag when using `yum`
Category: MAINTAINABILITY
Rationale: Without the `-y` flag or the equivalent `--assumeyes` flag, yum will not successfully install a package, because human input is expected. The -y flag makes yum assume 'yes' as the answer to prompts during the installation.
Problematic Code:

```
FROM centos
RUN yum install httpd-2.24.4 && yum clean all
```

Correct Code:

```
FROM centos
RUN yum install -y httpd-2.24.4 && yum clean all
```

Sources: Hadolint Wiki, (Henkel et al., 2020a)

DL3034

Code: DL3034
Message: Use the -y (assumed yes) flag when using `zypper`
Category: MAINTAINABILITY
Rationale: Omitting the non-interactive switch causes the command to fail during the build process, because zypper would expect manual input. Use the -y or the equivalent –no-confirm flag to avoid this.
Problematic Code:

```
RUN zypper install httpd=2.4.46 && zypper clean
```

Correct Code:

```
RUN zypper install -y httpd=2.4.46 && zypper clean
```

Sources: Hadolint Wiki, (Henkel et al., 2020a)

DL3038

Code: DL3038
Message: Use the -y (assumed yes) flag when using `dnf`
Category: MAINTAINABILITY
Rationale: Omitting the non-interactive flag causes the command to fail during the build process, as `dnf` would expect manual input. The assumed yes flag avoids this.
Problematic Code:

```
FROM fedora:32
RUN dnf install httpd-2.4.46 && dnf clean all
```

Correct Code:

```
FROM fedora:32
RUN dnf install -y httpd-2.4.46 && dnf clean all
```

Sources: Hadolint Wiki, (Henkel et al., 2020a)

DL9003

Code: DL9003
Message: Use the -y (assumed yes) flag when using `apt`
Category: MAINTAINABILITY
Rationale: Without the –assume-yes option it might be possible that the build breaks without human intervention.
Problematic Code:

```
FROM debian:<tag>
RUN apt install python=2.7
```

Correct Code:

```
FROM debian:<tag>
RUN apt install -y python=2.7
```

Sources:(Henkel et al., 2020a)

DL9004

Code: DL9004
Message: Use the –no-input (assumed yes) flag when using `pip3`
Category: MAINTAINABILITY
Rationale: Without the –no-input flag it might be possible that the build breaks without human intervention.
Problematic Code:

```
FROM debian:<tag>
RUN pip3 install python=2.7
```

Correct Code:

```
FROM debian:<tag>
RUN pip3 --no-input install -y python=2.7
```

Sources:

DL9005

Code: DL9005
Message: Use the –no-input (assumed yes) flag when using `pip`
Category: MAINTAINABILITY
Rationale: Without the –no-input flag it might be possible that the build breaks without human intervention.
Problematic Code:

```
FROM debian:<tag>
RUN pip install python=2.7
```

Correct Code:

```
FROM debian:<tag>
RUN pip --no-input install -y python=2.7
```

Sources:

DL2027

Code: DL2027
Message: gpg use batch flag
Category: MAINTENANCE
Rationale: non-interactive way to generate keys (the program will not ask you)
Problematic Code:

```
TODO
```

Correct Code:

```
TODO
```

Sources:(Henkel et al., 2020a)

A.1.3 DL3000: Use Absolute WORKDIR path

Code: DL3000
Message: Use absolute WORKDIR path
Category: MAINTAINABILITY
Rationale: By using absolute paths you will not run into problems when a previous WORKDIR instruction changes because relative paths are relative to the current WORKDIR. You also often times do not know the WORKDIR context of your base container.
Problematic Code:

```
FROM busybox
WORKDIR usr/src/app
```

Correct Code:

```
FROM busybox
WORKDIR /usr/src/app
```

Sources: Hadolint Wiki,("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.4 DL3003: Use WORKDIR to change working directory

Code: DL3003
Message: Use WORKDIR to change working directory.
Category: MAINTAINABILITY
Rationale: Docker provides the WORKDIR instruction if you need to change the current working directory
Problematic Code:

```
FROM busybox:<tag>
RUN cd /usr/src/app && git clone git@github.com:lukasmartinelli/hadolint.git
```

Correct Code:

```
FROM busybox:<tag>
WORKDIR /usr/scr/app
RUN git clone git@github.com:lukasmartinelli/hadolint.git
```

Sources: Hadolint Wiki,("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.5 DL3011: Use valid UNIX ports

Code: DL3011
Message: Valid UNIX ports range from 0 to 65535.
Category: MAINTAINABILITY
Rationale: Any port outside of this range does not make any sense.
Problematic Code:

```
FROM busybox:<tag>
EXPOSE 80000
```

Correct Code:

```
FROM busybox:<tag>
EXPOSE 65535
```

Sources: Hadolint Wiki, List of TCP and UDP port numbers

A.1.6 DL3012: No multiple HEALTHCHECK statements

Code: DL3012
Message: Multiple HEALTHCHECK instructions
Category: MAINTAINABILITY
Rationale: There can only be one HEALTHCHECK per Dockerfile.
Problematic Code:

```
FROM busybox:<tag>
HEALTHCHECK CMD /bin/healthcheck
[...]
HEALTHCHECK CMD /bin/something
```

Correct Code:

```
FROM busybox:<tag>
HEALTHCHECK CMD /bin/healthcheck
```

Sources: Hadolint Wiki, ("Dockerfile reference — Docker Documentation", n.d.)

