Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach
Benjamin Benni, Sébastien Mosser, Philippe Collet, Michel Riveill

To cite this version:

HAL Id: hal-01659776
https://hal.science/hal-01659776
Submitted on 8 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientifc research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach

Benjamin Benni
Université Côte d’Azur, CNRS, I3S
Nice, France
benni@i3s.unice.fr

Sébastien Mosser
Université Côte d’Azur, CNRS, I3S
Nice, France
mosser@i3s.unice.fr

Philippe Collet
Université Côte d’Azur, CNRS, I3S
Nice, France
collet@i3s.unice.fr

Michel Riveill
Université Côte d’Azur, CNRS, I3S
Nice, France
riveill@i3s.unice.fr

1 INTRODUCTION

The Service-Oriented Programming (SOP) paradigm has recently evolved to the definition of microservices that cooperate together in a scalable way. Monolithic deployments used until then do not comply with the needs associated with such ecosystem [19]. As part of the microservice paradigm comes the idea of quickly propagating a change from development to production [2], according to a DevOps approach. Development and Operations are no longer separated in the service lifecycle, and a change in a given service can be automatically propagated to production servers through an automated delivery pipeline. In this context, it is up to the service developer to carefully describe how a microservice will be delivered, using dedicated technologies.

Among these technologies, the adoption of the container approach is tremendously increasing [8]. Containers ensure that a given microservice will run the same regardless of its environment, easing the repeatability of build, test, deployment and runtime executions [4, 16]. Containers are faster at runtime and boot-time, lighter than virtual machines, and scale better [9, 17, 20, 25]. In the container field, the Docker engine quickly became the reference platform for builds and deployments of micro-services [23]. It is important to note that the work described in this paper is not tied to Docker, as the defined formal model is technology agnostic. However, for the sake of concision, we decided to mainly focus on the industrial standard as illustration for the challenges and validation case study.

Building a non-trivial container image is a difficult process. Describing an image is an imperative process, where a service deployment descriptor is written (e.g., a dockerfile in the Docker ecosystem) to describe as shell commands how the microservice is installed and configured inside the container. Following an off-the-shelf approach, a container is defined on top of others (reused as black boxes). However, this implementation is not compliant with the open/closed principle, as it is open for extensions (a descriptor extends another one), but not closed for modifications (a descriptor does not provide a clear interface about its contents, making reuse hazardous). By hiding the contents of an image as a black-box, deployment instruction can conflict with the hidden one, e.g., overriding executables, duplicating installation of the same piece of software in alternative versions, or shadowing executables. It leads to erroneous deployments, detected at runtime. Moreover, the technologies supporting microservice deployment evolve constantly,
to make it more efficient or scalable. This evolution can drastically change the way the deployment engine is implemented, and abstraction leaks can occur (i.e., an internal technological choice inside the deployment engine the final user must take into account when writing a service descriptor). It is up to the service developer to stay up to date with ever-changing guidelines that implements fixes to abstraction leaks.

This low level of abstraction make the process of describing containers tedious and unsafe for service developers. In this paper, we propose a static analysis approach that support the safe development of service deployment descriptor by service designers. The originality of the approach is (i) to define a sound formal model that is independent of a dedicated container technology and (ii) to support the definition of an evolving set of checking or rewriting rules while detecting conflict that can occur when applying such rules. The approach is validated on a real-life dataset of 24,357 deployment descriptors hosted on the GitHub open source platform.

2 BACKGROUND & CHALLENGES

In this section, we discuss how containers are supporting microservice deployment, through the prism of the Docker platform, the de facto industrial standard available since 2013. Our contribution can be applied to any container-based system that relies on commands (e.g., LXC [21], rkt [6], Docker [23], Vagrant [13]).

We assume a service developer named Alice, implementing a given microservice, e.g., a product catalog. Using the container approach, Alice will wrap her service inside a turnkey image. This image is built according to a service deployment descriptor (e.g., a dockerfile in the Docker ecosystem), i.e., a script that installs inside the image all the necessary software stack for the product catalog to run (e.g., software dependencies, configuration file, tools). Then, the compiler is compiled into an image, which can be pushed to an image repository, making it available to others. In the Docker ecosystem, the public repository is named the DockerHub, and contained in September 2017 more than 500,000 public images. At the operational level, the images are automatically downloaded from a repository (public or private) and started inside a container. A container can be seen as a light virtual machine, relying on operating system mechanisms at the kernel level to ensure isolation with other containers running on the very same machine.

```bash
# Descriptor: debian:jessie
FROM scratch

# Descriptor: buildpack-deps:jessie-curl
FROM debian:jessie
RUN apt-get install curl wget ...

# Descriptor: buildpack-deps:jessie-scm
FROM debian:jessie
RUN apt-get install bzip2 git ...

# Descriptor: openjdk:8-jdk
FROM buildpack-deps:jessie-scm
RUN apt-get install bzip2 ...

# Descriptor: jenkins:latest
FROM openjdk:8-jdk
RUN apt-get install git curl ...

CMD ["/usr/local/bin/jenkins.sh"]
```

