Ten Simple Rules for Writing Dockerfiles for Reproducible Research

Daniel Nüst [*], Vanessa Sochat , Ben Marwick , Stephen J. Eglen , Tim Head , Tony Hirst

[*] Corresponding author: daniel.nuest@uni-muenster.de

Abstract

Containers are greatly improving computational science by packaging software and data dependencies. In a scholarly context, transparency and support of reproducibility are the largest drivers for using these containers. It follows that choices that are made with respect to building containers can make or break a workflow's reproducibility. The build for the container image is often created based on the instructions in the `Dockerfile` format. The rules presented here help researchers to write understandable `Dockerfile`s for typical data science workflows. By following the rules in this article researchers can create containers suitable for sharing with fellow scientists, for inclusing in scholarly communication such as education or scientific papers, and for an effective and sustainable personal workflow.

Introduction

Computing infrastructure has advanced to the point where not only can we share data underlying resarch articles, we can share the code that processes these data. The sharing of code files is enabled by collaboration platforms such as GitHub or GitLab and has become quite common. The sharing of the computing environment is enabled by containerisation, but this possibility is not widely used yet. Containerisation allows for documenting and sharing entire workflows in a comprehensive way, which is desperately needed to increase reproducibility of computational resarch. Computational research is oftentimes way too complicated for "papers" based on the traditional journal article format to fully communicate the details of research [1], but where the actual contribution to knowledge includes the full computing environment that produced a result [2].

A solution for packaging the building blocks of computer-based research, i.e. code, data, documentation, and the computing environment, is the *research compendium* (cf. `https://research-compendium.science`). A research compendium greatly improves the transparancy and reproducibility of our research, because we not only publish abstract algorithms and study designs, but also the instructions for recreating and executing whole research workflows. Within a research compendium, it is desirable to include instructions, i.e. a human- *and* machine-readable recipe, for building containers that capture the computing environment. The computing environment comprises all software and data dependencies, as these are not captured by the the typical workflow script or documentation in a README. By providing this recipe, authors of scientific articles greatly improve their work's level of documentation, transparency, and reusability. Such practice is one important part of common practices for scientific

computing [3,4], with the result that it is much more likely both the author and others are able to reproduce and extend an analysis workflow. The containers built from these recipes are portable encapsulated snapshots of a specific computing environment. Such containers have been demonstrated for capturing scientific notebooks [5] and reproducible workflows [6]. Research compendia also allow for proper citation of the used computing environment, which is not possible within containers alone. Best practices are still a work in progress [7], but you should try your best to give credit to creators of software you rely on by following recommendations of projects such as CodeMeta (`https://codemeta.github.io/`) and the Citation File Format (`https://citation-file-format.github.io/`).

While there are several tutorials for using containers for reproducible research [8–12], there is no detailed *manual for how to write the actual instructions to create the containers for computational research* besides generic best practices [13]. This article introduces these instructions for the popular `Dockerfile` format with a focus on containers encapsulating data science workflows.

Prerequisites & scope

To start with, we assume you have a scripted scientific workflow, i.e. you can, at least at a certain point in time, execute the full process with a fixed set of commands, for example `make prepare_data` followed by `Rscript analysis.R`, or only `python3 my-workflow.py`. If you have an ordered set of commands building upon each other, you can write a main script that does just that, even if you in your daily work might not do that to save time. A key constraint is that containers you eventually share with others as part of a research compendium can only run open source software. Therefore tools like Mathematica and Matlab are out of scope of this work. This execution should be able to be triggered by command-line instruction, which is possible for all generic programming languages and widely used workflow tools, including tools primarily used interactively and/or with a graphical user interface. A workflow that does not support scripted execution is out of scope for reproducible research, and does not fit well with containerisation.

Furthermore, workflows interacting with many petabytes of data and executed in high-performance computing (HPC) infrastructures are out of scope. While containers *can* be helpful for analysing data at scale, the complexity of theses pipelines, e.g., using HPC job managers or cloud infrastructures, requires a collection of "Ten Simple Rules" articles in their own right. Instead, we focus on workflows that typically run on single machine, e.g., a researchers laptop computer or a virtual server, though they may take some time and resources to execute, such as a terabyte of data and 16 cores running over the weekend.

In the case of more complex applications that require multiple applications, e.g., web servers, databases, and worker containers, it might be better to have a *set of containers* using `docker-compose` [14]. A `docker-compose.yml` configuration file allows for defining volumes, environment variables, and exposed ports and helps to stick to *one purpose per container*, which often means one process running in the container, and to combine existing stable building blocks instead of bespoke massive containers for specific purposes. Using `docker-compose` for a data science workflow is out of scope for this article.

Because *"the number of unique research environments approximates the number of researchers"* [15], sticking to conventions helps every researcher to understand, modify, and eventually write container recipes suitable for their needs. Even if they are not sure how the technology behind them actually works, reserachers can leverage containerisation, but should craft their own definition of computing environments

following good practices, which are provided here. These practices are strongly related to software engineering in general and research software engineering (RSEng) in particlar, which is concerned with quality, training, and recognition of software in science [16]. Research Software Engineers (RSEs) are not the target audience for this work, but we want to encourage you to reach out to your local or national RSE community if your needs go beyond the rules of this work.

While there are many different container technologies, this article focuses on Docker [17]. Docker is a highly suitable tool for reproducible research (e.g., [18]) and our observations indicate it is the most widely used container technology in academic data science. The goal of this article is to guide you to write a `Dockerfile`, which is a file format for creating container images. The rules help you to ensure the `Dockerfile` allows for interactive development as well as the higher goals of reproducibility and preservation of knowledge. Such practices are generally not part of generic containerisation tutorials and are rarely found in `Dockerfile`s published as part of software projects, which are often used as templates by novices. The differences between a helpful, stable `Dockerfile` and one that is misleading, prone to failure, and full of potential obstacles, are not obvious, especially for researchers who do not have extensive software development experience or formal training. A commitment to this article's rules can ensure that workflows are reproducible and reusable, that computing environments are understandable by others, and researchers can collaborate effectively. Their application should not be triggered by the publication of a finished project, but be weaved into day-to-day habits (cf. thoughts on openness as an afterthought by [19] and on computational reproducibility by [2]).

Docker & Dockerfiles

Docker [17] is a container technology that is widely adopted and supported on many platforms, and has become highly useful for research. Containers are distinct from virtual machines (VM) or hypervisors as they do not emulate hardware or operating system kernels, and thus do not require the same system resources. Several platforms for facilitating reproducible research are built on top of containers [15,20–23], but they intentionally hide most of the complexity from the researcher.

To create Docker containers for specific workflows, we write text files that follow a particular format called `Dockerfile` [24]. `Dockerfile`s are machine- and human-readable recipes for building images. Images are inert, immutable, read-only files that include the application, e.g., the programming language interpreter needed to run a workflow, and the system libraries required by an application to run. A `Dockerfile` consists of a sequence of instructions to copy files and install software. Each instructions adds a layers to the image, which can be cached across image builds and platforms for minimizing build and download times. The images have a main executable that is started when they are run as stateful containers, which are the running instances of Docker images. Containers can be modified, stopped, restarted and purged. `Dockerfiles`, like Makefiles [25], are meant to be useful for both humans to read and for computers to execute. See Listing 1 for a full `Dockerfile`, which we will refer to throughout this article.