A.1.7 DL3020: Use COPY instead of ADD for copying items

Code: DL3020
Message: Use COPY instead of ADD for copying files and folders to the image
Category: MAINTAINABILITY
Rationale: For other items (files, directories) that do not require ADD's tar auto-extraction capability, you should always use COPY.
Problematic Code:

```
FROM python:3.4
ADD requirements.txt /usr/src/app/
```

Correct Code:

```
FROM python:3.4
COPY requirements.txt /usr/src/app/
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.8 DL3021: COPY with more than 2 arguments requires last one to end with /

Code: DL3021
Message: COPY with more than 2 arguments requires the last argument to end with /
Category: MAINTAINABILITY
Rationale: From Dockerfile reference: "If multiple resources are specified, either directly or due to the use of a wildcard, then must be a directory, and it must end with a slash ".
Problematic Code:

```
FROM node:carbon
COPY package.json yarn.lock my_app
```

Correct Code:

```
FROM node:carbon
COPY package.json yarn.lock my_app/
```

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.9 DL3022: COPY –from should reference alias

Code: DL3022
Message: COPY –from should reference a previously defined FROM alias
Category: MAINTAINABILITY
Rationale:
Problematic Code:

```
FROM debian:jesse
RUN stuff
FROM debian:jesse
COPY --from=build some stuff ./
```

Correct Code:

```
FROM debian:jesse as build
RUN stuff
FROM debian:jesse
COPY --from=build some stuff ./
```

Sources: Hadolint Wiki, Docker multi-stage building

A.1.10 DL3023: COPY –from cannot reference its own FROM alias

Code: DL3023
Message: COPY –from cannot reference its own FROM alias
Category: MAINTAINABILITY
Rationale: Trying to copy from the same image the instruction is running in results in an error
Problematic Code:

```
FROM debian:jesse as build

COPY --from=build some stuff ./
```

Correct Code:

```
FROM debian:jesse as build
RUN stuff

FROM debian:jesse
COPY --from=build some stuff ./
```

Sources: Hadolint Wiki

A.1.11 DL3024: FROM aliases must be unique

Code: DL3024
Message: FROM aliases (stage names) must be unique
Category: MAINTAINABILITY
Rationale: Defining duplicate stage names results leads to ambiguity and is prone to errors.
Problematic Code:

```
FROM debian:jesse as build
RUN stuff
FROM debian:jesse as build
RUN more_stuff
```

Correct Code:

```
FROM debian:jesse as build
RUN stuff
FROM debian:jesse as another-alias
RUN more_stuff
```

Sources: Hadolint Wiki

A.1.12 DL3025: JSON notation for CMD and ENTRYPOINT arguments

Code: DL3025
Message: Use arguments JSON notation for CMD and ENTRYPOINT arguments
Category: MAINTAINABILITY
Rationale: When using the plain text version of passing arguments, signals from the OS are not correctly passed to the executables, which is in the majority of the cases what you would expect. CMD should almost always be used in the form of CMD [“executable”, “param1”, “param2”...]. The shell form prevents any CMD or run command line arguments from being used, but has the disadvantage that your ENTRYPOINT will be started as a subcommand of /bin/sh -c, which does not pass signals. This means that the executable will not be the container’s PID 1 - and will not receive Unix signals - so your executable will not receive a SIGTERM from docker stop .
Problematic Code:

```
FROM busybox
ENTRYPOINT s3cmd
```

Correct Code:

```
FROM busybox
ENTRYPOINT ["s3cmd"]
```

Sources: Hadolint Wiki, Dockerfile best practices, Docker engine reference

A.1.13 DL3027: use apt-get or apt-cache instead of apt

Code: DL3027
Message: Do not use apt as it is meant to be an end-user tool, use apt-get or apt-cache instead.
Category: MAINTAINABILITY
Rationale: apt is discouraged by the Linux distributions as an unattended tool as its interface may suffer changes between versions. Better use the more stable apt-get and apt-cache.
Problematic Code:

```
FROM busybox
RUN apt install curl=1.1.0
```

Correct Code:

```
FROM busybox
RUN apt-get install curl=1.1.0
```

Sources: Hadolint Wiki

A.1.14 DL3029: Do not use –platform with FROM

Code: DL3029
Message: Do not use –platform= with FROM.
Category: MAINTAINABILITY
Rationale: Do not specify the platform in FROM as it forces Docker to build an image for a specific platform. This results in an image that is not suitable for multi-staged builds. Only specify it when it is absolutely necessary.
Problematic Code:

```
FROM --platform=x86 busybox
```

Correct Code:

```
FROM busybox
```

Sources: Hadolint Wiki

A.1.15 DL3044: Do not refer to ENV in definition instruction

Code: DL3044
Message: Do not refer to an environment variable with the same `ENV` instruction where it is defined.
Category: MAINTAINABILITY
Rationale: Docker will not expand a variable within the same `ENV` instruction where it is defined. While it will not crash Docker, it is prone to errors.
Problematic Code:

```
ENV FOO=bar \
    BAZ=${FOO}/bla
```

Correct Code:

```
ENV FOO=bar
ENV BAZ=${FOO}/bla
```

Sources: Hadolint Wiki

A.1.16 DL3045: no COPY to relative destination without WORKDIR set

Code: DL3045
Message: `COPY` to a relative destination without `WORKDIR` set.
Category: MAINTAINABILITY
Rationale: Using relative paths is prone to errors when changes are introduced to the `WORKDIR` without updating the destination of the `COPY` instruction.
Problematic Code:

```
FROM scratch:<tag>
COPY foo bar
```

Correct Code:

```
FROM scratch:<tag>
# No workdir set, use absolute paths
COPY foo /bar

# OR
FROM scratch:<tag>
# Workdir set, can use a path relative to the workdir
WORKDIR /
COPY foo bar
```

Sources: Hadolint Wiki

A.1.17 DL3048: Not all strings are supported as label keys

Code: DL3048
Message: Invalid label key
Category: MAINTAINABILITY
Rationale: Not all strings are supported as label keys.
Problematic Code:

```
LABEL +?not..valid--key="foo"
```

Correct Code:

```
LABEL valid-key.label="bar"
```

Sources: Hadolint Wiki, Docker label key format recommendations

A.1.18 DL3051: No empty label

Code: DL3051
Message: `label` is empty.
Category: MAINTAINABILITY
Rationale: The label should not be empty, or else there would be no sense in requiring the label in the first place.
Problematic Code:

```
LABEL foo=""
```

Correct Code:

```
LABEL oopsie="bar"
```

Sources: Hadolint Wiki

A.1.19 DL4000: Maintainer is deprecated

Code: DL4000
Message: MAINTAINER instruction is deprecated

Category: MAINTAINABILITY
Rationale: MAINTAINER is deprecated since Docker 1.13.0.
Problematic Code:

```
FROM busybox:<tag>
MAINTAINER John Doe <foo@johndoe.bar>
```

Correct Code:

```
FROM busybox:<tag>
#In case a maintainer is desired/required.
LABEL maintainer="John Doe <foo@johndoe.bar>"
```

Sources: Hadolint Wiki, Deprecated Engine Features

A.1.20 DL4003: No multiple CMD instructions

Code: DL4003
Message: Multiple CMD instructions
Category: MAINTAINABILITY
Rationale: If you list more than one CMD instruction, only the last CMD instruction will take effect. Having multiple CMD statements leads to confusion.
Problematic Code:

```
FROM busybox:<tag>
CMD /bin/true
CMD /bin/false
```

Correct Code:

```
FROM busybox:<tag>
CMD /bin/false
```

Sources: Hadolint Wiki, Dockerfile reference

A.1.21 DL4004: No multiple ENTRYPOINT instructions

Code: DL4004
Message: Multiple ENTRYPOINT instructions
Category: MAINTAINABILITY
Rationale: If you list more than one ENTRYPOINT then only the last ENTRYPOINT will take effect. Having multiple ENTRYPOINT statements leads to confusion.
Problematic Code:

```
FROM busybox:<tag>
ENTRYPOINT /bin/true
ENTRYPOINT /bin/false
```

Correct Code:

```
FROM busybox:<tag>
ENTRYPOINT /bin/false
```

Sources: Hadolint Wiki, Dockerfile reference

A.1.22 DL4005: Use SHELL to change the default shell

Code: DL4005
Message: Use SHELL to change the default shell.
Category: MAINTAINABILITY
Rationale: Docker provides a SHELL instruction which does not require overwriting /bin/sh in the container.
Problematic Code:

```
# Install bash
RUN apk add --update-cache bash=4.3.42-r3

# Use bash as the default shell
RUN ln -sfv /bin/bash /bin/sh
```

Correct Code:

```
# Install bash
RUN apk add --update-cache bash=4.3.42-r3

# Use bash as the default shell
SHELL ["/bin/bash", "-c"]
```

Sources: Hadolint Wiki, Dockerfile reference

A.1.23 DL4006: Set SHELL options before RUN instruction with a pipe in

Code: DL4006
Message: Set the SHELL options `-o pipefail` before a RUN instruction with a pipe in
Category: MAINTAINABILITY
Rationale: The command should fail to an error at any stage in the pipe, not only determined by the exit code of the last stage. Setting the pipefail ensures that an unexpected error prevents the build from inadvertently succeeding. There are some shells that do not accept this, it is recommended to always explicitly state the SHELL instruction before using pipes in RUN. (see DL4005)
Problematic Code:

```
RUN wget -O - https://some.site | wc -l > /number
```

Correct Code:

```
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
RUN wget -O - https://some.site | wc -l > /number
```

Sources: Hadolint Wiki, Dockerfile best practices

A.1.24 DL9027: ENV <name> <value> syntax is deprecated

Code: DL9027
Message: `ENV name value` syntax is deprecated
Category: MAINTAINABILITY
Rationale: Deprecated in release v20.10. This can be ambiguous. For example in the problematic code, only one environment variable is set whilst the programmer may have intended to be

setting three.
Problematic Code:

```
# A single env-variable ONE with value "TWO= THREE=world"
ENV ONE TWO= THREE=world
```

Correct Code:

```
ENV ONE="" TWO="" THREE="world"
```

Sources: Docker Engine deprecated features

A.1.25 DL9028: `apt-get update` should precede `apt-get install`

Code: DL9028
Message: `apt-get update` should precede `apt-get install`
Category: MAINTAINABILITY
Rationale: The command fetches metadata about the currently installed packages.
Problematic Code:

```
RUN apt-get install python
```

Correct Code:

```
RUN apt-get update && apt-get install python
```

Sources: (Henkel et al., 2020a)

A.1.26 DL9029: `apt update` should precede `apt install`

Code: DL9029
Message: `apt update` should precede `apt install`
Category: MAINTAINABILITY
Rationale: The command fetches metadata about the currently installed packages.
Problematic Code:

```
RUN apt install python
```

Correct Code:

```
RUN apt update && apt-get install python
```

Sources: (Henkel et al., 2020a)

A.1.27 DL9030: Use double quote in label definition

Code: DL9030
Message: Use double quote in label to allow for string interpolation
Category: MAINTAINABILITY
Rationale: Using single quote will take the string as-is. To prevent bugs arising from using single quotes instead of double quotes, always use double quotes.
Problematic Code:

```
LABEL version='1.0'
```

Correct Code:

```
LABEL version="1.0"
```

Sources: Dockerfile reference

A.1.28 DL9031: Sort multiline arguments

Code: DL9031
Message: Sort multiline arguments alphabetically
Category: MAINTAINABILITY
Rationale: Sorting multiline arguments makes later changes easier as they are sorted. It helps in avoiding duplicating packages and makes the list easier to update. It not only adds to maintainability, but also readability of the Dockerfile. One package per line, sorted alphabetically. (//TODO - there are no rules to define this indentation)
Problematic Code:

```
//TODO
```

Correct Code:

```
//TODO
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.29 DL9032: Leverage Build Cache

Code: DL9032
Message: Leverage Build Cache
Category: MAINTAINABILITY
Rationale: Whenever a Dockerfile is build, it checks for each layer if the layer already exists in cache and can be reused. As soon as this fails from one layer, the rest of the layers are built from scratch again. With this in mind, a proper ordering of the layers could result in reducing the build times significantly. For example, when building an NodeJS application, it is safe to assume that the code of the application itself will change the most. Therefore, it would be helpful to place this COPY instruction after installing all dependencies such that these can be taken from the cache need to be rebuild. **Problematic Code**:

```
//TODO
```

Correct Code:

```
//TODO
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.30 DL9033: Same LABEL key should not be given multiple values

Code: DL9033
Message: Same label key should not be given multiple values
Category: MAINTAINABILITY
Rationale: This impedes readable and maintainable code as it could lead to confusion. Only the latest value is remembered.
Problematic Code:

```
LABEL foo="foo"
LABEL foo="bar"
```

Correct Code:

```
LABEL foo="bar"
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.31 DL9034: Avoid the user of specific Docker namespaces

Code: DL9034
Message: Avoid the user of specific Docker namespaces
Category: MAINTAINABILITY
Rationale: The com.docker.*, io.docker.*, and org.dockerproject.* namespaces are reserved by Docker for internal use. **Problematic Code**:

```
LABEL com.docker.foo="bar"
```

Correct Code:

```
LABEL foo="bar"
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.32 DL9035: Do not define multiple labels on one line

Code: DL9035
Message: Do not define multiple labels on one line
Severity: WARNING
Category: MAINTAINABILITY
Rationale: the LABEL instruction does not create a new layer for each label definition. Therefore there is no harm in using a LABEL instruction for each label definition. Whilst defining multiple labels is still supported, it is not recommended as it decreases readability and maintainability. **Problematic Code**:

```
LABEL foo="bar" fu="baar"
```

Correct Code:

```
LABEL foo="bar"
LABEL fu="baar"
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.33 DL9036: Prefix label with reverse DNS of domain

Code: DL9036
Message: Prefix each label key with the revese DNS notation of the domain you own
Category: MAINTAINABILITY
Rationale: Authors of third-party tools should prefix each label key with the reverse DNS notation of a domain they own. **Problematic Code**:

```
LABEL foo="bar"
```

Correct Code:

```
LABEL com.mydomain.foo="bar"
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.34 DL9037: Instructions should be uppercase letters

Code: DL9037
Message: Instructions should be written in uppercase letters
Category: MAINTAINABILITY
Rationale: Instructions should be written in uppercase letters
Problematic Code:

```
from debian:latest
```

Correct Code:

```
FROM debian:latest
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.1.35 DL9038: Length of a line should not be longer than 75 characters

Code: DL9038
Message: Length of a line should not be longer than 75 characters
Category: MAINTAINABILITY
Rationale: Length of a line should not be longer than 75 characters such that each line fits on most screens and can be printed on paper. **Problematic Code**:

```

```

Correct Code:

```

```

Sources: (Sharma & Spinellis, 2018)

A.1.36 DL9039: Chown not using arg -r

Code: DL9039
Message: Chown should have argument -r
Category: MAINTAINABILITY
Rationale: Applying -r means that the permissions will not only be changed for a specific directory but also for every file within the directory.
Problematic Code:

```
RUN chown /path-to-dir
```

Correct Code:

```
RUN chown -r /path-to-dir
```

Sources: (Zhou et al., 2023)

