Ten Simple Rules for Writing Dockerfiles for Reproducible Research

Daniel Nüst [*], Vanessa Sochat , Ben Marwick , Stephen 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 a plain-text file. For example, one such container technology, Docker, provides instructions using a `Dockerfile`. By following the rules in this article researchers writing a `Dockerfile` can effectively build and distribute containers.

Introduction

With access to version control systems (VCS, [1]) and collaboration platforms based on these, such as GitHub or GitLab, it has become increasingly easy to not only share algorithms, but also instructions for executing research workflows, including building and testing of the software used. The publication of these instructions and related files (also known as 'artefacts') are a response to the increasing complexity of computer-based research, which is too complicated for "papers" based on the traditional journal article format to fully communicate the details [2], but where the actual contribution to knowledge is the full software environment that produced a result [3]. A *research compendium* is a compilation of data, code, and documentation that accompanies, or is itself, a scholarly publication for increased transparency and reproducibility (cf. `https://research-compendium.science`). Within such a compendium, it is desirable to include instructions, i.e. a human- and machine-readable recipe, for building containers that capture the computing environment, i.e., all software and data dependencies. By providing this recipe, authors of scientific articles greatly improve their documentation and apply one important part of common practices for scientific computing [4,5], 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 [6] and reproducible workflows (cf. Rule 10 of [7]).

While there are several tutorials for using containers for reproducible research [8–12], there is no extensive examination for how to write the actual instructions to create the containers. Several platforms for facilitating reproducible research are built on top of containers [13–17], but they hide most of the complexity from the researcher. Because *"the number of unique research environments approximates the number of researchers"*

[17], sticking to conventions helps every researcher to understand, modify, and eventually write container recipes, even if the are not sure how the technology behind them actually works. Therefore researchers publishing a research compendium should craft their own definition of computing environments following good practices.

While there are many different container technologies, this article focuses on Docker [18]. Docker is a highly suitable tool for reproducible research (e.g., [19]) 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` so that it best facilitates interactive development and computer-based research, as well as the higher goals of reproducibility and preservation of knowledge. Such practices are generally not part of generic containerization tutorials and are rarely found in published `Dockerfile`s, 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 software development experience.

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 single command, for example `make`. 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, but may also be opening and starting a process 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 containerization.

A commitment to these general practices can ensure that workflows are reproducible, and generation of a container is not merely triggered by the publication of a finished project (cf. thoughts on openness as an afterthought by [20] and on computational reproducibility by [3]). By following the *conventions* laid out in these ten rules, authors ensure readability by others and enable subsequent reuse and collaboration.

Docker

Docker is a container technology that is widely adopted and supported on many platforms, and has become highly useful in science. They are distinct from virtual machines (VM) or hypervisors as they do not emulate hardware, and thus do not require the same system resources. To build Docker containers, we write text files that follow a particular format called `Dockerfile`s [21]. `Dockerfile`s are machine- and human-readable recipes for building images. Images are inert, immutable, read-only files that includes the application (e.g., the programming language interpreter needed to run the workflow) and the environment required by the application to run. They consist of a sequence of instructions, which add layers to the image. These layers can be used for caching across image builds, which is important for minimizing build and download times. The images can then be run as stateful containers, which are the running instances of Docker images. Containers can be modified, stopped, restarted and purged.

While Docker was the original technology to support this format, other container technologies have developed around the format (and thus support it) including: podman/buildah supported by RedHat, kaniko, img, and buildkit. The Singularity container software [22] is optimized for high performance computing and although it uses its own format, the *Singularity recipe*, it can import Docker containers directly from a Docker registry. 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. [23]) and there are substantial technical challenges to maintain software and documentation, providing a `Dockerfile` or pre-built Docker or other type of container should become an increasingly easier task for the average researcher. If a researcher is able to find or create containers or `Dockerfile`s to address their most common use cases, then arguably it will not be extra work after this initial set up (cf. README of [24]). In fact, the `Dockerfile` itself can be used as documentation to clearly show from where data and code was derived, i.e. downloaded or installed, and consequently where a third party might obtain them again.