While Docker was the original technology to support the `Dockerfile` format, other container technologies with support include podman/buildah supported by RedHat, kaniko, img, and buildkit. The Singularity container software [26] is optimised for high performance computing and although it uses its own format, the *Singularity recipe*, it can import and run Docker images. Although the Singularity recipe format is different, the rules here are transferable to some extent. While some may argue for reasons to not publish reproducibly, e.g., lack of time and incentives, reluctance to share (cf. [27]) and

there are substantial technical challenges to maintain software and documentation, providing a `Dockerfile`, a pre-built Docker image, or other type of container should become an increasingly easier task for the average researcher. If a researcher is able to find and create containers or write a `Dockerfile` to address their most common use cases, then arguably it will not be extra work after this initial set up (cf. README of [28]). In fact, the `Dockerfile` itself can be a powerful documentation to show from where data and code was derived, i.e. downloaded or installed, and consequently where a third party might obtain them again.

```
FROM rocker/verse

# install Java, needed for X

# install Y, needed for Z
RUN apt-get update \
  && apt-get install
    # needed for RNetCDF:
    netcdf udunits-2 \
    # needed for ...:
    more \
    arguments

# Taken from https://github.com/rocker-org/geospatial/blob/master/Dockerfile
RUN install2.r --error \
    RColorBrewer \
    RandomFields \
    RNetCDF

WORKDIR /work

COPY myscript.sh

# ???
RUN snakemake --use-conda <other params>
RUN nextflow workflow.nf --in 'dataset/*.fa'
RUN java -jar cromwell-XY.jar run myWorkflow.wdl

# Create the plots
RUN Rscript plots.R

# CMD from base image used for development, uncomment the following line to have a
# "run workflow only" container
# CMD["./myscript.sh"]

### Usage instructions ###
# Build the images with
# > docker build --tag great_workflow:1.0.0 .
# Run the image:
# > docker run --it --port 80:8787 --volume ./input:/input \
#     --name gwf great_workflow
# Extract the data:
# > docker cp gwf:/output/ ./output
```

Listing 1: `Dockerfile` full example.

1. Consider tools to assist with Dockerfile generation

Rule 1 could informally be described as "Don't bother to write a Dockerfile!". Writing a `Dockerfile` from scratch can be difficult and even experts sometimes take shortcuts. Thus, it is a good strategy to first look to tools that can help to generate a `Dockerfile` for you. Such tools have likely thought about and implemented good practices, and they

may have added newer practices when reapplied at a later point in time. Therefore the most important rule is to apply a multi-step process for your specific use case.

First, you want to determine if there is an already existing container that you can use, and in this case, use it and add to your workflow documentation instructions for doing so. As an example, you might be doing some kind of interactive development. For interactive development environments such as notebooks and development servers or databases, you can readily find containers that come installed with all the software that you need. You look for information about images (a) in the documentation of the used software, (b) the Docker image registry *Docker Hub*, `https://hub.docker.com/`, or (c) in the source code projects of the used software, as many developers today rely on containers for development, testing, and teaching.

Second, in the case that there isn't an existing container for your needs, you might next look to well-maintained tools to help with `Dockerfile` generation. These tools can add required software packages to an existing image without you having to manually write a `Dockerfile` at all. "Well-maintained" includes the tool's own stability and usability, but also that suitable base images are used, likely from the official Docker library [29], to ensure that the container has the most recent security fixes for the operating system in question. See below box "Tools for container generation" for details.

Third, you may want to write you own `Dockerfile` if these tools do not suffice for your needs. *In this case, follow the remaining rules.*

Tools for container generation

Repo2docker [23] is a tool that is maintained by Project Jupyter that can help to transform a source code or data repository, e.g. GitHub, GitLab, or Zenodo, into a container. The tool relies on common configuration files for defining software dependencies and versions, and supports a few more special files for, see the supported configuration files. As an example, we might install `jupyter-repo2docker` and then run it against a repository with a `requirements.txt` file, an indication of being a Python workflow with dependencies on the Python Package Index (PyPI) with the following command.

```
jupyter-repo2docker https://github.com/norvig/pytudes
```

The resulting container image installs the dependencies listed in the requirements file, along with providing an entrypoint to run a notebook server to interact with any existing workflows in the repository. Since repo2docker is used within MyBinder.org, if you make sure your workflow is "Binder-ready", you and others can also get an online workspace with a single click.‘ A precaution that needs to be taken is that the default command above will create a home for the current user, meaning that the container itself would not be ideal to share, but rather any researchers interested in interaction with the code inside should run repo2docker themselves and create their own container. Because repo2docker is deterministic, the environments are the same (see Rule 6 for ensuring the same software versions).

Additional tools to assist with writing `Dockerfile`s include `containerit` [30] and `dockta` [31]. `containerit` automates the generation of standalone `Dockerfile`s for workflows in R. It can provide a starting point for users unfamiliar with writing `Dockerfile`s, or together with other R packages provide a full image creation and execution process without having to leave an R session. `dockta` supports multiple programming languages and configurations files, just as `repo2docker`, but attempts to create readable `Dockerfile`s compatible with plain Docker and to improve user experience by cleverly adjusting instructions to reduce build time.

2. Use versioned images

A good understanding of how base images and image tags work is crucial, as the image and tag that you choose has important implications for your container. It's good practice to use base images that are maintained by the Docker library, so called *"official images"* [32], which benefit from a review for best practices and vulnerability scanning [13]. You can identify these images by the missing the user portion of the image name, which comes before the `/`, e.g., `r-base` or `python`. However, these images only provide basic programming languages or very widely used software. So you will likely use images maintained by organisations or fellow researchers. While some organisations can be trusted to update containers with security fixes (see list below), for most individual accounts that provide ready to use images, it is likely that these will not be updated regularly. It is even possible that images or `Dockerfile`s may disappear, or that images are published with malicious intent, though we have not heard of any such case in academia. Therefore, for security, transparency, and reproducibility, you should only use images where you have access to the `Dockerfile`. Do save a copy of the `Dockerfile`, e.g., by cloning the public repository where the `Dockerfile`s are maintained.

Images have *tags* associated with them with specific meanings, e.g., a version indicator such as `1.2` or `dev`, or variants such as `slim` attempting to reduce image size. Tags are defined at image built time and appear in image name after the `:` when you use an image, e.g., `python:1.2`. By *convention* a missing tag is assumed to be the word `latest`, which gives you the latest updates but also a moving target for your computing environment that can break your workflow. Note that a version tag means that the tagged software is frozen, but it does not mean the image will not change, as backwards compatible fixes (cf. semantic versioning, [33]), e.g., version `1.2.3` fixing a security problem in version `1.2.2` or updates to an underlying system library, will still be included.