A.2 Bloaters

A.2.1 Keeping Cache

DL3009

Code: DL3009
Message: Delete the apt-get lists after installing something using apt-get
Category: BLOATER
Rationale: Cleaning up the apt cache and removing /var/lib/apt/lists helps keeping the image size down. The clean must be performed in the same RUN step, otherwise it will affect image size in terms a TF-code smell.
Problematic Code: Tag the python version?

```
RUN apt-get update && apt-get install --no-install-recommends -y python
```

Correct Code: Tag the python version?

```
RUN apt-get update && apt-get install --no-install-recommends -y python \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Sources: Hadolint Wiki,

DL3019

Code: DL3019
Message: Use the `--no-cache` switch
Category: BLOATER
Rationale: Alpine Linux 3.3 there exists a new `--no-cache` option for `apk`. It allows users to install packages with an index that is updated and used on-the-fly and not cached locally. Avoids the need to use `--update` and remove `/var/cache/apk/*` when done installing packages.
Problematic Code:

```
FROM alpine:3.7
RUN apk update \
    && apk add foo=1.0 \
    && rm -rf /var/cache/apk/*
```

Correct Code:

```
FROM alpine:3.7
RUN apk --no-cache add foo=1.0
```

Sources: Hadolint Wiki, docker-alpine disabling cache

DL3032

Code: DL3032
Message: `yum clean all` missing after yum command.
Category: BLOATER
Rationale: Clean cached package data after installation to reduce image size. Clean up must be performed in the same RUN step, otherwise it will not affect image size.
Problematic Code:

```
RUN yum install -y httpd-2.24.2
```

Correct Code:

```
RUN yum install -y httpd-2.24.2 && yum clean all
```

Sources: Hadolint Wiki, Dockerfile best practices

DL3036

Code: DL3036
Message: `zypper clean` missing after zypper use
Category: BLOATER
Rationale: Reduce layer and image size by deleting unneeded caches after running zypper.
Problematic Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd=2.4.46
```

Correct Code:

```
FROM opensuse/leap:15.2
RUN zypper install -y httpd=2.4.46 && zypper clean
```

Sources: Hadolint Wiki

DL3040

Code: DL3040
Message: `dnf clean all` missing after `dnf` command.
Category: BLOATER
Rationale: Clean cached package data after installation to reduce image size.
Problematic Code:

```
RUN dnf install -y httpd-2.24.2
```

Correct Code:

```
RUN dnf install -y httpd-2.24.2 && dnf clean all
```

Sources: Hadolint Wiki,Dockerfile best practices

DL3042

Code: DL3042
Message: Avoid cache directory with pip install –no-cache-dir ¡package¿
Category: BLOATER
Rationale: Once a package is installed, it does not need to be re-installed and the Docker cache can be leveraged instead. Since the pip cache makes the images larger and is not needed, it's better to disable it.
Problematic Code:

```
RUN pip3 install MySQL_python
```

Correct Code:

```
RUN pip3 install --no-cache-dir MySQL_python
```

Sources: Hadolint Wiki

DL3060

Code: DL3060
Message: `yarn cache clean` missing after `yarn install` was run.
Category: BLOATER
Rationale: `yarn` keeps a local cache of downloaded packages. This unnecessarily increases image size. It can be cleared by executing `yarn cache` clean in the same run instruction.
Problematic Code:

```
RUN yarn install
```

Correct Code:

```
RUN yarn install \
 && yarn cache clean
```

Sources: Hadolint Wiki

DL9008

Code: DL9008
Message: Make sure to clean the cache when using `npm`
Category: BLOATER
Rationale:
-++ **Problematic Code**:

```
npm install
```

Correct Code:

```
npm install && npm cache clean --force
```

Sources: (Henkel et al., 2020a)

DL9009

Code: DL9009
Message: Make sure to clean the cache when using `pip3`
Category: BLOATER
Rationale:
-++ **Problematic Code**:

```
pip3 install numpy
```

Correct Code:

```
pip3 install numpy --no-cache-dir
```

Sources:

DL9010

Code: DL9010
Message: Make sure to clean the cache when using `conda`
Category: BLOATER
Rationale:
Problematic Code:

```
conda install <package-name>
```

Correct Code:

```
conda install <package-name> && conda clean --all
```

Sources: (Henkel et al., 2020a)

DL9011

Code: DL9011
Message: Make sure to clean the cache when using `apt-get`
Category: BLOATER
Rationale:
Problematic Code:

```
apt-get install <package-name>
```

Correct Code:

```
apt-get install <package-name> && apt-get clean
```

Sources:

DL9012

Code: DL9012
Message: Make sure to clean the cache when using `apt`
Category: BLOATER
Rationale:
Problematic Code:

```
apt install <package-name>
```

Correct Code:

```
apt install <package-name> && apt-get clean
```

Sources:

A.2.2 Temporary Files

DL3010

Code: DL3010
Message: Use ADD for extracting archives into an image.
Category: BLOATER
Rationale: Temporary file smells are always accompanied by a form of transforming operation

such as decompressing. Copying an archive file and then extracting it introduces temporary files in the image that cannot be deleted, even if attempted to do so in later layers.The ADD instruction can directly decompress them in the importing process and is a good alternative that alleviates this problem. **Problematic Code**:

```
FROM centos:7               #layer 1
COPY rootfs.tar.xz <path>   #layer 2
RUN <extract rootfs.tar.xz> #layer 3
```