1. Consider tools to assist with Dockerfile generation

Writing a `Dockerfile` from scratch is not that simple, 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. It is always a good practice to start with your specific use case. You first 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 the software that you need. 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. Well maintained means that recent container bases are used, likely from the official Docker library [25], to ensure that the container has the most recent security fixes for the operating system in question. As an example, repo2docker [16] is a tool that is maintained by Jupyter Labs that can help to transform a repository in the format of some known kind of package (notebook, Python, R, etc.) into a container. Such a package contains well-defined files for defining software dependencies and versions, which repo2docker can understand. 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 package 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 easily interact with any existing workflows in the repository. 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 build their own container. For this reason, it is good practice to look at any help command provided by the tool and check for configuration options for user names, user ids, and similar. It's also recommended (if you are able) to add custom labels to your container build to define metadata for your analyses (see Rule 6). The container should be built to be optimized for its use case, whether it is intended to be shared or used by a single user.

Additional tools to assist with writing Dockerfiles include `containerit` [26] and `dockta` [27]. `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.

However, in the case that a tool or interactive container environment is not available, or their capabilities to configure specific resources or install bespoke tools are limited, you will likely need to write a `Dockerfile` from scratch. In this case, proceed with following the remaining rules to write your `Dockerfile`.

2. Use versioned and automatically built base images

It's good practice to use base images that are maintained by the Docker library. While some organizations can be trusted to update containers with security fixes (e.g., `rocker/r-ver`), for most individual accounts that build containers, it is likely that containers will not be updated regularly. It is even possible that images or `Dockerfiles` may disappear. Thus a good understanding of how base images and image tags work is crucial, as the tag that you choose has implications for your image build. You should know how to use `FROM` statements to trace all the way up to the original container base, an abstract base called `scratch`. Doing this kind of trace is essential because it makes clear all of the steps that were taken to generate your container and who took them. If you want to build on a third party image, carefully consider the author/maintainer and *never* use an image without a published `Dockerfile`. If you want to use an unofficial image, do save a copy of the Dockerfile(s) created by others. Alternatively, copy relevant instructions into your own Dockerfile, acknowledge the source in a comment, and configure an automated build for your own image. Automated builds can be complex to set up, and details are out of scope of this article. The documentation for, and toolsets provided by container registries such as Docker Hub or GitLab as well as CI platforms such as GitHub actions or CircleCI can help you get started; tools may also be available to automate builds using intermediate tools, such as the `repo2docker` Github action [28,29], You should avoid publishing an image in a public registry with `docker push`, because it is opaquely breaking the linkage between `Dockerfile` and image.

A tag like `latest` is good in that security fixes and updates are likely present, however it is bad in that it is a moving target so that it is more likely that an updated base could break your workflow. Other tags that should be avoided are `dev`, `devel`, or `nightly` that may provide possibly unstable versions of the software. Official builds are often tagged with corresponding release version numbers as well as with the `latest` tag. For example, if you build a container with `python:latest` that only supports Python version 3, when the base is updated to Python 4 it's likely that your software won't work as expected. In this case, a tag like `python:3.5` might be a good choice, where the third component of the version, i.e., `3.5.x`, ensures that security patches and bugfixes that won't break your code, i.e., changes that are backwards compatible, are applied (cf. semantic versioning, [30]). When you choose a base image, try to choose one with a Linux distribution that supports the software stack you are using, and also take into account the bases that are widely used by your community. As an example, Ubuntu is heavily used for geospatial research, and so the `rocker/geospatial` image would be a good choice for spatial data science with R, or `jupyter/tensorflow-notebook` could be a good choice for machine learning with Python.

Container tags may also provide an indication as the size of a container. Smaller images are often indicated by having `slim` or `minimal` as part of the tag. If you need to build smaller containers, consider a `busybox` base or a *multi-stage build* [31], which allows you to selectively copy files from one build step to another. Also take into account the software that you actually need, but mostly for clarity. In scientific contexts, the image size will be often the least of you concerns, because data size is much larger or image portability is not time critical.

Base images that have complex software installed (e.g. machine learning tool stacks, specific BLAS library) are helpful and fine to use. If you do not want to rely on a third party or other individual to maintain the recipe and container, you should copy the `Dockerfile` to your own repository, and also provide a deployment for it via your own automated build. Trusting that the third party `Dockerfile` will persist for your analyses may be risky because that `Dockerfile` could disappear without warning. Here is a selection of communities that produce widely used regular builds and updates:

- Rocker for R [19]
- Docker containers for Bioconductor for bioinformatics
- NeuroDebian images for neuroscience [32]
- [Jupyter Docker Stacks](https://jupyter-docker-stacks.readthedocs.io/en/latest/index.html for Notebook-based computing
- Taverna Server for running Taverna workflows

For example, here is how we would use a base image `r-ver` with tag `3.5.2` from the `rocker` organization on Docker Hub (`docker.io`).

```
FROM rocker/r-ver:3.5.2
```

3. Use formatting, document within, and favor clarity

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 easily 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.

You can use a linter [33] to avoid small mistakes and follow good practices from software development communities. The consistency added by linting also helps keeping your edits to a `Dockerfile` in a version control system (VCS) meaningful (see Rule 9). Note however that a linter's rules may not primarily serve the intention of reproducible scientific workflows.

As you are writing the `Dockerfile`, be mindful of how other people will read it. 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. Comments should include helpful usage guides and links for readers inspecting the `Dockerfile`, including future you. Dependencies can be grouped in this fashion, which also makes it easier to spot changes if your `Dockerfile` is managed in a VCS (see Rule 9). It can even be helpful to include comments about commands that did not work so you do not fall 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`.

Labels are useful to provide a more structured form of documentation about software, especially for users of supporting infrastructure that might not expose the actual `Dockerfile` (see Rule 6)¸. It is often helpful to provide commented lines with `docker build` and `docker run` within the `Dockerfile` to show how to build and run the image, even though these lines are not necessary for a valid Dockerfile, this is a convenient place to record them. These comments can be especially relevant if volume mounts or ports are important for using the container, and by putting them at the end of the file, they are more likely to be seen, easy to copy-paste after a container build, and to be in consistent state compared to documentation in another file. Here is an example of a commented section to show build and usage.

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

If you were to discover a previously written Dockerfile and 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"*. 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 availa
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 sourc

# following commands from instructions at LINK HERE
```

Clarity is 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 Dockerfiles aiming at small image size, you should keep the script in the container for a future user to inspect. 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 externalization of content to outside of the Dockerfile (see Rule 5). 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 statements so that each performs one scoped action (e.g., download, compile, and install one tool). Each statement 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 [34]. A RUN statement longer than a page requires scrolling, diminishing readability. This may be challenging for the next reader to digest and you should consider splitting it up, being aware of the extra layers added to the image.

When you install several system libraries, it is good practice to add comments about why the dependencies are needed. 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.

If you intend to build the image more than once (perhaps during development) and you can take advantage of build caching to avoid execution of time-consuming instructions, e.g., install from a remote resource or a file that gets cached. You should list commands *in order* of least likely to change to most likely to change and use the `--no-cache` flag to force a re-build of all layers. A recommended ordering based on this metric might be:

1. system libraries
2. language-specific libraries or modules
3. from repositories (binaries)
4. from source
5. own software/scripts (if not mounted)
6. labels
7. `RUN`/`ENTRYPOINT`

Finally, as a supplement to content inside the `Dockerfile`, it is good practice to also write a section in a `README` alongside the `Dockerfile` for exactly how to build, run, and otherwise interact with the container. If a pre-built image is provided on Docker Hub, you should direct the user to it in your `README`.

4. 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).

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 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`

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, and many packages are tested before release. Most package managers have a command line interface that can easily 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 [6]) The biggest risk with using package managers with respect to `Dockerfile`s is outsourcing configuration to file formats that are not supported. As an example, here are configuration files supported by commonly used languages in scientific programming:

- Python: `requirements.txt` (pip tool, [35]), `environment.yml` (Conda, [36])
- R: `DESCRIPTION` file format [37] and `r` ("littler", [38])
- JavaScript: `package.json` of `npm` [39]
- Julia: `Project.toml` and `Manifest.toml` [40]

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` statement 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 (cf. Rule 3), which can be mitigated by carefully organizing all these files in the same version-controlled project. The version pinning capabilities of these file formats are described in their respective documentation.

5. Add user scripts and data into containers by mounting volumes

The role of containers is to provide the methods, not to encapsulate datasets. 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 *volume 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 easily accessible to other users if published in an online research compendium. You can `COPY` 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.

Standalone *script files* are distinct from other software as they are not packaged and versioned, and are treated as files, not as managed software. If you developed software for a specific analysis in form of a software package, you should publish in a public source code or software repository and follow Rule 4 for installing it. You should avoid installing software packages from source after `COPY`ing the code it into the image, because the connection between the file outside of the image and the one copied in is easily lost. Consider using the suitable package system even for small scripts if the functions are reusable across datasets or workflows by you or others.

Storing *data files* outside of the container further allows handling of very large datasets and datasets with data worthy of protection, e.g., proprietary data or personal information. For 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.

You can use the -v/--volume or --mount flags to docker run to configure bind mounts of directories or files [41], 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 directory with write access relative to current path (Linux)
docker run --volume $(pwd)/inputdata:/data:rw mycontainer
```

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.

6. Capture structured environment metadata

Labels and build arguments can be very helpful to both provide metadata and allow for customization of a build.

Labels

Labels serve as structured metadata that can be exposed by APIs, e.g., https://microbadger.com/labels, along with tools to inspect the container binaries, e.g., docker inspect. For example, software versions, maintainer contact information, along with vendor specific metadata are commonly seen. The OCI Image Format Specification provides some common label keys (see the "Annotations" section in [42]) to help standardize field names across container tools, as shown below. These labels match the org.label-schema.* specification, which has been deprecated in favour or the new namespace but are still found a lot in existing containers.

```
LABEL org.opencontainers.image.created='2019-12-10' \
  org.opencontainers.image.authors='author@example.org' \
  org.opencontainers.image.url='https://github.com/nuest/ten-simple-rules' \
  org.opencontainers.image.documentation='https://github.com/...' \
  org.opencontainers.image.version='0.0.1'

LABEL org.opencontainers.image.vendor='nuest' \
  org.opencontainers.image.title='Demo title' \
  org.opencontainers.image.description='Demo description'
```

You can add multiple fields within the same instruction. Important metadata attributes to include as labels would include any of the following, ideally with globally unique identifiers:

- Author and contact (e.g., email, project website, or ORCID; you will often see the deprecated `MAINTAINER` instruction - use a label instead)
- Research organizations (identified with https://ror.org/)
- Funding agency/grant number
- A repository link where the `Dockerfile` is published, e.g., a GitHub project or a repository record with a DOI, e.g., Zenodo, where you can pre-register a DOI and add it to your `Dockerfile` before publishing the record
- License

Proper software citation is still a work in progress [43] and you should follow current recommendations of projects such as CodeMeta (`https://codemeta.github.io/`) for describing software and the Citation File Format (`https://citation-file-format.github.io/`) to enable citations of software.

Build arguments

Build arguments can provide more dynamic metadata and also allow for customization of a build. As an example, the following build argument would default to `1.0.0` but allow you to change it with `--build-arg MYVERSION=2.0.0` when building the image:

```
ARG MYVERSION=1.0.0
```

Along with specifying versions, e.g., a git commit hash, or adding a date and timestamp, build arguments can be useful to provide the context of the build, e.g., building user, production versus development environment, and automated or not. Examples of build arguments that are useful to include to describe a container are:

7. 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, e.g. as demonstrated in [44]. A workflow that does not at all support headless execution may even be seen as not reproducible. These two usages can be configured by the Dockerfile author and exposed to the user based on the Dockerfile's `CMD` and `ENTRYPOINT` instructions. 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 [45], 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. In any case you should document different variants, if you choose to provide them, in a `Makefile` [46]. To support both one click execution and interactive interfaces and even allow for custom configuration, it's helpful to expose settings via a configuration file which can be bound from the host, via environment variables [47], or special Docker-based wrappers such as Kliko [48].

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>`. A person who is unfamiliar with Docker but wants to use your image may rely on graphical tools like Kitematic [49] or ContainDS for assisstance. 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 [..]
```

Interactive usage of a command-line interfaces (CLI) are quite straightforward to access from containers, if users are familiar with them. 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.

8. Establish templates for new projects

It is likely going to be the case that over time you will develop workflows that are similar in nature to one another. In this case, you should consider adopting a standard workflow that will give you a clean slate for a new project. If you decide to build your own standard, 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 container image to a suitable container registry, and taking into consideration of how the code can be given a DOI or proper publication (e.g., Zenodo, Journal of Open Source Software).

As an example, cookie cutter templates [55], project starter kits [REF], or community templates (e.g., [56]) can provide files, directory organization, and build instructions that include basic steps for getting started. A good project template should get you started with a template for documentation, setting up testing via continuous integration (CI), building a container, and even choosing a license. In the case of using a common `Dockerfile` or base, Docker's build caching will take shared lines into account and speed up build time even between projects. When developing or working on projects with containers you can easily 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. At most, if a port is shared by two projects to expose a user interface in the browser, you would need to configure non-conflicting ports.

In the case of more complex web applications that require applications and web servers, databases, and workers or messaging, the entire infrastructure can easily be brought up or down with a combination of templates and orchestration tools like `docker-compose` [57]. `docker-compose` also allows definition of services using multiple containers via it's own `docker-compose.yml` file. This file can help to template options including mounted volumes, to permissions, environment variable, and exposed ports.

9. Publish one Dockerfile per project in a code repository with version control

Because a `Dockerfile` is a plain text-based format, it works well with version control systems. Including a `Dockerfile` alongside your code and (if size permits) data 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. [58]). Online collaboration platforms (e.g., GitHub, GitLab) also make it easy to use CI services to test building your image in an independent build 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 [16], 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. [59]), it's generally a simple and clear approach to provide one `Dockerfile` per project. If you find that you need to provide more than one, use `docker-compose` and consider if it's possible to use build arguments to flip between states (e.g., development vs. production, see Rule 7) or to separate tools into different repositories.

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 [60]. You should prioritize 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 7).

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 that the longer that you wait to rebuild the image, the more likely you are to encounter an error that will interfere with the build process. 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 not only increases security due to updating underlying software, but also helps to reveals issues requiring manual interference, i.e., changes to code or configuration not documented in the `Dockerfile` (but perhaps should be). This habit can be easily supported by using continuous deployment or CI strategies. If the container is linked to an automated build, then pushing updates to a VCS repository with a `Dockerfile` can easily trigger a build to validate execution of the `Dockerfile`.

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.

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 [61].

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. Depositing the image with other project files (e.g., data, code, `Dockerfile`) in a public repository makes them likely to be preserved. Applying proper preservation strategies (cf. [58]) can be highly complex, but simply running an image "as-is", i.e. with the default command and entrypoint (see Rule 7), 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 [34]. 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 4) 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 [20,62–64].

Example Dockerfiles

To demonstrate the ten rules, we maintain a GitHub repository with example `Dockerfile`s, some of which we took from public repositories and updated to adhere 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. In this article we have provided guidance for using `Dockerfile`s in computational research. Our goal is to help the researcher to work towards creating a "time capsule" [65] which, given some expertise and the right tools, can be used to come as close as possible to the original workflow with reasonably little effort. 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 (cf. [66]'s notion of "preproducibility"). So please, make a best effort with your current knowledge, and strive to write efficient, readable `Dockerfile`s 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.

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. SE 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. Wikipedia contributors. Version control [Internet]. Wikipedia. 2019. Available: https://en.wikipedia.org/w/index.php?title=Version_control&oldid=926593231

2. 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

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

4. 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

5. 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

6. 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

7. 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

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. 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

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

15. 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

16. 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

17. 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

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

19. 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

20. 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

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

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

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

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

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

26. 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

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

28. Husain H, Silkaitis R. Machine-learning-apps/repo2docker-action [Internet]. ML Apps; 2019. Available: `https://github.com/machine-learning-apps/repo2docker-action`

29. Henderson S. Scottyhq/repo2docker-githubci [Internet]. 2019. Available: `https://github.com/scottyhq/repo2docker-githubci`

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

31. Docker Inc. Use multi-stage builds [Internet]. Docker Documentation. 2020. Available: `https://docs.docker.com/develop/develop-images/multistage-build/`

32. 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

33. Wikipedia contributors. Lint (software) [Internet]. Wikipedia. 2019. Available: `https://en.wikipedia.org/w/index.php?title=Lint_(software)&oldid=907589761`

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

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

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

37. 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

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

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

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

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

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

43. 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

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

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

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

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

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

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

58. 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

59. 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

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

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

62. 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

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

64. 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

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

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