For data science workflows, you should always rely on version-specific image tags both for base images that you use, and for images that you build yourself and then run. Also do not worry about image size, because the size will be often be much smaller than the data anyway. It is good practice to publish an image in an image registry, and we refer you to the documentation on automated builds for details, see Docker Hub Builds or GitLab's Container Registry. Do not `docker push` a locally built image, because that counteracts the considerations outlined above. If a pre-built image is provided in a public image registry, do not forget to direct the user to it in your documentation, e.g. in the `README` file or in an article.

The following list a selection of communities that produce widely used regularly updated images, including ready-to-use images with preinstalled software stacks. Do take advantage of images with complex software environments pre-installed, e.g., machine learning tool stacks, specific BLAS library.

- Rocker for R and RStudio images [18]
- Bioconductor Docker images for bioinformatics with R
- NeuroDebian images for neuroscience [34]
- Jupyter Docker Stacks for Notebook-based computing
- Taverna Server for running Taverna workflows

For example, here is how we would use a base image `verse`, which provides the populary Tidyverse suite of packages [35], with R version `3.5.2` from the `rocker` organisation on Docker Hub (`docker.io`, which is the default and can be omitted).

```
FROM docker.io/rocker/r-ver:3.5.2
```

3. Fomat intentionally and favour clarity

First, it is good practice to think of the `Dockerfile` as a human *and* machine readable file. This means that you should use indentation, new lines, and comments to make your `Dockerfile`s well documented and readable. Specifically, carefully indent commands and their arguments to make clear what belongs together, especially when connecting multiple commands in a `RUN` instruction with `&&`. Use \ at the end of a line to break a single command into multiple lines. This will ensure that no single line gets too long to comfortably read. Use long versions of parameters for readability (e.g., `--input` instead of `-i`). When you need to change a directory, use `WORKDIR`, because it not only creates the directory if it doesn't exist but also persist the change across multiple `RUN` instructions.

Second, clarity is nearly always more important than brevity. For example, if your container uses a script to run a complex install routine, instead of removing it from the container upon completion, which is commonly seen in production `Dockerfile`s aiming at small image size, you should keep the script in the container for a future user to inspect. A common pattern you might encounter is a single `RUN` instruction that updates the database of available packages, installs a software from a package repository, and then purges the cache of the package manager. For Depending on the programming language used, your project may already contain files to manage dependencies and you may use a package manager to control this aspect of the computing environment. This is a very good practice and helpful, though you should consider the externalisation of content to outside of the `Dockerfile` (see Rule 7). A single long `Dockerfile` with sections and helpful comments can be complete and thus more understandable than a collection of separate files.

Generally, aim to design the `RUN` instructions so that each performs one scoped action, e.g., download, compile, and install *one tool*. Each instruction will result in a new layer, and reasonably grouped changes increase readability of the `Dockerfile` and facilitate inspection of the image, e.g., with tools like dive [36]. Convoluted `RUN` instructions can be acceptable to reduce the number of layers, but careful layout and consistent formatting should be applied (see Rule 3). A `RUN` instruction not fully visible on your screen requires scrolling and thereby diminishes readability. This may be challenging for the next reader to digest and you should consider splitting it up, accepting the extra layers added to the image.

While you will find `Dockerfile`s using *build-time variables* to dynamically change parameters at build time, such a customisation option reduces clarity for data science workflows based on research compendia.

4. Document within the Dockerfile

Comments

As you are writing the `Dockerfile`, be mindful of how other people will read it and why. Are your choices and commands being executed clear, or is further comment warranted? To assist others in making sense of your `Dockerfile`, you can add comments that include links to online forums, code repository issues, or VCS commit messages to give context for your specific decisions. For example this `Dockerfile` by Kaggle does a good job at explainig the reasoning and steps of the contained instructions. Comments should include helpful usage guides and links for readers inspecting the `Dockerfile`, including future you. For example, if you copy instructions from another `Dockerfile`, acknowledge the source in a comment. It can even be helpful to include comments about commands that did not work so you do not repeat past

mistakes. If you find that you need to remember an undocumented step, that is an indication that it should be documented in the `Dockerfile`. All instructions can be grouped starting with a short comment, which also makes it easier to spot changes if your `Dockerfile` is managed in a VCS (see Rule 9).

Here is a selection of typical kinds of comments that are useful to include in a `Dockerfile`:

```
# apt-get install specific version, use 'apt-cache madison <pkg>'
# to see available versions
RUN apt-get install python3-pandas=0.23.3+dfsg-4ubuntu1

# RUN command spreading several lines
RUN R -e 'getOption("repos")' && \
  install2.r \
    fortunes \
    here

# this library must be installed from source to get version newer
# than in sources
RUN git clone http://url.of/repo && \
  cd repo && \
  make build && \
  make install

# following commands from instructions at <LINK HERE>
```

Labels

Docker captures useful information in the image metadata automatically, such as the version of Docker used for building the image. The `LABEL` instruction can add *custom metadata* to images. You can view all labels with `docker inspect` command. Labels serve as structured metadata that can be leveraged by services, e.g., https://microbadger.com/labels. For example, software versions, license, and maintainer contact information are commonly seen and very useful if a `Dockerfile` is discovered out of context. While you can add arbitrarily complex information with labels, for research compendia the user-facing documentation is much more important. If you want to earn extra points, and you never know what future algorithms will be able to make sense of, include global identifiers such as ORCID identifiers for people, a DOI of the research compendium, e.g., reserved on Zenodo before publishing the research compendium, or your funding agency's grant number.

The OCI Image Format Specification provides some common label keys (see the "Annotations" section in [37]) to help standardise field names across container tools, as shown below. These labels match the `org.label-schema`-specification, which has been deprecated but is still found a lot in `Dockerfile`s.

```
"" LABEL org.opencontainers.image.created='2019-12-10'
org.opencontainers.image.authors='Nüst, Sochat, Marwick, Eglen, Head, and Hirst'
org.opencontainers.image.url='https://github.com/nuest/ten-simple-rules'
org.opencontainers.image.documentation='https://nuest.github.io/ten-simple-rules-dockerfiles/ten-simple-rules-dockerfiles.pdf'
org.opencontainers.image.version='0.0.1'
```

build-time variable with a default, set with –build-arg BUILD_DATE=$(date -u +"%Y-%m-%dT%H:%M:%SZ")

```
ARG BUILD_DATE unspecified
    LABEL maintainer="daniel.nuest@uni-muenster.de"
    org.label-schema.vendor='Ten Simple Corp.'
    org.label-schema.description='A great image is important (published under MIT license)'
    org.label-schema.build-date=$BUILD_DATE
    org.label-schema.docker.schema-version="rc1" "'
```

Usage instructions

It is often helpful to provide usage instructions, i.e., how to `docker build` and `docker run` the image, *within* the `Dockerfile`. Such documentation is especially relevant if volume mounts or ports are important for using the container, see for example the final lines of Listing 1. By putting the usage documentation at the end of the file, they are easy to copy-paste after a container build when you are likely to have the `Dockerfile` open, and, most importantly, they can be kept in consistent state compared to documentation in another file. This helps both you and others. It helps you when you pick up a previously written `Dockerfile` and for example do not remember the container identifier you used, it would be represented in the `Dockerfile` in the `--name` parameter, preserved in a comment. Following a common coding aphorism, we might say *"A Dockerfile written three months ago may just as well have been written by someone else"*. It helps others, because it quickly gets them running your workflow and interacting with the container in the intended way without reading all of the instructions (a "tl;dr"-kind of usage). Your documentation, e.g. `README` file, should not duplicate this information alongside the `Dockerfile` for exactly how to build, run, and otherwise interact with the container but instead point to the `Dockerfile`. Duplicate usage instructions with eventually diverge and lead to frustration, and referring to the `Dockerfile` has the added advantage that you point your reader to it, demonstrating your careful work habits and good intentions for transparency and computational reproducibility.