Correct Code:

```
FROM centos:7
ADD rootfs.tar.xz <path>
```

Sources: Hadolint Wiki

DL9013

Code: DL9013
Message: ADD/rm pattern causing temporary file
Category: BLOATER
Rationale: A temporary file is introduced by the `COPY` instruction in layer 2 and deleted with a run `RUN` instruction in layer 3. This means that the file will not be visible anymore from layer 3, but it is still contained within the image in lower layers.
Problematic Code:

```
FROM centos:7          #layer 1
ADD temp.txt .        #layer 2
RUN rm -f temp.txt     #layer 3
```

Correct Code:

```
# multi-stage builds could help
```

Sources: (Xu et al., 2019)

DL9014

Code: DL9014
Message: COPY/rm pattern causing temporary compressed file
Category: BLOATER
Rationale: A temporary file is introduced by the `COPY` instruction in layer 2 and deleted with a run `RUN` instruction in layer 4. This means that the file will not be visible anymore from layer 4, but it is still contained within the image in lower layers.
Problematic Code:

```
FROM centos:7                          #layer 1
COPY jdk-8u171-linux_x64.tar.gz .      #layer 2
RUN tar x jdk-8u171-linux_x64.tar.gz   #layer 3
RUN rm -f jdk-8u171-linux_x64.tar.gz   #layer 4
```

Correct Code:

```
FROM centos:7
ADD jdk-8u171-linux_x64.tar.gz .
# In case of normal files, multi-stage building could help.
```

Sources: (Xu et al., 2019)

DL9015

Code: DL9015
Message: build-in-cmd/rm pattern
Category: BLOATER
Rationale: Commands such as wget can introduce temporary files which are required to be deleted within the same layer. There are many variations since there are many different ways to import temporary files into an image. This pattern has a common feature such as the import and deletion of temporary files in different image layers. The defined smell is shown with a wget example.
Problematic Code:

```
FROM centos:7
RUN wget http://download.oracle.com/.../jdk-8u171-linux-x64.rpm
RUN rpm -ivh jdk-8u171-linux-x64.rpm
RUN rm -f jdk-8u171-linux-x64.rpm
```

Correct Code:

```
FROM centos:7
RUN wget http://download.oracle.com/.../jdk-8u171-linux-x64.rpm \
    && rpm -ivh jdk-8u171-linux-x64.rpm \
    && rm -f jdk-8u171-linux-x64.rpm
```

Sources: (Xu et al., 2019)
TOOD This case cannot exist

DL9016

Code: DL9016
Message: ADD introduced compressed file through an URL which was decompressed but not deleted.
Category: BLOATER
Rationale:
Problematic Code:

```
FROM debian:<tag>
ADD https://foobar.com/foo.tar.gz
RUN tar -xf foo.tar.gz
```

Correct Code:

```
FROM debian:<tag>
RUN wget https://foobar.com/foo.tar.gz && tar -xf foo.tar.gz && rm -rf foo.tar.gz
# Another option would be multi-stage builds.
```

Sources:
TODO COPY would introduce such a file but it would be marked a DL3010

DL9017

Code: DL9017
Message: COPY introduced compressed file which was decompressed but not deleted.
Category: BLOATER
Rationale:
Problematic Code:

```
TODO
```

Correct Code:

```
TODO
```

Sources:

DL9018

Code: DL9018
Message: BUILT-IN introduced compressed file which was decompressed but not deleted.
Category: BLOATER
Rationale:
Problematic Code:

```
FROM debian:<tag>
RUN wget https://foobar.com/foo.tar.gz
RUN tar -xf foo.tar.gz
```

Correct Code:

```
FROM debian:<tag>
RUN wget https://foobar.com/foo.tar.gz  && tar -xf foo.tar.gz && rm -rf foo.tar.gz
```

Sources:

DL9019

Code: DL9019
Message: Compressed file, introduced from URL through ADD, was deleted in a later layer
Category: BLOATER
Rationale: The `ADD` instruction has extra functionalities such as fetching resources from the web, but if that resource is a compressed file then it will not decompress that file before adding it to the image. This means that deleting this compressed file later is a temporary file smell.
Problematic Code:

```
FROM debian:<tag>
ADD http://foobar.com/foo.tar.gz
RUN tar -xf foo.tar.gz && rm -rf foo.tar.gz
```

Correct Code:

```
# If working with a base-image that has some web tools installed,
# fetching it all in a run instruction can help
FROM debian:<tag>
RUN wget http://foobar.com/foo.tar.gz && tar -xf foo.tar.gz && rm -rf foo.tar.gz
```

Sources:

A.2.3 Recommended packages

DL3015

Code: DL3015
Message: Using `apt-get`, avoid additional packages by specifying –no-install-recommends .
Category: BLOATER
Rationale: Avoid installing additional packages that you did not explicitly want. If they are needed, add them explicitly.
Problematic Code:

```
FROM busybox:<tag>
RUN apt-get install -y python=2.7
```

Correct Code:

```
FROM busybox:<tag>
RUN apt-get install -y --no-install-recommends python=2.7
```