Listing 2: Dockerfiles hierarchy example

```bash
FROM alpine:latest
MAINTAINER Alice <alice@awesome-services.cc>
RUN apt-get update
RUN apt-get install nodejs npm
RUN npm install express mongoose
WORKDIR /catalogue-service
ADD config.json *.js 
CMD ["nodejs"]
```

Listing 1: Product catalogue descriptor

To create an image, Alice creates a descriptor (List 1), where she assembles setup instructions for her product catalog. She starts by installing all the software stack needed by her service, e.g., the NodeJS software stack and the associated dependency manager NPM (L3-4). She also installs the javascript packages needed by her service, i.e., express for the service exposition and mongoose to connect to a MongoDB database. Then, she moves to a directory named catalogue-service, add a configuration file located on her filesystem and the javascript source code into this directory, and finally starts the nodejs environment to host her service. It is a good practice to publish the descriptor associated to a given image on the repository, and tools exist to decompile an image and retrieve the essential instructions used to build it.

An important line to notice is the first one, where Alice states that her image is built on top of the alpine:latest image. This allows her to not worry about the details of how to setup an operating system inside her container, by simply reusing the Alpine (a lightweight linux distribution, weighting 8Mb and known to start quickly) image available on the repository. This black-box reuse mechanism is the key of any deployment technology, allowing one to reuse off-the-shelf elements. In Docker, the hierarchy is recursive, until reaching the scratch root image. We describe in List. 2 an example of the hierarchy associated to the Jenkins continuous integration service official image, involving a descriptor and 4 successive ancestors. The complete hierarchy is available on the companion webpage of this paper.

```bash
FROM jenkins:latest
RUN apt-get install npm bzip2=1.0.1
CMD ["nodejs"]
```

Listing 3: Service deployment descriptor (bad reuse)

The mechanisms under the container approach triggers the following two challenges with respect to microservices deployment.

Following evolving guidelines (C1). The container approach was recently adopted by the industry (e.g., Docker started in 2013, even if the container underlying technology exists in the linux kernel since 2008). This effervescent context makes features available in

1 https://hub.docker.com/r/centurylinklabs/dockerfile-from-image/
tools added at the same rate obsolete ones are removed, and suffer from abstractions leaks. For example, the descriptor described in List. 1 violates a Docker guideline: the update command is not executed in the same RUN as the installation one (see Sec. 4.3 for details). This guideline, among others, (i) is the visible side of an internal flaw of the engine, (ii) must be followed until now, but (iii) might be removed in future version.

**Safer black-box reuse (C2).** The strength of image reusing is also a strong weakness of the container approach. Following the open/closed principle, an image is "open for extension", and "closed for modification". But as the interface of the image is not defined and only considered as a black box with no clear interface contract, when reusing an image, Alice has no idea of the contents of the reused image. It is then possible for Alice to override elements existing in the source image, without knowing it. These errors cannot be detected at build time for an image (e.g., installing a piece of software in a conflicting version, like the bzip2 package install in List. 3 that conflicts with the one installed in List. 2, 1.16), and triggers errors when running the deployed services.

### 3 Modeling Descriptors

This section describes the formalism defined to model container descriptors and support (i) the static analysis of a given descriptor and (ii) the rewriting of a descriptor to fix common errors when possible. Considering the diversity of existing containers platform, this model must be technology agnostic. We will show in Sec. 4 how to instantiate the instruction concept to fit the Docker platform.

#### 3.1 Formal Model

A service deployment descriptor is implemented as a sequence of shell commands, so in our formalism a descriptor \( d \in D \) as a totally ordered set of setup instructions (e.g., running a shell command, copying a configuration file, denoted as \( I_d \in I \)). To model the relationship that may exist between two descriptors (e.g., a `dockerfile` extends another one, a shell script loads another one with the source command), we define a function named parent that returns for a given descriptor its parent, or the root descriptor (denoted as \( \emptyset \)) if no parent exists for this descriptor (Eq.1). This enables to define a normalized version of a deployment descriptor \( d \), noted \( \overline{d} \), which contains all instructions in order from \( d \) to the top of the parent hierarchy (Eq.2).

This model is simple, but expressive enough to support the definition of checkers and rewriting rules. For a given service deployment platform, it needs to be instantiated at the instruction level, i.e., which kind of setup instructions are available for this very platform.

\[
\begin{align*}
&d \in D = \{i_1, \ldots, i_n\} \in I^2_d \\
&\text{parent} : D \rightarrow D \\
&d \mapsto d' : d \text{ loads } d'
\end{align*}
\]