5. Order instructions

You will regularly build an image more during development of your workflow. You can take advantage of *build caching* to avoid execution of time-consuming instructions, e.g., install from a remote resource or a copying a file that gets cached. Therefore you should add instructions *in order* of least likely to change to most likely to change. Docker will execute the instructions in the order as they appear in the `Dockerfile`. When one instruction is completed, the result is cached, and the build moves to the next one. If you change something in the Dockerfile, and rebuild the container, each instruction is inspected in turn. If it hasn't changed, the cached layer is used and the build progresses. If the line has changed, that build step is executed afresh and then every following instruction will have to be executed in case the changed line influences a later instruction. You should regularly re-build the image using the `--no-cache` option to learn about broken instructions as soon as possible (cf. Rule 10).

A recommended ordering based on these considerations is as follows.

1. System libraries
2. Language-specific libraries or modules
3. from repositories (i.e., binaries)

4. from source (e.g., GitHub)
5. Installation of own software and scripts (if not mounted)
6. Copying data and configuration (if not mounted)
7. Labels
8. Default commands and entrypoints

6. Define version numbers for reproducible builds

The reproducibility of your `Dockerfile` heavily depends on how well you define the versions of software to be installed in the image. The more specific, the better, because using the desired version leads to reproducible builds. The practice of specifying versions of software is called *version pinning* (e.g., on `apt`: https://blog.backslasher.net/my-pinning-guidelines.html). For stable workflows in a scientific context, it is generally advised to freeze the computing environment explicitly and not rely on the "current" or "latest" software, which is a moving target.

System libraries

System library versions can largely come from the base image tag that you choose to use, e.g., `ubuntu:18.04`, because the operating system's software repositories are very unlikely to introduce breaking changes, but predominantly fix errors with newer versions. However, you can also install specific versions of system packages with the respective package manager. For example, you might want to demonstrate a bug, prevent a bug in an updated version, or pin a working version if you suspect an update could lead to a problem. Generally, system libraries are more stable than software modules supporting analysis scripts, but in some cases they can be highly relevant to your workflow. *Installing from source* is a useful way to install very specific versions, however it comes at the cost of needing to build libraries. Here are some examples of terminal commands that will list the currently installed versions of software on your system:

- Debian/Ubuntu: `dpkg --list`
- Alpine: `apk -vv info|sort`
- CentOS: `yum list installed` or `rpm -qa`

When you install several system libraries, it is good practice to add comments about why the dependencies are needed (see Listing 1). This way, if a piece of software is removed from the container, it will be easier to remove the system dependencies that are no longer needed, thereby reducing maintenance overhead: you will not unnecessarily fix problems with a library that's not needed anymore or include long-running installations.

Extension packages and programming language modules

In the case of needing to install packages or dependencies for a specific language, package managers are a good option. Package managers generally provide reliable mirrors or endpoints to download software, many packages are tested before release, and most importantly the provide access to specific versions. Most package managers have a command line interface that can be used from `RUN` commands in your `Dockerfile`, along with various flavors of "freeze" commands that can output a text file listing all software packages and versions
(cf. https://markwoodbridge.com/2017/03/05/jupyter-reproducible-science.html cited by [5]) The biggest risk with using package managers with respect to `Dockerfile`s is outsourcing configuration. As an example, here are configuration files supported by commonly used languages in scientific programming:

- Python: `requirements.txt` (pip tool, [38]), `environment.yml` (Conda, [39])
- R: `DESCRIPTION` file format [40] and `r` ("little R", [41])
- JavaScript: `package.json` of `npm` [42]
- Julia: `Project.toml` and `Manifest.toml` [43]

In some cases (e.g., Conda) the package manager is also able to make decisions about what versions to install, which is likely to lead to a non-reproducible build. In all of the above, the user is required to inspect the file or the build to see what is installed. For this reason, in the case of having few packages, it is suggested to write the install steps and versions directly into the `Dockerfile` (also for clarity, see Rule 3). For example, the `RUN` instruction here:

```
RUN pip install geopy==1.20.0 && \
    pip install uszipcode==0.2.2
```

serves as more clear documentation in a `Dockerfile` than a `requirements.txt` file that lists the same:

```
RUN pip install -r requirements.txt
```

This modularisation is a potential risk for understandability and consistency, which can be mitigated by carefully managing all these files in the same version-controlled project. You can also use package manager to install software from source code `COPY`ied into the image (see Rule 7). And finally, you can use many package managers to install software from source obtained from code management repositories, e.g., installing a specific tool identified by a GitHub version tag or commit hash. Be aware of the risk of depending on such repositories, especially if they are out of your control, although the installation command with a full URL should allow readers of your `Dockerfile` to dig deeper if problems arise. Such an installation from source gives you a lot of freedom, including the one to shoot yourself in the foot. The version pinning capabilities of these file formats and package managers are described in their respective documentation.

As a final note on software installation, you should be aware of the `USER` instruction in a `Dockerfile` and how your base image might adjust the user to enable particular usages. It is common to use images with the default user `root`, e.g., , `root` is required for installing system dependencies. We recommend to make sure the image works without specifying a users, and to document precisely if your image deviates from that. One reason you will encounter base images running a non-root user (popular examples are the Jupyter and Rocker image stacks) is to avoid permission problems when mounting files into the container, especially for "output" files (see Rule 7).

7. Mount user scripts and data

The role of containers is to provide the computing environment, not to encapsulate datasets or control scripts. It is better to insert data files and scripts files from the local machine into the container at run time, and using the container image primarily for the software and dependencies. This insertion is achieved by using *bind mounts*. Mounting these files is preferable to using the `ADD`/`COPY` instructions in the `Dockerfile`, because files persist when the container instance or image is removed from your system, and the files are more accessible when the workspace is published as a research compendium. If you want to add local files to the container, prefer `COPY` because it is explicit and you do not need `ADD`'s extra features, which are (i) URL as a source, for which you may use it instead of a `RUN wget` or `RUN curl` to download files, and (ii) a local tar file as a source,

which you should avoid because such an archive file will be harder for future readers of your `Dockerfile` to make sense of and cannot be properly version controlled. Unless specific features are needed, bind mounts are also preferable to storage volumes.

Standalone *script files* are distinct from other software as they are treated as files and managed software, i.e. installed with a package manager and versioned. If you developed extensive software for a specific analysis, you should ideally publish it as a software package and follow Rule 6 for installing it. Consider using the suitable package system with pinned versions even for small scripts if the functions are reusable across datasets or workflows by you or others. You should avoid installing software packages from source after `COPY`ing the code into the image, because the connection between the file outside of the image and the one copied in is easily lost (cf. Rule 7). If you cannot publish a software project albeit it being in a language's package structure, you should make sure the files are always published together with the `Dockerfile` (see Rule 9).

Storing *data files* outside of the container allows handling of very large datasets and datasets with data worthy of protection, e.g., proprietary data or private information. Do not include such data in an image. To avoid publishing sensitive data by accident, you can add the data directory to the `.dockerignore` file, which excludes files and directories from the build context, i.e., the set of files considered by `docker build`. Ignoring data files also speeds up the build in case of very large files or many small files, the latter of which makes it common to ignore for example the `.git` folder of the Git VCS. You should include dummy or test data into the image to be able to ensure that a container is functional without a larger custom dataset, e.g., for automated tests or instructions in the user manual. For all these cases you should provide clear instructions for users in the `README` how to obtain actual or dummy data. When publishing your workspace, e.g., on Zenodo, having data and script contents as regular files outside of the container makes them more accessible to others, for example for reuse or analysis.

A mount can can also be used to access *output data* from a container. This can be an extra mount, or the same `data` directory. Alternatively, you can use the `docker cp` command to access files from a running or stopped container, but this requires a specific handling, e.g., naming the container when starting it or using multiple shells, that require very detailed instructions for users.

You can use the `-v`/`--volume` or `--mount` flags to `docker run` to configure bind mounts of directories or files [44], including options, as shown in the following examples. If the target path exists within the image, the bind mount will replace it for the started container.

```
# mount directory
docker run --volume /home/user/project:/project mycontainer

# mount directory as read-only
docker run --volume /home/user/project:/project:ro mycontainer

# mount multple directories, one with write access relative to current path(Li
docker run --volume /homse/user/Download/article-x-supplement/data:/data:ro--v
```

How your container expects external resources to be mounted into the container should be included in the example commands (see Rule 3). In these commands you can also make sure to avoid issues with file permissions by using Docker's `--user` option. For example, by default, writing a new file from inside the container will be owned by user `root` on your host, because that is the default user within the container.

8. Enable interactive usage and one-click execution

Containers are very well suited for day-to-day development tasks (see also Rule 10), because they support common interactive environments for data science and software development. But they are also useful for a "headless" execution of a full workflows. For example, [45] demonstrates a container for running an agent-based model with video files as outputs, or this article's R Markdown source, which could include cells with analysis code, is rendered into a PDF in a container. A workflow that does not support headless execution may even be seen as irreproducible.

These two usages can be configured by the `Dockerfile`'s author and exposed to the user based on the `Dockerfile`'s `CMD` and `ENTRYPOINT` instructions. An images main purpose is reflected by the default process and configuration, though the `CMD` and `ENTRYPOINT` can also be changed at runtime. It is considered good practice to have a combination of default command and entrypoint that meets reasonable user expectations. For example, a container known to be a workflow should execute the workflow, or provide instructions for how to do so. Since you are likely the primary user of the `Dockerfile` and image, so you should choose a selection or combination that works for you, only then accomodate other user's needs. A possible weakness with using containers is the limitation on only providing one default command and entrypoint. However tools, e.g., The Scientific Filesystem [46], have been developed to expose multiple entrypoints, environments, help messages, labels, and even install sequences. With plain Docker, you can override the defaults as part of the `docker run` command or in an extra `Dockerfile` using the primary image as a base, as shown in Listing 2. In any case you should document different variants ver well, potentially capture build and run commans a `Makefile` [25]. To support advanced custom configuration, it's helpful to expose settings via a configuration file, which can be bind mounted from the host [45], via environment variables [47], or via wrappers using Docker, such as Kliko [48].

```
# base image (interactive)
FROM jupyter/TODO

# Usage instructions:
# $ docker build --tag workflow:1.0 .
# $ docker run workflow:1.0
```

```
# interactive image
FROM workflow:1.0

ENTRYPOINT ["python"]
CMD ["/workspace/run-all.sh"]

# Usage instructions:
# $ docker build --tag workflow-runner:1.0 --file Dockerfile.runner .
# $ docker run -e ITERATIONS=10 -e ALGORITHM=advanced --volume /tmp/results:/workspace/output_data
```

Listing 2: Workflow `Dockerfile` and derived "runner image" `Dockerfile` with filename `Dockerfile.runner`

Interactive graphical interfaces, such as RStudio, Jupyter, or Visual Studio Code, can run in a container to be used across platforms via a regular web browser. The HTML-based user interface is exposed over HTTP. Use the `EXPOSE` instruction to document the ports of interest for both humans and tools, because they need to be bound to the host to be accessible to the user using the `docker run` option `-p/--publish <host port>:<container port>`. The container should also print to the screen the used ports along with any login credentials needed. For example, as done in the last few lines of the output of running a Jupyter Notebook server locally (lines abbreviated).

```
docker run -p 8888:8888 jupyter/datascience-notebook:7a0c7325e470
```

```
[...]
[I 15:44:31.323 NotebookApp] The Jupyter Notebook is running at:
[I 15:44:31.323 NotebookApp] http://9027563c6465:8888/?token=6a92d [..]
[I 15:44:31.323 NotebookApp]  or http://127.0.0.1:8888/?token=6a92 [..]
[I 15:44:31.323 NotebookApp] Use Control-C to stop this server and [..]
```

A person who is unfamiliar with Docker but wants to use your image may rely on graphical tools like Kitematic [49] or ContainDS for assistance in managing containers on their machine without using the Docker CLI.

Interactive usage of a command-line interfaces is quite straightforward to access from containers, if users are familiar with this style of user interface. Running the container will provide a shell where a tool can be used and help or error messages can assist the user. For example, complex workflows in any programming language can, with suitable pre-configuration, be triggered by running a specific script file. If your workflow can be executed via a CLI you may use that to validate correct functionality of an image in automated builds, e.g. using a small toy example and checking the output, by checking successful responses from HTTP endpoints provided by the container, e.g. via an HTTP response code of `200`, or by using a controller such as Selenium [50].

The followig example runs a simple R command counting the lines in this articles source file. The file path is passed as an environment variable.

```
docker run \
  --env CONFIG_PARAM="/data/ten-simple-rules-dockerfiles.Rmd" \
  --volume $(pwd):/data \
  jupyter/datascience-notebook:7a0c7325e470 \
  R --quiet -e "
l = length(readLines(Sys.getenv('CONFIG_PARAM')));
print(paste('Number of lines: ', l))
"
```

```
> l = length(readLines(Sys.getenv('CONFIG_PARAM')));
> print(paste('Number of lines: ', l))
[1] "Number of lines:  568"
```

If there is only a regular desktop application, the hosts window manager can be connected to the container. This has notable security implications, which are reduced by using the "X11 forwarding" natively supported by Singularity [51], which can execute Docker containers, or by leveraging supporting tools such as `x11docker` [52]. Bidge containers [53] and exposing a regular desktop via the browser (e.g., for Jupyter Hub [54]) are further alternatives. This variety of approaches render seemingly more convenient uncontainerised environments, i.e. just using the local machine, unneccessary in favour of reproducibility and portability.

9. Use a Dockerfile per project and publish it with a version control system

Because a `Dockerfile` is a plain text-based format, it works well with VCS. Including a `Dockerfile` alongside your code and data in a research compendium is an effective way to consistently build your software, to show visitors to the repository how it is built and

used, to solicit feedback and collaborate with your peers, and to increase the impact and sustainability of your work (cf. [55]). Online collaboration platforms (e.g., GitHub, GitLab) also make it easy to use CI services to test building and executing your image in an independent environment. Continuous integration increases stability and trust, and gives the ability to publish images automatically. If your `Dockerfile` includes an interactive user interface, you can also adapt it so that it is ready-to-use as a Binder instance [23], providing an online work environment to any user with a simple click of a link. Furthermore, the commit messages in your version controlled repository preserve a record of all changes to the `Dockerfile`.

While there are exceptions to the rule (cf. [56]), it's generally a simple and clear approach to provide one `Dockerfile` per project. Alternatively, you could build an *image stack* to serve different use cases with a common base image, but this reduces understandability and scatters information across multiple places. We recommend to avoid this at the price of a longer `Dockerfile` and mitigate with consistent form (see Rule 3). Yet most importantly, you should publish *all* files `COPY`ied into the container, e.g., test data or files for software installation from source (see Rule 7), in the same research compendium.

It is likely going to be the case that over time you will develop workflows that are similar in nature to one another. When developing or working on projects with containers you can switch between isolated project environments by stopping the container and restarting it when you are ready to work again, even on another machine or in a cloud environment. You can even run projects in parallel without interference. In case you feel like constantly repeating yourself, you should consider adopting a standard workflow that will give you a clean slate for a new project. As an example, cookie cutter templates [57], project starter kits [REF], or community templates (e.g., [58]) can provide required files, e.g., for documention, CI, and licenses, and structure for getting started. If you decide to build your own best practice, consider to collaborate with your community during development of the standard to ensure it will be useful to others. Part of your project template should be a protocol for publishing the `Dockerfile` and even an export of the image in a suitable location, e.g., a container registry or data repository, taking into consideration of how your workflow can receive a DOI for citation.

10. Use the container daily, rebuild the image weekly, clean up and preserve if need be

Using containers for research workflows does not only require technical understanding, but also an awareness of risks that can be managed efficiently by following a number of good *habits*, which we outline below. While there is no firm rule, if you use a container daily, is good practice to rebuild that container every once or two weeks. At the time of publication of research results it is good practice to save a copy of the image in a public data repository so that readers of the publication can access the resources that produced the published results.

First, use your container every time you work on a project and not just as a final step during publication. If the container is the only platform you use, the confidence in proper documentation of the computing environment can be very high [59]. You should prioritise this usage over others, e.g., non-interactive execution of a full workflow, because it gives you personally the highest value and does not limit your use or others' use of your data and code at all (see Rule 8).

Second, for reproducibility, we can treat containers as transient and disposable, and even intentionally rebuild an image at regular intervals. Ideally, containers that we built

years ago should rebuild seamlessly, but this is not necessarily the case, especially with rapidly changing technology relevant to machine learning and data science. It can almost be guaranteed future changes will interfere with the image build process or even the functionality of the software within the image. The longer that you wait to rebuild the image, the more change happens, the higher the risk that your build will not work, and the harder it becomes to identify and fix the problem. If you are using an interactive container and find that you need to manually install a package or change a parameter, it is best practice to add this dependency to the container and rebuild it right away, but one tends to take shortcuts. Therefore, a habitual deletion of a container and cache-less rebuild of the image (a) increases security due to updating underlying software, (b) helps to reveals issues requiring manual interference, i.e., changes to code or configuration not documented in the `Dockerfile` but perhaps should be, and (c) allows you to more incrementally debug issues. This habit can be supported by using continuous deployment or CI strategies. If a VCS repository with a `Dockerfile` is linked to an automated build, then a CI pipeline can build images and execute a validating run the new image, e.g., using contained demo data (see Rule 7).

In the case of needing setup or configuration for the first two habits, it is good practice to provide a `Makefile` alongside your container, which can capture the specific commands. The effective use of a `Makefile` can help avoiding undocumented steps or manual, extra commands to be run on the local machine. A fully scripted configuration makes it easier for both you and future users, and can increase trust in your workflow. If you use a `Makefile`, consider pointing to it instead of adding usage instructions in a comment at the end of the `Dockerfile`. A `Makefile` will also help you to take advantage of build-time variables in a useful way (unlike Rule **??**) for inserting specific metadata into an image, e.g., the current VCS commit hash or a timestamp.

Third, from time to time you can reduce the system resources occupied by Docker images and their layers or unused containers, volumes and networks by running `docker system prune --all`. After a prune is performed, it follows naturally to rebuild a container for local usage, or to pull it again from a newly built registry image. This habit can be automated with a cron job [60].

Fourth, you can export the image to file and deposit it in a public data repository, where it not only becomes citable but also provides a snapshot of the *actual* environment you used at a specific point in time. You should include instructions how to import and run the workflow based on the image archive and add your own image tags for clarity. Depositing the image next to other project files, i.e., data, code, and the used `Dockerfile`, in a public repository makes them likely to be preserved, but is is highly unlikely that over time you will be able to recreate it precisely from the accompanying `Dockerfile`. Publishing the image and the contained metadata therein (e.g., the Docker version used) may even allow future science historians to emulate the Docker runtime environment. Applying proper preservation strategies (cf. [55]) can be highly complex, but simply running an image "as-is", i.e. with the default command and entrypoint (see Rule 8), and observing the output is quite likely to work for many years into the future. If the image does not work anymore, a user can still extract the image contents and explore the files of each layer manually, or if an import still works, with exploration tools like dive [36]. However, if you want to ensure usability and extendability, then you could run import, run, and export an image regularly to make sure the export format still works with the then current version of Docker.

The exported image and a version controlled `Dockerfile` together allow you to freely experiment and continue development of your workflow and keeping the image up to date, e.g., updating versions of pinned dependencies (see Rule 6) and regular image building (see above).

Finally, for a sanity check and to foster even higher trust in the stability and

documentation of your project, you can ask a colleague or community member to be your code copilot (see `https://twitter.com/Code_Copilot`) to interact with your workflow container on a machine of their own. You can do this shortly before submitting your reproducible workflow for peer-review, so you are well positioned for the future of scholarly communication and open science where these may be standard practices required for publication [19,61–63].

Example Dockerfiles

To demonstrate the ten rules, we maintain a collection of annotated example `Dockerfile`s in a GitHub repository, some of which we took from public repositories and updated to adhere better to the rules (see `Dockerfile.original`): `https://github.com/nuest/ten-simple-rules-dockerfiles/`

Conclusion

Reproducibility in research is an endeavor of incremental improvement and best efforts, not about achieving the perfect solution, which may be not achievable for many researchers with limited resources, and the definition of which may change over time. Containerisation can support researchers. Containers can be optimised for very different use cases, such as supporting many hosts, being small in size for quick deployments in cloud infrastructures, but in this article we have provided guidance for using `Dockerfile`s to create containers for use and communication in computational research. Our goal is to help the researcher to work towards creating a "time capsule" [64] in form of a research compendium. Given some expertise and the right tools, such a compendium can come as close as needed to the original workflow with reasonably little effort for better science. Even if such a capsule decays over time, the effort to create and document it provides incredibly useful and valuable transparency for the project. We encourage researchers to value these steps taken by their peers to use `Dockerfile`s to create "time capsules", and promote change in the way scholars communicate towards an open and friendly "preproducibility" [65]. So please, make a best effort with your current knowledge, and strive to write readable `Dockerfile`s for functional containers, that are realistic about what might break and what is unlikely to break. In a similar vein, we accept that you should freely break these rules if another way makes more sense *for your use case*. Most importantly, share and exchange your `Dockerfile` freely and collaborate in your community to spread the knowledge about containers as a tool for research and scholarly collaboration and communication. Together we can develop common practices and shared materials for better transparency, higher efficiency, and faster innovation.

Acknowledgements

DN is supported by the project Opening Reproducible Research II (https://o2r.info/; https://www.uni-muenster.de/forschungaz/project/12343) funded by the German Research Foundation (DFG) under project number PE 1632/17-1. DN and SJE are supported by a Mozilla mini science grant.

Contributions

DN conceived the idea, outlined the first rules, and contributed to all rules. VS wrote the first draft and contributed to all rules. BM revised the text and contributed to all rules. SJE contributed to the overall structure and selected rules. THe contributed to the rule structure and particularly Rule 1. THi gave extensive feedback on early drafts and contributed to the discussion. This articles was written collaboratively on GitHub, where all contributions in form of text or discussions comments are documented: `https://github.com/nuest/ten-simple-rules-dockerfiles/`.

References

1. Marwick B. How computers broke science – and what we can do to fix it [Internet]. The Conversation. 2015. Available: `https://theconversation.com/how-computers-broke-science-and-what-we-can-do-to-fix-it-49938`

2. Donoho DL. An invitation to reproducible computational research. Biostatistics. 2010;11: 385–388. doi:10.1093/biostatistics/kxq028

3. Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, et al. Best Practices for Scientific Computing. PLOS Biology. 2014;12: e1001745. doi:10.1371/journal.pbio.1001745

4. Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. Good enough practices in scientific computing. PLOS Computational Biology. 2017;13: e1005510. doi:10.1371/journal.pcbi.1005510

5. Rule A, Birmingham A, Zuniga C, Altintas I, Huang S-C, Knight R, et al. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. PLOS Computational Biology. 2019;15: e1007007. doi:10.1371/journal.pcbi.1007007

6. Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten Simple Rules for Reproducible Computational Research. PLoS Comput Biol. 2013;9: e1003285. doi:10.1371/journal.pcbi.1003285

7. Katz DS, Chue Hong NP. Software Citation in Theory and Practice. In: Davenport JH, Kauers M, Labahn G, Urban J, editors. Mathematical Software – ICMS 2018. Springer International Publishing; 2018. pp. 289–296. doi:10.1007/978-3-319-96418-8_34

8. Nüst D. Author Carpentry : Docker for reproducible research [Internet]. Author Carpentry : Docker for reproducible research. 2017. Available: `https://nuest.github.io/docker-reproducible-research/`

9. Chapman P. Reproducible data science environments with Docker Phil Chapman's Blog [Internet]. 2018. Available: `https://chapmandu2.github.io/post/2018/05/26/reproducible-data-science-environments-with-docker/`

10. rOpenSci Labs. R Docker tutorial [Internet]. 2015. Available: `https://ropenscilabs.github.io/r-docker-tutorial/`

11. Udemy, Zhbanko V. Docker Containers for Data Science and Reproducible Research [Internet]. Udemy. 2019. Available: `https://www.udemy.com/course/docker-containers-data-science-reproducible-research/`

12. Psomopoulos FE. Lesson "Docker and Reproducibility" in Workshop "Reproducible analysis and Research Transparency" [Internet]. Reproducible analysis and Research Transparency. 2017. Available: `https://reproducible-analysis-workshop.readthedocs.io/en/latest/8.Intro-Docker.html`

13. Docker Inc. Best practices for writing Dockerfiles [Internet]. Docker Documentation. 2020. Available: `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`

14. Docker Inc. Overview of Docker Compose [Internet]. Docker Documentation. 2019. Available: https://docs.docker.com/compose/

15. Nüst D, Konkol M, Pebesma E, Kray C, Schutzeichel M, Przibytzin H, et al. Opening the Publication Process with Executable Research Compendia. D-Lib Magazine. 2017;23. doi:10.1045/january2017-nuest

16. Cohen J, Katz DS, Barker M, Chue Hong NP, Haines R, Jay C. The Four Pillars of Research Software Engineering. IEEE Software. 2020; doi:10.1109/MS.2020.2973362

17. Wikipedia contributors. Docker (software) [Internet]. Wikipedia. 2019. Available: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=928441083

18. Boettiger C, Eddelbuettel D. An Introduction to Rocker: Docker Containers for R. The R Journal. 2017;9: 527–536. doi:10.32614/RJ-2017-065

19. Chen X, Dallmeier-Tiessen S, Dasler R, Feger S, Fokianos P, Gonzalez JB, et al. Open is not enough. Nature Physics. 2019;15: 113. doi:10.1038/s41567-018-0342-2

20. Brinckman A, Chard K, Gaffney N, Hategan M, Jones MB, Kowalik K, et al. Computing environments for reproducibility: Capturing the "Whole Tale". Future Generation Computer Systems. 2018; doi:10.1016/j.future.2017.12.029

21. Code Ocean [Internet]. 2019. Available: https://codeocean.com/

22. Šimko T, Heinrich L, Hirvonsalo H, Kousidis D, Rodríguez D. REANA: A System for Reusable Research Data Analyses. EPJ Web of Conferences. 2019;214: 06034. doi:10.1051/epjconf/201921406034

23. Jupyter P, Bussonnier M, Forde J, Freeman J, Granger B, Head T, et al. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. Proceedings of the 17th Python in Science Conference. 2018; 113–120. doi:10.25080/Majora-4af1f417-011

24. Docker Inc. Dockerfile reference [Internet]. Docker Documentation. 2019. Available: https://docs.docker.com/engine/reference/builder/

25. Wikipedia contributors. Make (software) [Internet]. Wikipedia. 2019. Available: https://en.wikipedia.org/w/index.php?title=Make_(software)&oldid=929976465

26. Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLOS ONE. 2017;12: e0177459. doi:10.1371/journal.pone.0177459

27. Boettiger C. An Introduction to Docker for Reproducible Research. SIGOPS Oper Syst Rev. 2015;49: 71–79. doi:10.1145/2723872.2723882

28. Ben Marwick. 1989-excavation-report-Madjebebe. 2015; doi:10.6084/m9.figshare.1297059

29. Docker Inc. Official Images on Docker Hub [Internet]. Docker Documentation. 2019. Available: https://docs.docker.com/docker-hub/official_images/

30. Nüst D, Hinz M. Containerit: Generating Dockerfiles for reproducible research with R. Journal of Open Source Software. 2019;4: 1603. doi:10.21105/joss.01603

31. Stencila. Stencila/dockta [Internet]. Stencila; 2019. Available: https://github.com/stencila/dockta

32. Docker Inc. Official Images on Docker Hub [Internet]. Docker Documentation. 2020. Available: https://docs.docker.com/docker-hub/official_images/

33. Preston-Werner T. Semantic Versioning 2.0.0 [Internet]. Semantic Versioning. 2013. Available: https://semver.org/

34. Halchenko YO, Hanke M. Open is Not Enough. Let's Take the Next Step: An Integrated, Community-Driven Computing Platform for Neuroscience. Frontiers in Neuroinformatics. 2012;6. doi:10.3389/fninf.2012.00022

35. Wickham H, Averick M, Bryan J, Chang W, McGowan L, François R, et al. Welcome to the tidyverse. Journal of Open Source Software. The Open Journal; 2019;4: 1686. doi:10.21105/joss.01686

36. Goodman A. Wagoodman/dive [Internet]. 2019. Available: https://github.com/wagoodman/dive

37. Opencontainers. Opencontainers/image-spec v1.0.1 - Annotations [Internet]. GitHub. 2017. Available: https://github.com/opencontainers/image-spec/blob/v1.0.1/annotations.md

38. The Python Software Foundation. Requirements Files — pip User Guide [Internet]. 2019. Available: https://pip.pypa.io/en/stable/user_guide/#requirements-files

39. Continuum Analytics. Managing environments — conda documentation [Internet]. 2017. Available: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

40. R Core Team. The DESCRIPTION file in "writing r extensions" [Internet]. 1999. Available: https://cran.r-project.org/doc/manuals/r-release/R-exts.html#The-DESCRIPTION-file

41. Eddelbuettel D, Horner J. Littler: R at the command-line via 'r' [Internet]. 2019. Available: https://CRAN.R-project.org/package=littler

42. npm. Creating a package.json file npm Documentation [Internet]. 2019. Available: https://docs.npmjs.com/creating-a-package-json-file

43. The Julia Language Contributors. 10. Project.Toml and Manifest.Toml · Pkg.Jl [Internet]. 2019. Available: https://julialang.github.io/Pkg.jl/v1/toml-files/

44. Docker Inc. Use bind mounts [Internet]. Docker Documentation. 2019. Available: https://docs.docker.com/storage/bind-mounts/

45. Verstegen JA. JudithVerstegen/PLUC_Mozambique: First release of PLUC for Mozambique [Internet]. Zenodo; 2019. doi:10.5281/zenodo.3519987

46. Sochat V. The Scientific Filesystem. GigaScience. 2018;7. doi:10.1093/gigascience/giy023

47. Knoth C, Nüst D. Reproducibility and Practical Adoption of GEOBIA with Open-Source Software in Docker Containers. Remote Sensing. 2017;9: 290. doi:10.3390/rs9030290

48. Molenaar G, Makhathini S, Girard JN, Smirnov O. Kliko—The scientific compute container format. Astronomy and Computing. 2018;25: 1–9. doi:10.1016/j.ascom.2018.08.003

49. Docker Inc. Docker/kitematic [Internet]. Docker; 2019. Available: https://github.com/docker/kitematic

50. Selenium contributors. SeleniumHQ/selenium [Internet]. Selenium; 2019. Available: https://github.com/SeleniumHQ/selenium

51. Singularity. Frequently Asked Questions Singularity [Internet]. 2019. Available: http://singularity.lbl.gov/archive/docs/v2-2/faq#can-i-run-x11-apps-through-singularity

52. Viereck M. X11docker: Run GUI applications in Docker containers. Journal of Open Source Software. 2019;4: 1349. doi:10.21105/joss.01349

53. Yaremenko E. JAremko/docker-x11-bridge [Internet]. 2019. Available: https://github.com/JAremko/docker-x11-bridge

54. Panda Y. Yuvipanda/jupyter-desktop-server [Internet]. 2019. Available: https://github.com/yuvipanda/jupyter-desktop-server

55. Emsley I, De Roure D. A Framework for the Preservation of a Docker Container International Journal of Digital Curation. International Journal of Digital Curation. 2018;12. doi:10.2218/ijdc.v12i2.509

56. Kim B, Ali TA, Lijeron C, Afgan E, Krampis K. Bio-Docklets: Virtualization Containers for Single-Step Execution of NGS Pipelines. bioRxiv. 2017; 116962. doi:10.1101/116962

57. {Cookiecutter contributors}. Cookiecutter/cookiecutter [Internet]. cookiecutter; 2019. Available: https://github.com/cookiecutter/cookiecutter

58. Marwick B. Benmarwick/rrtools [Internet]. 2019. Available: https://github.com/benmarwick/rrtools

59. Marwick B. README of 1989-excavation-report-Madjebebe. 2015; doi:10.6084/m9.figshare.1297059

60. Wikipedia contributors. Cron [Internet]. Wikipedia. 2019. Available: https://en.wikipedia.org/w/index.php?title=Cron&oldid=929379536

61. Eglen S, Nüst D. CODECHECK: An open-science initiative to facilitate sharing of computer programs and results presented in scientific publications. Septentrio Conference Series. 2019; doi:10.7557/5.4910

62. Schönbrodt F. Training students for the Open Science future. Nature Human Behaviour. 2019;3: 1031–1031. doi:10.1038/s41562-019-0726-z

63. Eglen SJ, Mounce R, Gatto L, Currie AM, Nobis Y. Recent developments in scholarly publishing to improve research practices in the life sciences. Emerging Topics in Life Sciences. 2018;2: 775–778. doi:10.1042/ETLS20180172

64. Blank D, Twitter. Twitter thread on reproducibility time capsules on Twitter [Internet]. Twitter. 2019. Available: https://twitter.com/dougblank/status/1135904909663068165

65. Stark PB. Before reproducibility must come preproducibility [Internet]. Nature. 2018. doi:10.1038/d41586-018-05256-0