Sources: Hadolint Wiki, Dockerfile reference

DL9006

Code: DL3015
Message: Using `apt`, avoid additional packages by specifying –no-install-recommends .
Category: BLOATER
Rationale: Avoid installing additional packages that you did not explicitly want. If they are needed, add them explicitly.
Problematic Code:

```
FROM busybox:<tag>
RUN apt install -y python=2.7
```

Correct Code:

```
FROM busybox:<tag>
RUN apt install -y --no-install-recommends python=2.7
```

Sources: Dockerfile reference

DL9007

Code: DL9007
Message: Using `zypper`, avoid additional packages by specifying –no-recommends .
Category: BLOATER
Rationale: Avoid installing additional packages that you did not explicitly want. If they are needed, add them explicitly.
Problematic Code:

```
FROM <baseimage>:<tag>
zypper install
```

Correct Code:

```
FROM <baseimage>:<tag>
RUN zypper install -y --no-recommends python=2.7
```

Sources: Dockerfile reference

A.2.4 DL3046: useradd with flag -l

Code: DL3046
Message: useradd without flag -l and high UID will result in excessively large Image
Category: BLOATER
Rationale: Without the -l or the –no-log-init flag, useradd will add the user to the lastlog and faillog databases. This can result in the creation of logically large (sparse) files under /var/log, which in turn unnecessarily inflates container image sizes. This is due to the lack of support for sparse files in overlay filesystems. For what it's worth, this behavior becomes more apparent with longer UIDs, resulting in a few megabytes of extra image size with a six digit UID, up to several gigabytes of excessive image size with even longer UIDs. Disabling this functionality from useradd has minimal disadvantages in a container but saves space and build time.
Problematic Code:

```
RUN useradd -u 123456 foobar
```

Correct Code:

```
RUN useradd -l -u 123456 foobar
```

Sources: Hadolint Wiki

A.2.5 DL3047: Use wget flag to avoid bloated build logs

Code: DL3047
Message: Use wget –progress to avoid excessively bloated build logs
Category: BLOATER
Rationale: How can we know if a file is large or not and when should this have effect or not?
Problematic Code:

```
FROM ubuntu:20
RUN wget https://example.com/big_file.tar
```

Correct Code:

```
FROM ubuntu:20
RUN wget --progress=dot:giga https://example.com/big_file.tar
```

Sources: Hadolint Wiki

A.2.6 DL3059: Consolidate multiple consecutive RUN instructions

Code: DL3059
Message: Multiple consecutive RUN instructions.
Category: BLOATER
Rationale: Each RUN instruction will create a new layer in the resulting image. Therefore, consider consolidation as it this will reduce the layer count and reduce the size of the image
Problematic Code:

```
# big files stored in image layer!
RUN command_creating_big_files
RUN command_deleting_these_files
```

Correct Code:

```
RUN command_creating_big_files \
 && command_deleting_these_files
```

Sources: Hadolint Wiki, Docker development best practices

A.2.7 DL4001: Either use Wget or Curl

Code: DL4001
Message: Either use Wget or Curl, but not both.
Category: BLOATER
Rationale: Do not install two tools that have the same effect and keep your image smaller than it would be.
Problematic Code:

```
FROM debian:<tag>
RUN wget http://google.com
RUN curl http://bing.com
```

Correct Code:

```
FROM debian:<tag>
RUN curl http://google.com
RUN curl http://bing.com
```

Sources: Hadolint Wiki

A.2.8 DL4001: Use the smallest base image possible with FROM

Code: DL4001
Message: Use the smallest base image possible with FROM
Category: BLOATER
Rationale: Smaller base image results in a smaller container. For example: use Alpine instead of Debian.
Problematic Code:

```
FROM debian:<tag>
```

Correct Code:

```
FROM alpine:<tag>
```

Sources: ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.3 Security Smells

A.3.1 DL3001: Use no dangerous commands

Code: DL3001
Message: Command does not make sense in a container
Category: SECURITY
Rationale: For some POSIX commands it makes no sense to run them in a Docker container because they are bound to the host or are otherwise dangerous (like ´shutdown´, ´service´, ´ps´, ´free´, ´top´, ´kill´, ´mount´, ´ifconfig´). Interactive utilities also don't make much sense

(´nano´, ´vim´).
Problematic Code:

```
FROM busybox:<tag>
RUN top
#Or any other of the mentioned POSIX commands
```

Correct Code:

```
FROM busybox:<tag>
```

Sources: Hadolint Wiki
Reservation: find out which ones specifically in Hadolint and what about multistage builds?

A.3.2 DL3002: Last user should not be root

Code: DL3002
Message: Last user should not be root
Category: SECURITY
Rationale: Switching to the root USER opens up certain security risks if an attacker gets access to the container. In order to mitigate this, switch back to a non-privileged user after running the commands you need as root.
Problematic Code:

```
FROM busybox:<tag>
USER root
RUN ...
```

Correct Code:

```
FROM busybox:<tag>
USER root
RUN ...
USER guest
```

Sources: Hadolint Wiki,Dockerfile best practices

A.3.3 DL3004: Do not use sudo

Code: DL3004
Message: Do not use sudo
Category: SECURITY
Rationale: Do not use sudo as it has unpredictable TTY and signal-forward behaviour that can cause problems. Consider using ”gosu”.
Problematic Code:

```
FROM busybox:<tag>
RUN sudo apt-get install
```