(1)

\[
\begin{align*}
&\vdash D \times D \rightarrow D \\
&(d_1, d_2) \mapsto d_{12} : \text{Let } d_1 = [i_1, \ldots, i_m], \\
&\quad d_2 = [i'_1, \ldots, i'_n], \\
&\quad d_{12} = d_1; d_2 = [i_1, i_n, i'_1, \ldots, i'_m] \\
&\quad \land \text{parent}(d_{12}) = \text{parent}(d_1) \\
&\quad \land \text{parent}(d_{12}) = \text{parent}(d_1) \\
&\quad \\
&\vdash D \rightarrow D \\
&d \mapsto \overline{d} = \left\{ \begin{array}{l}
\text{parent}(d) = \emptyset \Rightarrow \overline{d} \\
\text{parent}(d) \neq \emptyset \Rightarrow \overline{\text{parent}(d)}; \overline{d}
\end{array} \right.
\end{align*}
\]

(2)

#### 3.2 Checking rules: \( \Phi \)

The intention of a checking rule (or checker) is to statically identify an error that exists in a given deployment describe. A checker is formally defined as a function \( \phi \) taking as input a descriptor and returning a boolean stating whether the detect is detected. Thus, for a given platform or a given company, one can model the set of guidelines relevant for her context as a set of rules to be checked (rules = \( \{\phi_1, \ldots, \phi_n\} \)). This is classic when defining a lint (a static analyzer), where users can define their own set of rules. Considering the composition operator defined previously, the strength of the proposed approach is to support the application of a checking rule to the normalized version of the deployment descriptor, allowing one to identify an error that comes from an interaction between the current deployment instructions and the one inherited from the parent hierarchy (Eq.3). In addition to being technology-specific, state of the art deployment linters do not provide a way to leverage this composition, and only provides a static analysis of the current descriptor.

\[
\phi_i : D \rightarrow B \in \Phi \\
\text{violation?} : D \times \Phi^n \rightarrow B \\
(d, \{\phi_1, \ldots, \phi_n\}) \mapsto \bigvee_{i=1}^n \phi_i(d)
\]

(3)

#### 3.3 Automated Rewriting rules: \( R \)

Checkers support the identification of errors that can be automatically detected. It is then up to the writer of the service deployment descriptor to fix it. However, for some errors, it is possible to rewrite the descriptor to fix it in an automated way. For example, to reduce deployment artifact size, reducing the number of instructions in a descriptor helps (this is inherent to the container technology, where each instruction adds an overhead to the final size). One can write a rule detecting instructions that can be merged together. It is then possible to automatically compute how the descriptor should be rewritten. Unfortunately, rules can overlap and conflict in their decisions. For example, to augment modularity and reuse potential of a service deployment descriptor, keeping the instructions as separated as possible is a good practice. This clearly overlaps with the previous intention of reducing artifacts size. In the context of ever-changing guidelines associated to containers, it is important to automatically detect such conflicts.
To address this issue, we consider here a rewriting rule \( \rho \in \mathcal{R} \) as a function taking as input a descriptor \( d \), and producing as output a patch to apply to the descriptor (i.e., a delta) to make it compliant with the guideline, and denoted as \( \delta \in \Delta \). The obtained \( \delta \) models how the descriptor must be modified (by changing the instruction sequence) to achieve the rewriting [18]. By reasoning on the set of deltas \( \{\delta_1, \ldots, \delta_n\} \) obtained when multiple rules must be applied, one can automatically identify conflicts. Considering the previous example of instruction squashing versus modularity, one can identify a conflict as the two rules would produce \( \delta \)s that would concurrently modify the same instructions in different ways. Inspired by Stickel’s work [24], a \( \delta \) is defined as a set of substitution pairs \( (i \to i') \in \Sigma \). Applying such a substitution to a descriptor \( d \) means to produce a new descriptor \( d' \) where \( i' \) replace \( i \) in the ordered set of instructions. To remove an instruction means to substitute \( i \) by void and to introduce a new one at the beginning of a sequence means to substitute void by this instruction \( (0 \to i') \). To identify conflicts between modifications, we look for concurrent substitutions that might alter the very same instruction.

\[
\begin{align*}
\text{do} : D \times \Sigma & \to D \\
(d, i) \mapsto d' & : \text{let } d = [\ldots, i_n, i, i_m, \ldots], \\
\sigma & = (i \to i') \\
d' & = [\ldots, i_n, i', i_m, \ldots] \\
d_0^+ : D \times \Delta & \to D \\
(d, \delta) \mapsto \delta & = \emptyset \Rightarrow d \\
\delta & = (\sigma) \cup \delta' \Rightarrow d_0^+(do(d, \sigma), \delta') \\
\text{conflict} : \Delta & \to \mathcal{E} \\
\delta & \mapsto \exists (i, a, b) \in I^3, i \neq \emptyset, (i \to a) \in \delta, (i \to b) \in \delta, a \neq b \neq \emptyset \\
\end{align*}
\]

Considering this representation of \( \delta \)s that support conflict detection and the functional representation of rewrite rules described in the previous paragraph, optimizing a given descriptor \( d \) by applying several rewriting rules \( \rho_i \) means to compute all the \( \delta \)s associated to the given rules, verifying the absence of \text{conflict} and then applying it. Like the checking mechanism, the rewriting mechanism \text{rw} benefits from the composition operator and it considers the complete hierarchy, being applied to the normalized version \( d' \) of the descriptor \( d \).

\[
\begin{align*}
\rho_i : D & \to \Delta \in \mathcal{R} \\
rw : D \times \mathcal{R}^n & \to D | \text{Error} \\
(d, \text{rules}) \mapsto d' & : \text{let } \delta = \bigcup_{\rho \in \text{rules}} \rho(\delta), \\
d' & = \begin{cases} \text{conflict}?(\delta) & \Rightarrow \text{Error} \\
\neg\text{conflict}?(\delta) & \Rightarrow d_0^+(\delta, \delta) \\
\end{cases}
\end{align*}
\]

4 APPLICATION: THE DOCKER CASE

In this section, we refine the formal model presented in the previous section to fit the Docker container platform, i.e., refining the available instructions and implementing checkers and rewriters associated to this service deployment environment. Among the guidelines defined by the Docker best practices reference, we focus here on three rules from a qualitative point of view, to show how the formal framework can be applied. A quantitative evaluation is performed in Sec. 5.

To refine the formal model and adapt it to the Docker platform (defining \( L_d \subseteq I \)), we need to define what instructions are available for the rules to work with. The complete model is available on the companion webpage, as well as the associated parser that compiles a \textit{Dockerfile} into an instance of the model. We focus here only of the kind of commands necessary to illustrate the checking and rewriting rules described in this section. We consider an instruction as a kind (e.g., RUN, CMD), and a totally ordered set of arguments passed to the command. One can access to the kind of an instruction using a function named \textit{kind}, and to the arguments using a function named \textit{args}. A special kind \textit{MULTI-RUN} is used to model the RUN instructions containing multiple shell commands (separated by ampersands).

4.1 Checker: Overriding Services

Let us consider here the service deployment descriptor described in List. 3. From the writer point of view, to build a continuous delivery pipeline, we reuse the latest Jenkins official image and simply add an homemade bridge service (implemented as a node.js artifact) to interconnect the Jenkins instance with some internal tools. Unfortunately, starting our bridge service (List. 3, line 6) will override the \textit{CMD} command that exists inside the Jenkins container we are reusing (List. 2, line 22). As a consequence, only our bridge service is started, and considering the blackbox reuse mechanism advocated by Docker, there is no way for the descriptor writer to know why the Jenkins instance does not start. It is up to the user to break the black-box approach by hand and crawl the official Docker store to analyze the root cause of the error by going through the hidden hierarchy.

We are defining here a checker instead of a rewriter, and it is up to the user to decide which service to start\(^1\). Given a descriptor \( d \), the checker \( \varphi_{os} \) looks inside the descriptor \( d \) for a pair of deployment instruction \((i, i')\) starting two different services.

\[
\varphi_{os} : D \to \mathcal{E} \in \Phi \\
d \mapsto \exists (i, i') \in I^2, i < i' \\
\land \text{kind}(i) = \text{kind}(i') = \text{CMD}
\]

4.2 Rewriter: Reducing Number of Layers

We discussed earlier the issue of optimizing an image size by reducing the number of instructions used to build it (Sec. 3). This is a use case for the rewriting mechanism, through the implementation of a set of rules that identify useless instructions and rewrite the service deployment descriptor adequately. The rewriter looks for equivalent instructions (considering a given equivalence relation \( \equiv \))

\(^1\)Under some assumptions, one can imagine generating a script that starts the conflicting services simultaneously and rewrite the conflicting commands. But this is not true in the general case, as for example some services might use the same network port.
inside a given descriptor, keep one and delete the others.

\[
\rho_m : D \rightarrow \Delta \in R
\]
\[
d \mapsto \delta : \text{Let } Is = \{i : i \in d, \exists i' \in d, i \equiv i'\},
\]
\[
\delta = \begin{cases} 
    Is = \emptyset & \Rightarrow \emptyset \\
    Is = \{i\} \cup Is' & \Rightarrow \{(i') : i' \in Is'\}
\end{cases}
\] (7)

The difficulty here is to define the equivalence class that exists among setup instructions. As shell commands have side effects by design, even two consecutive invocations of the very same instruction might not be equivalent. For example, (i) creating a directory named dir, (ii) moving to another location and (iii) creating another directory named dir. The two instructions used to create the directory cannot be merged into one. However, the download of the same file from the web (e.g., an application server using the wget tool) can be detected and unified (Eq.8).

\[
\equiv_d : I_d \times I_d \rightarrow \mathbb{B}
\]
\[
(i, i') \mapsto \text{kind}(i) = \text{kind}(i') = \text{RUN}
\]
\[
\wedge \text{wget} \in \text{args}(i) \wedge \text{wget} \in \text{args}(i')
\]
\[
\wedge \exists \text{url} \in \text{URL},
\]
\[
\text{url} \in \text{args}(i) \wedge \text{url} \in \text{args}(i')
\] (8)

Another way of reducing an image size is to merge instructions that can be merged together, even if their arguments differ. This is classical with package management tools, such as NPM in javascript or APT at the linux operating system level. In List. 4, the first descriptor uses three instructions to install three javascript packages used by the service, where the second descriptor using a single instruction to install the packages. This is particularly important as the writer can be unaware of the fact that some packages were installed in a parent deployment descriptor, leading to multiple downloads in addition to overweighted service container image.

```bash
# Multiple RUNs to install several NPM packages
#RUN npm install soap
#RUN npm install json-schema-napper
#RUN npm install xml2js
#
# Merged instruction
#RUN npm install json-schema-napper soap xml2js
```

Listing 4: Merging instructions arguments

To achieve such rewriting, we define the \(\rho_{pack}\) rule that catches all the instructions \(i \in Is\) using the expected package manager (here npm, but apt, opam, yum and other package managers follow the same principle), and (i) creates a merged instruction \(m\) that installs all packages, (ii) substitutes the first instruction for the merged one (Rewritten), while (iii) removing the previously existing instructions (Removed).

\[
\text{install}?: I_d \rightarrow \mathbb{B}
\]
\[
i \mapsto \text{kind}(i) = \text{RUN} \wedge \text{npm} \in \text{args}(i)
\]
\[
\wedge \text{install} \in \text{args}(i)
\]

\[
\rho_{pack} : D \rightarrow \Delta \in R
\]
\[
d \mapsto \delta : \text{Let } Is = \{i : i \in d, \text{install}(i)\},
\]
\[
\delta = \begin{cases} 
    Is = \emptyset & \Rightarrow \emptyset \\
    Is = \{i\} \cup Is' & \Rightarrow \{(i') : i' \in Is'\}
\end{cases}
\] (9)

\[
\text{Rewritten} = \{\text{args}(i)_d : \text{install}(i)\} \cup \text{packages}
\]
\[
m = \text{run}(\text{npm, install})
\]

4.3 Checker & Rewriter: Layer Caching

From the point of view of the service deployment descriptor, executing two instructions containing one command should be equivalent to executing a single instruction containing the previous command. However, due to an internal optimization made by the Docker engine called layer caching\(^4\), this is not the case. The prototypical example of such an error is the usage of the apt-get package mechanisms to install the software stack supporting the service to deploy. It is classical to first update the package source before installing the pieces of software needed to deploy a given service (see List. 5).

```bash
# Single RUN with multiple commands
#RUN apt-get update
#RUN apt-get install nginx nodejs
#
# Merged instruction
#RUN apt-get install nginx nodejs
```

Listing 5: Multiple & Single RUN instructions

If a `dockerfile` containing an apt-get update command has already been executed on the computer, the engine will use the associated layer instead of running the update again, leading the apt-get install instruction to work with outdated packages. We show in this section how the situation can be detected using a checker, and how it can be rewritten automatically. This illustrates the expressiveness of both checking and rewriting rules, showing how the two principles apply to the same use case. We first define two functions \(\text{apt}_i\) and \(\text{apt}_u\) to identify the instructions involved in this section (Eq.10).

\[
\text{apt}_i?: I_d \rightarrow \mathbb{B}
\]
\[
i \mapsto \text{kind}(i) = \text{RUN} \wedge \text{apt-get} \in \text{args}(i)
\]
\[
\wedge \text{install} \in \text{args}(i)
\]

\[
\text{apt}_u?: I_d \rightarrow \mathbb{B}
\]
\[
i \mapsto \text{kind}(i) = \text{RUN} \wedge \text{apt-get} \in \text{args}(i)
\]
\[
\wedge \text{update} \in \text{args}(i)
\] (10)

\(^4\)https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

run.
4.4 Conflict detection

We consider here (i) the rewriters defined in the previous sections to support layer caching and image weight reduction (the latter adapted to work with apt instead of npm) and (ii) the deployment descriptor given in List. 6, and denoted as $d_c$.

- $\Delta_{\text{pack}} = \rho_{\text{pack}}(d_c)$. The rewriter catches the two installation instructions $i_2$ and $i_3$, creates a merged one denoted as $i_{23}$ that installs the two packages, substitutes $i_2$ for the merged instruction and removes $i_3$.

- $\Delta_{\text{gap}} = \rho_{\text{gap}}(d_c)$. The rewriter catches the three instructions, and merges them in a multi-run instruction denoted as $i_{123}$. The update instruction is substituted for $i_{123}$, and the others are removed.

- $\Delta = \Delta_{\text{pack}} \cup \Delta_{\text{gap}}$. The conflict detection function is applied on the union set containing all the expected substitutions. When triggered, it identifies a conflict on $i_2$, as the first rule wants to remove it when the second one would substitute with something else.

$$\exists (i_2, \emptyset, i_{23}) \in I^3, (i_2 \rightarrow \emptyset) \in \Delta \land (i_2 \rightarrow i_{23}) \in \Delta$$

$$\Delta_{\text{pack}} = \{(i_2 \rightarrow i_{23}), (i_3 \rightarrow \emptyset)\}$$

$$\Delta_{\text{gap}} = \{(i_1 \rightarrow i_{23}), (i_2 \rightarrow \emptyset), (i_3 \rightarrow \emptyset)\}$$

$$\Delta = \{(i_1 \rightarrow i_{23}), (i_2 \rightarrow \emptyset), (i_2 \rightarrow i_{23}), (i_3 \rightarrow \emptyset)\}$$

5 QUANTITATIVE VALIDATION

In this section we validate that (i) our proposition can handle a real-world use-case in an acceptable amount of time, (ii) analyzing the composition of dockerfiles generates a reasonable overhead (iii) there is a substantial gain obtained in detecting guideline violations when taking the whole hierarchy into account instead of isolated deployment descriptors.

To do so, we built a dataset of dockerfiles available on GitHub, the community code-versioning reference platform. We collected an initial set of 24,357 deployment descriptors. Details about this collection, the composition, and the building of the dataset are available in the companion webpage. Over these dockerfiles, 5.8% (1,412) were considered as trivial (i.e., having less than 3 instructions\(^1\)) and were removed from the dataset. The remaining 22,945 dockerfiles regroup 178,088 instructions and represent our experimental dataset, denoted DS. The normalized version of our dataset, denoted DS, is made of also 22,945 dockerfiles but due to extension mechanism, it regroups 285,142 instructions.

Isolated dockerfiles of DS contain between 3 and 202 instructions with 7.76 instructions per dockerfile on average; normalized dockerfiles of DS have between 3 and 202 instructions, with 14.37 on average. The smallest sizes are the same since it is our lower-threshold for trivial dockerfile. The highest size is a single dockerfile that is bigger than every normalized dockerfiles. The most interesting metric here is that dockerfiles double in size on average. Normalized dockerfiles of DS have between 0 and 6 parent dockerfiles with 1.45 level of parents on average. These numbers show the scale our approach must reach to address real life deployment descriptors.

From the guidelines in the official Docker webpages\(^2\), we identified and implemented 15 of them as they are both highly used by the community and general enough to be relevant for a wide number of dockerfiles.

5.1 Analyzing Atomic Descriptors

The remaining 22,945 dockerfiles in DS regroup 178,088 instructions. In this dataset, no parent-child relationship exists and each dockerfile is analyzed as an isolated descriptor.

Isolated dockerfiles contain between 3 and 202 instructions with 7.76 instructions per dockerfile on average. We first applied each guideline on our experimental dataset to show the impact of our contribution on the build-execution time. We underline the fact that, in this case, each dockerfile is considered as isolated from others, and no parent-child relationship is taken into account. The output of this experiment can be considered as the raw execution time of our analyzer on thousands of unrelated files.

\(^1\) A parent reference and a single command.

Listing 6: Descriptor triggering a conflict: $d_c$

```bash
RUN apt-get update
RUN apt-get install nginx
RUN apt-get install nodejs
```
Measures were made with the JMH\textsuperscript{7} framework in its 1.5.2 version and targeting Java 8 code. Each measure is made 10 times, after a warm-up phase of the JVM used to reduce measure errors and garbage collection interferences. We ran this benchmark on a virtual machine running CentosOS 7 operating system, with one Intel Xeon E5-2637v2 3.5GHz of 4 cores, and 4 GB/1600MHz of RAM. The benchmark code, as well as more details on the experimental protocol can be found in the companion webpage.

Fig. 1 displays the execution times running our 15 official guidelines over the 22,945 dockerfiles, both in isolation and with the normalized set we discuss in the next section. In isolated mode the analyzer takes around 4 seconds to get through our dataset and yield conflicts. This execution time must be compared to the amount of time needed to list, load and parse those dockerfiles (2, 5 seconds on average). We consider that this time is perfectly compatible with a real-life build-chain of nowadays software companies (e.g., continuous deployment pipeline).

5.2 Analyzing Normalized Descriptors

To show the value of our contribution and our normalized operator, we need to automatically build hierarchies of dockerfiles, but there is no pre-established or accessible parent-child relationship with these files. This could have been automatically possible if (i) a dockerfile was a named artefact, and (ii) a named dockerfile could be accessible via an API, which is actually not the case.

As our dataset was built without targeting any specific user, dockerfile, embedded framework or official dockerfile, we analyzed it and sorted dockerfiles by the most used parents. We then handle aliases since the very same parent-dockerfile can be extended by child-dockerfiles using different names. Fig. 2 depicts the evolution of the percentage of the dataset which would have a parent/child relation established if the parent was known. The resulting curve is not surprising since a lot of dockerfiles extends the same set of dockerfiles (i.e., the official ones).

Since dockerfiles have to be manually retrieved, we decided to cover 50% of our dataset with an established parent/child relationship. To do so, we manually retrieved the 40 most extended dockerfiles, from common repositories (e.g., DockerHub, DockerStore).

As a result, we have 11,527 dockerfiles with a known parent. We believe that this coverage is relevant enough to serve our purpose and that manually grabbing more dockerfiles will not change our conclusions.

Execution time. We ran the same guidelines as in Sec. 5.1 on the whole dataset, this time using the normalized operator. We can apply this operator even if no parent/child relationship has been established\textsuperscript{8}. Therefore we applied our guidelines on the normalized version of all the dockerfiles, some having parents, some not.

The upper part of Fig. 1 displays the execution times running the 15 official guidelines, this time over DS. Our analyzer took around 6.5 seconds to analyze our experimental dataset and yield conflicts, which contains 2.5 seconds of file loading. We consider that this time is also perfectly compatible with a real-life pipeline of nowadays companies. We also note that, since dockerfiles have between 0 and 6 parent dockerfiles with 1.45 level of parents on average, and that they have between 3 and 202 instructions with 7.76 instructions per dockerfile on average, our contribution fits well the relative constrained complexity of our targeted problem.

Guideline violation. Fig. 3 shows how many dockerfiles are detected as violation of a given guideline. We note that guidelines G3, G4, G5, G9, G10 and G11 are violated the same amount of times, which is very low. This is due to the fact that (i) those errors are rarely made and (ii) are more likely to be made by beginners (i.e., at the bottom of the hierarchy). We also note that the 9 remaining guidelines are more violated when applying the normalized operator. This difference corresponds to guidelines violation that cannot be detected without taking the normalized descriptor into account, as our approach does.

Fig. 4 shows how many commands are detected as violating a given guideline in isolated and normalized modes (using a logarithmic scale). We note that patterns from Fig. 3 are found in this figure. This gives insights about how many times a dockerfile violates a given guideline and therefore that a high-level dockerfile has a given flaw. For instance, guideline #2 is violated by a very small amount of instructions, which impacts a lot of dockerfiles. This means that fixing a small amount of instructions may fix a lot of dockerfiles.

5.3 Conflict detection

Some guidelines are going to conflict with each other, by construction (e.g., updating before installing, and adding specific arguments to at --get command). Half of the extracted guidelines target RUN commands, hence are more likely to be conflicting. An interference matrix of conflicting guidelines can then be built.

\textsuperscript{7}http://openjdk.java.net/projects/code-tools/jmh/.  

\textsuperscript{8}as described in Eq.2: parent(d) = ∅ ⇒ d.
Figure 3: Number of dockerfiles violating a given guideline

Table 1: Dockerfiles containing guidelines violation pairs

<table>
<thead>
<tr>
<th>Gi,j</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124</td>
<td>2,212</td>
<td>2,901</td>
<td>1,731</td>
<td>2,810</td>
<td>1,816</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>524</td>
<td>685</td>
<td>176</td>
<td>436</td>
<td>629</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8,492</td>
<td>2,110</td>
<td>6,331</td>
<td>6,531</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2,601</td>
<td>8,690</td>
<td>10,223</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2,351</td>
<td>1,861</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5,965</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Instructions containing guidelines violation pairs (i.e., real conflicts)

<table>
<thead>
<tr>
<th>Gi,j</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>1,795</td>
<td>5,445</td>
<td>4,100</td>
<td>3,509</td>
<td>671</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>360</td>
<td>12</td>
<td>174</td>
<td>314</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20,094</td>
<td>1,211</td>
<td>9,155</td>
<td>14,655</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5,239</td>
<td>19,474</td>
<td>50,256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1,815</td>
<td>1,211</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8,680</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Number of instructions violating a given guideline

Table 1 shows the number of dockerfiles that present potential conflicts for each guideline pair. We represent only the upper right part of the matrix, as it is symmetric by construction. There is a potential conflict when two guidelines are violated on the same dockerfile \( d \) and target the same kind \( k \) of commands. For example, 8,492 dockerfiles violate guideline 6 and guideline 7, whereas only 124 dockerfiles violated guideline 1 and guideline 5. Two issues can occur on a single dockerfile but on different instructions and therefore create no new conflict, but this information is not available on Table 1.

Table 2 shows the number of instructions that are really conflicting, i.e., conflicts occurring on the same instructions of the same dockerfile and producing different results. We note that some guidelines pairs (e.g., \( G_1 \) and \( G_5 \)) are violated on many dockerfiles (1816) but that only 671 instructions are really in conflict; whereas others (e.g., \( G_6 \) and \( G_7 \)) are violated on around 8,500 dockerfiles, and that more that 20,000 instructions are really conflicting. This result shows that these guidelines are often violated together, on a lot of instructions, exposing an understanding problem of the platform by the service designers.

5.4 Rewriting

Finally, we applied two rewriting rules on our normalized dataset to show the benefit of an automated analysis and automated rewriting of service descriptors. We remember that \( DS \) is made of 22,945 dockerfiles, grouping 285,142 instructions. Among \( DS \), 10,711 dockerfiles (46.68%), grouping 21,686 RUN instructions (7.60%), install packages via the apt manager. These instructions represent 26.84% of the 80,799 RUN instructions of our dataset, confirming that installing packages with apt-get is a pretty common operation.

Layer caching. We implemented the rule defined in Sec. 4.3. We found that 19,309 instructions violate this guideline. This means that close to 89% of the analyzed descriptors install software without properly updating their dependencies first. These 19,309 flawed commands are spread over 8,496 dockerfiles, which represent 79.32% of the descriptors using this package manager, and 37.02% of \( DS \). This means that 3 out of 4 dockerfiles that installs software using apt are introducing a flaw that leads to outdated dependencies. It represents more than a third of our entire experimental dataset emphasizing how dangerous this abstraction leak is. Thanks to our automatic rewriting system, all these errors have been automatically detected and fixed by appending the proper arguments into the body of the targeted instructions.

Reducing number of layers. As we said in Sec. 4.2, the weight of the final artifact is bound to its number of instructions, i.e., the less instructions are in the dockerfile, the lighter the artifact and vice-versa. By running the described rewrite rule on our dataset, we found that 79,045 RUN instructions can be automatically deleted, lightening the weight of concerned dockerfiles and of their children.
This means that 97.83% of the RUN commands of our dataset (which contains 80,799 RUN instructions in total) can be safely deleted, lightening the size of artifacts. This is due to the fact that developers assume a 1 → 1 mapping between commands executed on a classic shell and what happens inside a dockerfile at build time, pictured in List. 4. Again, thanks to our automatic rewriting system, all these RUN have been automatically detected and merged together by merging their instructions.

6 RELATED WORK

The distributed system community faces the challenge of deploying multi-tenant pieces of software since decades. Automated approaches have been proposed to support the scripting (using both declarative [10] or imperative [7] descriptors) of distributed deployments, for example using dedicated architecture deployment languages [11]. These academic approaches were complemented by industrial implementations during the rise of the cloud era, with systems such as Chef [15] or Puppet [1], to orchestrate different sequences of shell commands to support distributed deployment. These systems also propose a black-box reuse mechanism that can be used to address the difficulty of writing a dockerfile by providing domain-specific languages (DSLs [12]) dedicated to this task, written in Ocaml9, Javascript10, or Go languages. Using these languages enables the static analysis of a given descriptor, and the safe generation of a valid dockerfile.

Rockers11 is a tool that adds features to ease the writing of dockerfiles. It provides a higher-level language to avoid many mistakes made when writing docker descriptors. Even if it does not provide reasoning on dockerfiles, it shows how difficult the process of writing a dockerfile can be.

The community-linter system12 supports the analysis of a given dockerfile with respect to the referenced best practices listed by the Docker company. However, contrary to our contribution, these approaches do not support the analysis of the whole hierarchy associated to the dockerfile, are focused on syntactical errors only (e.g., missing or extra symbol in command body), and are not capable of detecting conflicts in the file.

An anti-pattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences [5]. An anti-pattern describes a general form, symptoms describing how to recognize the general form, its consequences, and a refactored solution describing how to change the anti-pattern into a healthier situation [5]. An anti-pattern can be modeled as rule that checks itself against a model and proposes, if applied, a refactoring. Therefore, the rules described in this paper can be considered as anti-pattern detection for micro-service deployment descriptors.

7 CONCLUSIONS & PERSPECTIVES

In this paper, we illustrated how crucial the deployment descriptor are in the micro-service development community, and the associated challenges for service designers. We identified two crucial challenges in this context: (i) how to support an ever-changing set of guidelines when writing descriptors (C1) and (ii) how to deal with issues introduced by the black-box reuse mechanism associated to containers (C2). To address the latter challenge, we defined a formal model in a technology independent way, reifying a composition operator that leverages the image reusing mechanisms to build normalized descriptors. To address the former one, we described how the framework can be instantiated to fit the Docker platform specificities. Moreover, a conflict detection mechanism defined at the formal level using first order logic supports the detection of overlapping rules, addressing both challenges. We evaluated this contribution on a set of more than 20,000 real deployment descriptors, showing the benefits of the normalized approach to identify hidden errors in designers’ descriptors.

To pursue this work, we plan to address two limitations encountered by this contribution. The first limitation comes with the substitution mechanism: even if it seems to us simple to understand and use, its expressiveness is limited and makes the implementation of complex rewriting rules tedious. We plan to extend it by moving from logical terms substitutions to a graph algebra, and reason on more precise operations. Another limit to address this perspective is to rely on the Praxis approach [3] to reason on descriptor modifications rather than on substitutions. The second perspective to address is the definition of a traceability model associated with service descriptors. For now, the approach works at the normalized descriptor level, for checking and rewriting, and consider it as a whole. Scattering the rewriting to several descriptors, as well as identifying the root causes of the rules violations or conflicts will help the maintenance of the service descriptor hierarchy. Finally, from a service engineering point of view, measuring the impact of the other tools associated with containers to support micro-service scaling is an interesting field which is not addressed by the scientific literature.

REFERENCES