Correct Code:

```
FROM busybox:<tag>
RUN apt-get install
```

Sources: Hadolint Wiki

A.3.4 DL3026: Only use official images as the basis for your images

Code: DL3026
Message: Use only an allowed registry in the FROM image
Category: SECURITY
Rationale:
Problematic Code:

Correct Code:

Sources: Hadolint Wiki, ("Intro Guide to Dockerfile Best Practices - Docker", 2019)

A.3.5 DL9027: Do not use an url with ADD

Code: DL9027
Message: Do not use an url with ADD
Category: SECURITY
Rationale: You are better off using wget or curl as these are dedicated tools to fetch resources from the web. The `ADD` instruction has functionalities that could potentially render this operation unsafe.
Problematic Code:

```
FROM debian:<tag>
ADD https://foobar.com/foo.bar
```

Correct Code:

```
FROM debian:<tag>
RUN wget https://foobar.com/foo.bar
```

Sources:

A.3.6 DL9021: Use HTTPS instead of HTTP

Code: DL9021
Message: Use of HTTP without TLS
Category: SECURITY
Rationale: Use of HTTP is susceptible to man-in-the-middle attacks.
Problematic Code:

```
FROM debian:<tag>
RUN wget http://foobar.com/foo.bar
```

Correct Code:

```
FROM debian:<tag>
RUN wget https://foobar.com/foo.bar
```

Sources: (Rahman et al., 2019)

A.3.7 DL9022: No suspicious comments

Code: DL9022
Message: Suspicious comments
Category: SECURITY
Rationale: Suspicious comments can hint to defects, weaknesses or missing functionality. Therefore, it should not be part of production code.
Problematic Code:

```
# TODO | BUG | FIXME | HACK <whatever>
FROM debian:<tag>
```

Correct Code:

```
# Have no comment explaining vulnerable items, document this internally
FROM debian:<tag>
```

Sources: (Rahman et al., 2019)

A.3.8 DL9023: No hardcoding sensitive information in ARG

Code: DL9023
Message: hardcoded ARGs exposing sensitive information such as usernames, passwords and keys is a no-go
Category: SECURITY
Rationale:
Problematic Code:

```
ARG USERNAME="ADMIN"
ARG PASSWORD="PASSWORD"
ARG KEY="KEY"
```

Correct Code:

```
# Use secrets to leave no trace of sensitive information within Dockerfiles
```

Sources: (Rahman et al., 2019), Dockerfile Secrets

A.3.9 DL9024: No hardcoding sensitive information in ENV

Code: DL9024
Message: Hardcoded ENVs exposing sensitive information such as usernames, passwords and keys is a no-go
Category: SECURITY
Rationale:
Problematic Code:

```
ENV USERNAME="ADMIN"
ENV PASSWORD="PASSWORD"
ENV KEY="KEY"
```

Correct Code:

```
# Use secrets to leave no trace of sensitive information within Dockerfiles
```

Sources: (Rahman et al., 2019), Dockerfile Secrets

A.3.10 DL9025: Use SHA to verify the downloaded file

Code: DL9025
Message: Use SHA to verify the downloaded file
Category: SECURITY
Rationale:
Problematic Code:

```
RUN debian:<tag>
RUN curl -o foo.zip https://foobar.com/foo.zip
```

Correct Code:

```
RUN debian:<tag>
ENV HASH=<hash>
RUN curl -o foo.zip https://foobar.com/foo.zip
# Calculate the hash of the zip and compare it with the official hash value in the ENV variable
```

Sources: (Zhou et al., 2023)

A.3.11 DL9026: Use GPG to verify the downloaded file

Code: DL9026
Message: Use GPG to verify the downloaded file
Category: SECURITY
Rationale:
Problematic Code:

```
TODO
```

Correct Code: **Sources**: (Zhou et al., 2023)

A.4 Layer Optimisation

A.4.1 DL9000

Code: DL9000
Message: Using `npm`, install dependencies before copying source code
Category: LAYER-OPTIMIZATION
Rationale: The `npm install` command can take up a significant amount of time when building a Dockerfile. Whilst this is unavoidable when building a Docker image for the first time, you can significantly decrease incremental build times by leveraging the Docker cache by copying the source code from the build context to the images after installing the dependencies. This way, changes to the source code will not cause the image to install the dependencies in `package.json`.
Problematic Code:

```
FROM node:<tag>
COPY . /
RUN npm install
CMD ["node", "index.js"]
```

Correct Code:

```
FROM node:<tag>
COPY package*.json ./
RUN npm install
COPY . /
CMD ["node", "index.js"]
```

Sources:

A.4.2 DL9020

Code: DL9020
Message: Using `pip`, install dependencies before copying source code
Category: LAYER-OPTIMIZATION
Rationale: The `pip install` command can take up a significant amount of time when building a Dockerfile. Whilst this is unavoidable when building a Docker image for the first time, you can significantly decrease incremental build times by leveraging the Docker cache by copying the source code from the build context to the images after installing the dependencies. This way, changes to the source code will not cause the image to install the dependencies in `requirements.txt`.
Problematic Code:

```
FROM <image>:<tag>
COPY . /
RUN pip install -r requirements.txt
```

Correct Code:

```
FROM <image>:<tag>
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . /
```

Sources: