

A Reproducible Data Analysis Workflow With R Markdown, Git, Make, and Docker

Aaron Peikert^{1,2} , Andreas M. Brandmaier^{1,3} 

[1] Center for Lifespan Psychology, Max Planck Institute for Human Development, Berlin, Germany. [2] Department of Psychology, Humboldt-Universität zu Berlin, Berlin, Germany. [3] Max Planck UCL Centre for Computational Psychiatry and Ageing Research, Berlin, Germany.

Quantitative and Computational Methods in Behavioral Sciences, 2021, Article e3763,
<https://doi.org/10.5964/qcmb.3763>

Received: 2020-05-27 • Accepted: 2021-01-25 • Published (VoR): 2021-05-11

Corresponding Author: Andreas M. Brandmaier, Max Planck Institute for Human Development, Lentzeallee 94, 14195 Berlin, Germany. E-mail: brandmaier@mpib-berlin.mpg.de

Supplementary Materials: Materials [see [Index of Supplementary Materials](#)]



Abstract

In this tutorial, we describe a workflow to ensure long-term reproducibility of R-based data analyses. The workflow leverages established tools and practices from software engineering. It combines the benefits of various open-source software tools including R Markdown, Git, Make, and Docker, whose interplay ensures seamless integration of version management, dynamic report generation conforming to various journal styles, and full cross-platform and long-term computational reproducibility. The workflow ensures meeting the primary goals that 1) the reporting of statistical results is consistent with the actual statistical results (dynamic report generation), 2) the analysis exactly reproduces at a later point in time even if the computing platform or software is changed (computational reproducibility), and 3) changes at any time (during development and post-publication) are tracked, tagged, and documented while earlier versions of both data and code remain accessible. While the research community increasingly recognizes dynamic document generation and version management as tools to ensure reproducibility, we demonstrate with practical examples that these alone are not sufficient to ensure long-term computational reproducibility. Combining containerization, dependence management, version management, and dynamic document generation, the proposed workflow increases scientific productivity by facilitating later reproducibility and reuse of code and data.

Keywords

reproducibility, R, version management, dynamic document generation, dependency management, containerization, open science



This is an open access article distributed under the terms of the [Creative Commons Attribution 4.0 International License](#), CC BY 4.0, which permits unrestricted use, distribution, and reproduction, provided the original work is properly cited.

In this tutorial, we describe a workflow to ensure long-term and cross-platform reproducibility of data analyses in R ([R Core Team, 2020](#)). Reproducibility is the ability to obtain identical results from the same statistical analysis and the same data. For us, statistical results are only reproducible if their generating, computational workflow is reported completely and transparently, and remains permanently available, such that the workflow can be re-run by a different person or later in time, and that the results remain identical to those initially reported ([Claerbout & Karrenbach, 1992](#); [Heroux, Barba, Parashar, Stodden, & Taufer, 2018](#); [The Turing Way Community et al., 2019](#)). The need to ensure reproducibility directly follows from commonly accepted rules of good scientific practice (such as the guidelines of the German Research Foundation; [Deutsche Forschungsgemeinschaft, 2019](#)). Ensuring reproducibility is a prerequisite for replicability (the ability to reach consistent conclusions from the same analysis and *new* data), and a means to increase the trustworthiness of empirical results ([Epskamp, 2019](#)). Transparency and accessibility are central scientific values, and open, reproducible projects will increase the efficiency and veracity of knowledge accumulation ([Nosek & Bar-Anan, 2012](#)).

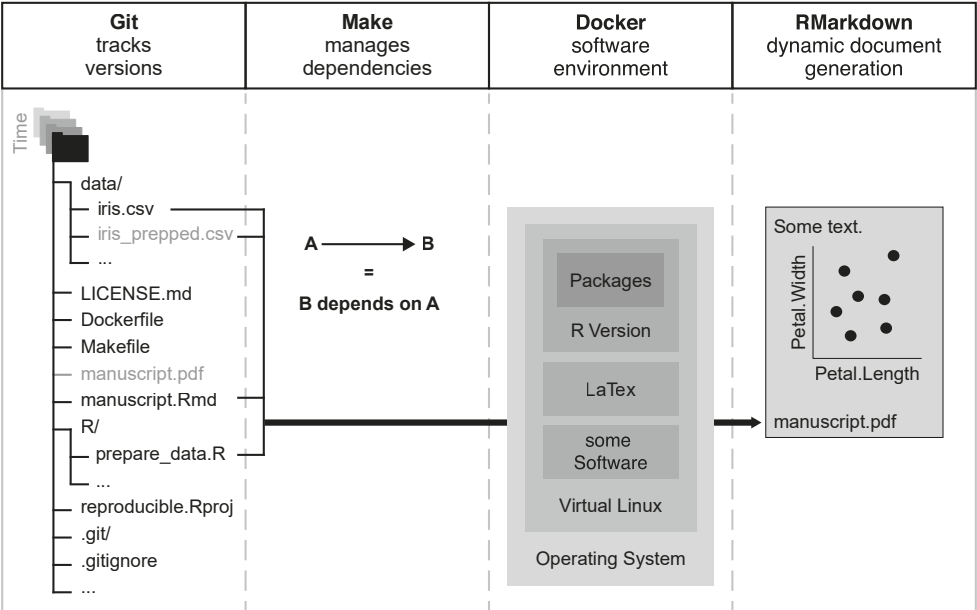
Here, we combine four software tools, whose interplay can guarantee full computational reproducibility of data analyses and their reporting. There are various ideas on how to enhance reproducibility ([Piccolo & Frampton, 2016](#)), four of which we believe to be particularly important: dynamic document generation: ([Rule et al., 2019](#)), version control ([Barba, 2016](#)), dependency management ([Askren et al., 2016](#)), and containerization ([Clyburne-Sherin, Fei, & Green, in press](#)). We argue that only a workflow using all four concepts in unison can guarantee confidence in reproducing a scientific report (see [The Turing Way Community et al., 2019](#) for similar arguments). Various implementations of these concepts exist, but we consider the following four best suited for analyses centered on the R environment ([R Core Team, 2020](#)) but also allowing for external dependencies: R Markdown ([Xie, Allaire, & Golemund, 2018](#)) for dynamic document generation, Git ([Chacon & Straub, 2014](#)) for version control, Make ([Feldman, 1979](#)) for dependency management, and Docker ([Merkel, 2014](#)) for containerization. Each of these software solutions serves a valuable meta-scientific goal (reproducibility) and increases the researchers' productivity. They are all very flexible and powerful, so their complete mastery requires a significant amount of practice. However, for our purposes, it is sufficient to master a valuable minimal subset of functions to ensure the reproducibility of scientific analyses. We recommend using RStudio, an integrated development environment (IDE) for R, which provides simplified access to essential features of some of the tools.

Components of the Reproducible Workflow

The Reproducible Workflow in a Nutshell

Figure 1 gives an overview of how the four components of our workflow interact to ensure computational reproducibility. Before we describe the four components in more detail, we begin with a minimal description of the roles of each component. In the remainder of this tutorial, we will further detail each of the four components of our workflow.

Figure 1
Schematic Illustration of the Interplay of the Four Components Central to the Reproducible Workflow



Note. Git tracks changes to the project over time; Make manages dependencies among the files; Docker provides a container, in which the final report is built using dynamic document generation in R Markdown. Git = Version Control; Make = Dependency Management; Docker = Containerization; R Markdown = Dynamic Document Generation

The first component is version control. Version control manages changes to files (e.g., data and code) over time so that you can recall specific versions of files later or revert the entire project to a past state. Version control offers snapshots of your workflow at different time points identified by a unique identifier. How different parts of an analysis and a corresponding report relate to each other and in what order they need to be executed is documented using dependency management. The arrows in Figure 1

visualize dependencies, such as an analysis depending on the availability of a particular data file. Third, all computer code (such as a statistical analysis in R) is executed in a virtual environment that guarantees exact reproduction of results independent of the host operating system, the locally installed R version, and installed package versions. Finally, dynamic document generation (also known as the literate programming paradigm) interweaves human-readable code and computed results (such as point estimates, *p* values, or confidence intervals) to eliminate inconsistency errors such as those arising from copy-and-paste errors.

Dynamic Document Generation

The translation of computational results into a human-readable summary, for example into a technical report, a presentation, or a manuscript, is time-consuming and error-prone. Typical errors result from copy-and-paste mistakes, erroneous rounding, or missed updates of the manuscript when the associated computer code and computed results have changed. In order to create not only fully reproducible results but also fully reproducible reports, we resort to the literate programming paradigm (Knuth, 1984), in which human-readable language and computer code are mixed to create dynamic documents whose order follows the logic of thought rather than the order of the computer. R Markdown is a simple markup language to create dynamic documents with embedded chunks of R code that can be exported to standard formats such as documents (docx, pdf, rtf, epub), presentations (ppt, html) or websites (html) using the **knitr** package (Xie, 2015, 2019). Several packages extend the functionality of **knitr**. Of particular note are the **papaja** package (Aust & Barth, 2018), which offers additional functions to enable American Psychological Association (APA) style document formatting, including a journal-style final typeset format, and the **stargazer** package (Hlavac, 2018), which provides journal-ready tables and reports of statistical models. Figure 2 illustrates R Markdown syntax using the **papaja** package and Figure 3 shows the resulting rendered document.

Figure 2*Exemplary Excerpt of an R Markdown File*

```

### Dynamic Document Demonstration

```{r:setup, echo=FALSE}
library("knitr")
library("papaja")
```

This is a simple analysis of the `sleep` dataset (Student, 1908) taken from
`help(t.test)`.

```{r:t-test}
data("sleep")
result <- t.test(extra ~ group, data = sleep, paired = TRUE)
```

The difference in means of hours slept between the groups
was `r:ifelse(result$p.value >= .05, "***not**", "")`
significantly different from zero (`r:apa_print.htest(result)$full_result`).

```

Note. This excerpt of an R Markdown file shows a combination of executable R code, which will be dynamically rendered to content on document creation, and English manuscript text. Code is either given in separate chunks (shown in grey background delimited by triple backticks) or inline (single backticks). The resulting document is shown in [Figure 3](#).

Figure 3*Rendered Result of the Source Code Shown in [Figure 2](#)*

Dynamic Document Demonstration

This is a simple analysis of the `sleep` dataset (Student, 1908) taken from `help(t.test)`.

```
data("sleep")
result <- t.test(extra ~ group, data = sleep, paired = TRUE)
```

The difference in means of hours slept between the groups was significantly different from zero ($M_d = -1.58$, 95% CI $[-2.46, -0.70]$, $t(9) = -4.06$, $p = .003$).

Version Control

Fundamentally, reproducibility means that computational results remain identical if neither the script nor the data have changed. It is often not trivial to find out whether any element in a project has changed over time and if so, to “go back in time.” The Git program enables you to do both. A good mental model for Git is that it takes a sequence of snapshots of all files it is supposed to track. In the language of Git, these snapshots are “commits.” A commit represents a complete copy of the state of all tracked files. Each commit has a short, unique identifier (a hash code) and a human-readable description (commit message). Going back to one state is as easy as traversing the history of all commits and switching the repository to a given previous state; it is possible to visually compare changes between different versions. The collection of all snapshots is called a “repository,” which ideally tracks your entire R project.

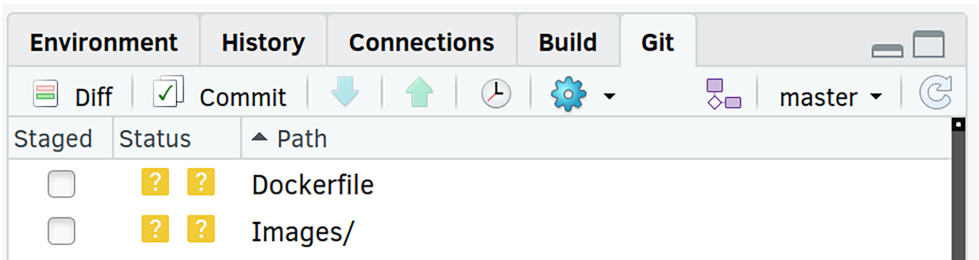
A typical Git workflow in the terminal looks like this:

```
# -- type this on the command line --

git init # to initialize Git in the current directory
git add ./data/iris.csv ./R/analysis.R # track specific files
git commit -m "added data and analysis" # take snapshot with comment
# once script or data were changed, take a new snapshot
git commit -a -m "completed data collection" # add and commit all changes
```

To keep track of all changes on your local computer, you only need to use `git add` and `git commit` or `git commit -a` to add and commit at the same time. Adding a file means to save its changes on the next commit. These commands need to be executed in the terminal, which you can access from within RStudio (Shift + Alt + R). RStudio also offers a graphical user interface for Git. For most basic operations, this interface is convenient and sufficient (see [Figure 4](#)).

Figure 4
Git Pane Providing Easy Access to Basic Functions in RStudio



In a given Git project, you can inspect all changes (`git log`) and examine any previous state by stating the identifier of the commit to `git checkout`:

```
# -- type this on the command line --  
  
# inspect all changes  
git log  
# revert local directory to previous version with hash '77db06f78e'  
git checkout 77db06f78e
```

Git also makes it particularly easy to share and collaborate on a project with other researchers. A popular service for sharing materials via Git is [GitHub](#). Alternatively, institutions can host an equally feature-rich open-source service called [GitLab](#), avoiding the reliance on commercial service providers. At the time of writing, sharing repositories on GitHub with the public is free, private repositories (only visible to persons you invite) are [free for researchers](#) or have limited features. After creating a user account, one can create a new repository and GitHub provides information on how to upload your repository from the terminal, for example, for our repository (here with user name “aaronpeikert” and repository name “reproducible-research”):

```
# -- type this on the command line --  
  
# link remote github repository to local directory  
git remote add origin https://github.com/aaronpeikert/reproducible-research.git  
# push all changes from local repository to the remote repository  
git push -u origin master
```

`git push` or the green upward arrow in the Git pane (see [Figure 4](#)) uploads local updates. To download the remote Git repository on another computer, type into the terminal:

```
# -- type this on the command line --  
  
git clone https://github.com/aaronpeikert/reproducible-research.git
```

Git and GitHub can do even more to support you when collaborating with fellow researchers, for example, by providing a web interface to track issues and their status (open/closed/resolved) and further means to manage and merge multiple, parallel versions of code (such as branches, pull requests, or merges), but this is beyond the scope of this tutorial. In particular, GitHub’s issue management can be leveraged as a post-publication platform to discuss manuscripts and their results (to comment on our paper, please add an issue to the GitHub repository of our paper, see [Supplementary Materials](#)). Another benefit of using Git and GitHub is that experimentation is highly encouraged since you can go back to any state quickly. Even when you lose access to the file on your computer, everything can be backed up on a remote Git server (like GitHub or GitLab).

Further, one can reduce the likelihood of dead code accumulating (e.g., lines that have been commented out) because it is safe to simply remove unneeded code blocks and track their removal in Git.

GitHub allows you to archive and label a specific version of your repository in the form of a release. A release tags a particular commit with an arbitrary label, for example, as “submission,”¹ “preprint,” or “published,”² and archives also “binary” products of your code, for example, the resulting pdf of the manuscript or the docker image (see Section “Containerization”). From such a release, zenodo.org or figshare.com can create a DOI, making it easier to reference and retrieve it (see the [GitHub Guide](#)³).

Dependency Tracking and Management

Even when you have obtained a given version of a project with the aim to reproduce reported results, and you can confirm that this version is unchanged, you may not know exactly how to reproduce the results because it may be unclear which scripts or commands must be executed in which order. This is particularly the case when complex preprocessing pipelines are part of the computation or there are dependencies on external programs. Handling such dependencies is easy with Make because it allows you to manage dependencies by creating (computational) recipes to create or recreate files.

Fundamentally, a `Makefile` is a list of recipes. Each recipe has a target (the name of the recipe) followed by a colon, a list of dependent targets or files, and finally a list of system commands to create the target. This is similar to a cooking recipe where the name of the dish appears first, then the required ingredients and finally the steps to follow to prepare the dish. If any of the dependencies have changed since the last time the target was built, the recipe’s commands are executed to recreate the target file. We illustrate the use of Makefiles with an example. Assume the final product is a manuscript (`manuscript.pdf`). This manuscript is written in R Markdown (`manuscript.Rmd`) and includes dynamically generated plots from a raw data file (`data/iris.csv`) that needs to be preprocessed first using a separate script (`R/prepare_data.R`) into a prepared data file (`iris_prepped.csv`). You find a graphical representation of this example in [Figure 1](#). A `Makefile` for these dependencies may look like this:

1) We created an release for the submission: <https://github.com/aaronpeikert/reproducible-research/releases/tag/v0.1.1-submission>

2) We created a release for the final version: <https://github.com/aaronpeikert/reproducible-research/releases/latest>

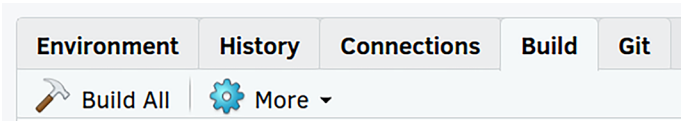
3) Retrieved from <https://guides.github.com/activities/citable-code/>


```
# -- this is a Makefile --
# indent by tabs, not spaces
all: manuscript.pdf

manuscript.pdf: data/iris_prepped.csv manuscript.Rmd
  Rscript -e 'rmarkdown::render("manuscript.Rmd")'
data/iris_prepped.csv: R/prepare_data.R data/iris.csv
  Rscript -e 'source("R/prepare_data.R")'
```

The first line after the comment is the first (default) target called “all,” which depends on manuscript.pdf, which itself is a target. If Make is called without an argument, the first target is built. To create manuscript.pdf (the second target in the file), the file manuscript.Rmd needs to be rendered, which depends on data/iris_prepped.csv. This dependency is itself a target (the third target in the file). To create data/iris_prepped.csv, R/prepare_data.R and data/iris.csv are needed. If you type make manuscript.pdf, Make first checks whether the dependencies do exist and, if not, creates them. Here, if data/iris_prepped.csv does not exist, Make creates it by executing the third target (running the preprocessing script R/prepare_data.R). Also, if one of the dependencies of a target is newer than the target itself, Make updates everything that directly or indirectly depends on the target. Here, if the original data (data/iris.csv) is newer than the preprocessed data (data/iris_prepped.csv) and thus was possibly modified since the preprocessing was done the last time, Make will attempt to recreate data/iris_prepped.csv first before recreating manuscript.pdf. If there is a dependency missing, and there is no target to make it, Make stops with an error message. This way Make ensures that all four dependencies of the manuscript (the raw data data/iris.csv, the data preparation script R/prepare_data.R, the prepared data data/iris_prepped.csv and the manuscript source file manuscript.Rmd) are correctly resolved. It is a convention to have the first target named all, which creates the entire project. Subsequently, the command make without any argument automatically creates everything possible in the project. The button Build All from within RStudio triggers this process (see Figure 5).

Figure 5
Build Pane in RStudio With Access to Makefile Target “All”



If you have followed our workflow as presented thus far, you are (almost) only three commands away from fully reproducing the authors' version of our paper. You simply have to type the following commands on the command line:

```
# -- type this on the command line --

# (1) obtain a local copy of the remote repository
git clone https://github.com/aaronpeikert/reproducible-research.git
# (2) enter the project directory
cd reproducible-research
# (3) run the analysis/data preparation etc. with the local R installation
make all
```

However, if you execute the above on your system, there is a good chance that you cannot reproduce our manuscript and the `make all` command results in an error. Successful reproducibility relies on the crucial assumption that your computational environment is identical or sufficiently compatible to the original one, that is, all required software dependencies need to be installed (e.g., R and all additional R Packages) and no updates or other changes to the computational environment must break or alter the original analysis. As we will shortly see, ensuring full computational reproducibility requires one further level of documentation, that is, documentation and reproduction of the computational environment.

Containerization

Docker is a tool that allows encapsulation, sharing, and re-creation of a computational environment on most operating systems (Windows, macOS, & Linux). Docker achieves these goals by setting up a virtual computer, on which it can execute commands (e.g., installing software). It then saves the resulting state of the virtual computer in what is called an “image.” This image can be started and execute commands on the virtual computer, for example, running `Rscript` or `make`. A running instance of an image is called a container. An image can be transferred and executed on any machine that has Docker installed. Regardless of the machine that is executing the container, the computational environment is identical for the programs running inside the container. The most important advantage over traditional virtual machines is that containers are lightweight: they start rapidly, run with little overhead, and do not need much storage space. Docker achieves this by reusing large parts of the host’s operating system.

With the following example, we demonstrate the importance of documenting and (re-)storing the computational environment. Generally, with containers, we would like to safeguard against changes to the computational environment resulting in unexpected consequences, for example, changes in the functionality or default options in packages or even in the R environment itself. While the R programming language is considered stable and much effort is put into backward compatibility, even basic functions like

`read.csv()` (to load data) or `sample()` (to randomly sample from a set) sometimes change their behaviour from one version to another. For example, to ensure reproducibility of analyses based on a computer's pseudo-random number generator (PRNG), it is good practice to rely on fixed PRNG seeds, which are numeric values that set the PRNG into a deterministic state, that is, the sequence of pseudo-random numbers reproduces exactly. Consider the following R code to randomly draw five numbers between 1 and 10:

```
# -- R code --
set.seed(1234)
sample(1:10, 5)
```

The usual expectation is that this code delivers the same pseudo-random five numbers regardless of the operating system or R Version (because of `set.seed()`). Using Docker, we can start an image which contains the R (Version 3.5.0), and execute the code there.

```
R.version$version.string
set.seed(1234)
sample(1:10, 5)
```

This outputs:

```
## [1] "R version 3.5.0 (2018-04-23) "
## [1] 2 6 5 8 9
```

When executing the code in an image with a more recent version of R (Version 3.6.1), the function returns a different sample despite the identical random seed:

```
R.version$version.string
set.seed(1234)
sample(1:10, 5)
```

This outputs:

```
## [1] "R version 3.6.1 (2019-07-05) "
## [1] 10 6 5 4 1
```

Note, that this is intended behaviour as it is the result of a [bugfix](#) in the random number generator implemented as of R (Version 3.6.0). Now, such changes may strictly render analyses run on previous R versions not reproducible if they contain, for example, multiple imputations, bootstrapping, simulations studies, graphics with random jitter, Bayesian estimations using sampling algorithms (such as Markov Chain Monte Carlo), or similar techniques that involve random sampling. We would like to illustrate this with a

more concrete example (the full R code to reproduce this non-reproducibility is provided in the GitHub repository of this manuscript). We ran a linear regression model on a simulated dataset with two variables x and y with R's `lm()` function regressing x on y . Using the **boot** package (Canty & Ripley, 2019), we bootstrapped the 95% confidence intervals around the regression coefficient estimate with 1,000 bootstrap samples to evaluate whether the estimated confidence interval included zero. To make the analysis reproducible, we set a random seed. We ran this once in R (Version 3.5.0):

```
R.version$version.string
set.seed(seed)
results <- boot(data=simdata, statistic=bs,
                R=1000, formula=y~1+x)

# get beta estimates' confidence intervals
round(confint(results, type = "bca", parm = 2), 4) # parm = 2 -> b

## [1] "R version 3.5.0 (2018-04-23)"

## 2.5 % 97.5 %
## 0.0097 0.3842
```

Subsequently, we ran the identical script with the identical seed in R (Version 3.6.1):

```
R.version$version.string
set.seed(seed)
results <- boot(data=simdata, statistic=bs,
                R=1000, formula=y~1+x)

# get beta estimates' confidence intervals
round(confint(results, type = "bca", parm = 2), 4) # parm = 2 -> b

## [1] "R version 3.6.1 (2019-07-05)"

## 2.5 % 97.5 %
## -0.0005 0.3748
```

As we see from these R outputs, the latter of the estimated confidence intervals does include zero while the former does not. Please note that one could discuss deeper issues with null hypothesis significance testing here, but with this example, we would simply like to stress that computational reproducibility in the strict sense requires capturing the full computational environment.

Only rarely does an analysis depend on base R only. Typically, a considerable number of packages is required that each may depend on multiple other packages. Each update of each package in this dependency hierarchy and updates to base R itself will increase

the likelihood of breaking reproducibility (the resulting frustration is sometimes referred to as *dependency hell*). The whole endeavour of reproducibility is therefore at stake every time an update is rolled out. To ensure long-term reproducibility, our workflow replicates the original computational environment of an analysis exactly. Note, that we do not intend to advocate that software should not be updated; updates typically promote bugfixes and provide new functionality; our point is that full computational reproducibility is only achieved if the software versions used originally are precisely documented. Among other things, this makes it possible to trace back update histories to discover which change in which package caused the non-reproducibility. Quite to the contrary, with containerization, it gets easier than ever to safely update to new versions just by changing the R version number of the Docker image (and reverting back if this update breaks code). This convenience is possible because of the efforts of the [Rocker project](#) (Boettiger & Eddebuettel, 2017), which provides Docker images pre-configured with an installation of selected R versions. These packages are taken from MRAN (Revolution Analytics, 2019), a repository for R packages fixed to the last date on which the R version of the image was the most recent. Building upon these Rocker images, researchers can easily build their own Docker images with all required R packages. The rocker project also provides images that include RStudio (`rocker/rstudio`), the **tidyverse** package (`rocker/tidyverse`) and the **R Markdown** package with LaTeX (`rocker/verse`). Because our workflow relies on R Markdown, we suggest using the `rocker/verse` image (which also contains `rstudio` and `tidyverse`). These images are stored on Dockerhub (<https://hub.docker.com/>).

Building on a basic Rocker image, we can specify further software dependencies in a Dockerfile. For example, the basis for this manuscript's Docker image is the following Dockerfile:

```
# -- this is a Dockerfile --

# Define the R version to be installed from rocker project
FROM rocker/verse:3.6.1
# install CRAN R packages: pacman, here, and pander
RUN install2.r --error --skipinstalled\
    pacman here pander
# install additional R packages from github: papaja and wordcountaddin
# the package version fixed by hash (user/package@hash)
RUN installGithub.r\
    crsh/papaja@b6cd70f benmarwick/wordcountaddin@fdf70d9
# set the working directory inside the container
WORKDIR /home/rstudio
```

The FROM statement specifies which Docker image to use, in this case, the `rocker/verse` image with the tag 3.6.1 (referring to the R Version 3.6.1). The RUN statement describes a command to execute, in this case, to run an R script `install2.r` which is available on all Rocker images, to install the specified packages (here, **pacman**,

here and **pander**). A Dockerfile allows more than one `RUN` statement, executing arbitrary system commands. Those `RUN` statements can install dependencies that are not an R package, for example, other programming languages like python or Matlab. The `WORKDIR` statement is not strictly necessary but simplifies commands. The command `docker build -t image-name` creates an image named `image-name` from the Dockerfile in the project. A way to identify the dependencies automatically and generate a docker image from them is provided in the `liftR` package (Xiao, 2019).

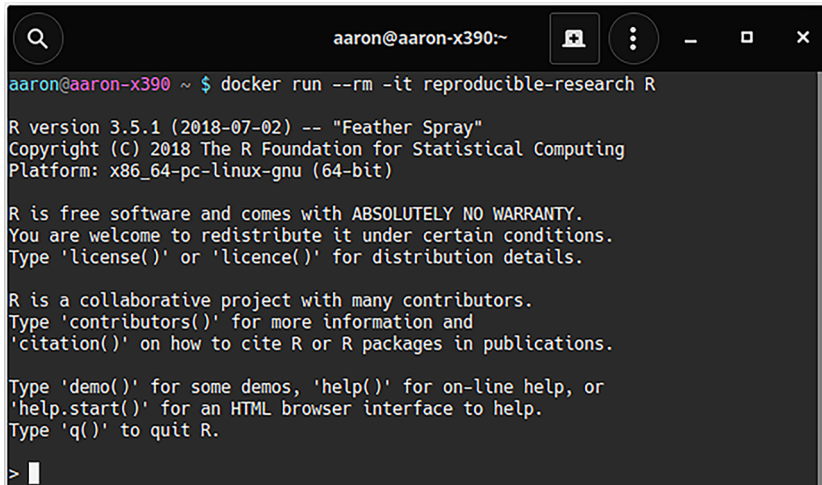
The flexibility to fully control the software environment is of particular interest for software infrastructures where users cannot install software because of limited access rights, for example, on cloud computing platforms or high-performance computing clusters. However, Docker needs unrestricted access rights to the system, which are rarely granted on high-performance computing clusters. For this case, Singularity provides a fully compatible alternative (see Section “Linux”) that can be executed with limited access rights.

There are two ways to share a Docker image; either by sharing the Dockerfile that creates the image or by sharing the image itself, for example, through a service like Dockerhub. While both ways guarantee a replicable computational environment, sharing the Dockerfile is more transparent and more space-saving; in our workflow, we can use Git to track changes in the Dockerfile (such as updates to dependencies). A possible downside is that in order to create an image from a Dockerfile, all software repositories need to be still available. Hence, to guarantee long term reproducibility, it is best to archive the complete binary image at major points of the projects’ progress, for example, on publication (ideally, using a release tag; see Section “Version Control” for details).

There are two options to execute commands in a container. Both options are based on the `docker run` command. The first way is to run a command inside the container. The call takes the form:

```
# -- type this on the command line --  
  
# execute a command in a container image; do not save the state  
# of the container; accept inputs from and return outputs to terminal  
docker run --rm -it <IMAGENAME> <COMMAND>
```

The `--rm` flag means that the state of the container after the command will have finished is not going to be saved. The `-it` flag tells Docker to run the command interactively, that is, to accept keyboard inputs and return outputs to the terminal. For example, this is the command to start an interactive R session inside a Docker image called `reproducible-research` (see Figure 6 for a screenshot):

Figure 6*R Terminal Running Inside Docker*


```

aaron@aaron-x390 ~ $ docker run --rm -it reproducible-research R

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>

```

```

# -- type this on the command line --

# start an interactive R session in the
# container named 'reproducible-research'
docker run --rm -it reproducible-research R

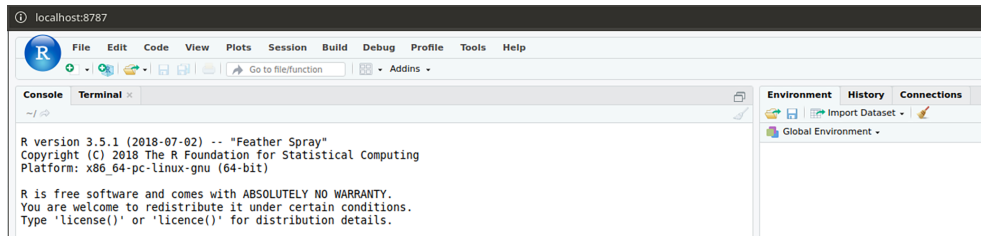
```

The second option is to start the container in the background and to interact with the container via the web browser and the RStudio server instance running in it. In order to do so, you need to supply a password to log into the RStudio server (`-e PASSWORD=<YOUR_PASS>`) and open a local network service on a specified port (`-p 127.0.0.1:8787:8787`).

```
docker run -e PASSWORD=<YOUR_PASS> -p 127.0.0.1:8787:8787 image-name
```

The address to connect to the RStudio server is your IP address (or `localhost` on Linux) in this scheme: `<IPADDRESS>:8787`. This offers a fully functioning RStudio instance that runs in the image but is accessible through a local web browser.

Figure 7 shows a screenshot of Rstudio running inside Docker accessed from a local web browser.

Figure 7*RStudio Running Inside Docker*

By default, programs inside the container cannot access files on the local computer, thus requiring an explicit link to a local folder to enable access (and on macOS and Windows this also has to be allowed in the settings):

```
docker run -v /folder/on/your/computer:/folder/in/docker
```

The main directory for RStudio inside the container is `/home/rstudio`, so the complete call to start RStudio inside a Docker container may look like this in the local terminal:

```
# start docker in the background, open a local web service with a virtual
# Rstudio instance and enable access to selected local directories
docker run --rm -it -e PASSWORD=<YOUR_PASS> -p 8787:8787 -v
/path/to/project:/home/rstudio reproducible-research
```

Figure 7 shows a screenshot of the result.

Since Docker commands tend to grow long and become tedious to type manually, we recommend using some automatic way to generate them. Fortunately, one can use Make to automatically generate the `docker` commands, for example, the (simplified) Makefile for this paper allows the command after `$` (run) to be conditionally passed through Docker if one types `make DOCKER=TRUE` (otherwise, they are run locally):

```
# -- this is a Makefile --
# indent by tabs, not spaces

# set local variables for later use
project := $(notdir $(CURDIR))
current_dir := $(CURDIR)
home_dir := $(current_dir)
uid = --user $(shell id -u)

# determine if DOCKER=TRUE was given
# if so, run everything in docker
# if not, run everything locally
```



```

ifeq ($(DOCKER),TRUE)
  run:=docker run --rm --user $(uid) -v $(home_dir):/home/rstudio $(project)
  current_dir=/home/rstudio
endif

# default target is target 'manuscript.pdf'
all: manuscript.pdf

# build the docker container
build: Dockerfile
  docker build -t $(project) .

# build manuscript.pdf from Rmd file
# run in docker if DOCKER=TRUE else locally
manuscript.pdf: manuscript.Rmd reproducible-research.bib
  $(run) Rscript -e 'rmarkdown::render("$(current_dir)/$<")'

```

Installing and Setting Up the Workflow

Other than on R, RStudio, and R Markdown, our workflow relies on three pieces of software from outside the R environment: Git, Make, and Docker. The smoothness of the installation process of these software packages varies across operating systems. For example, on macOS, Make is always available, whereas Linux systems are typically shipped with both Git and Make. In the following section, we share what we consider the easiest way to install those packages across common operating systems. However, installation processes may be subject to change, and we advise readers to also consult the documentations of the packages or see our collection of links to tutorials and installation instructions on our GitHub repository.

Windows

Windows systems typically require the biggest efforts to install all necessary pieces of software. Note, that you must have either Windows Pro, Enterprise, Education, or Server installed, as Microsoft prevents the use of Docker on Windows Home (see Section “Related Approaches” for alternatives to Docker in case you cannot avoid Windows Home). There is a package manager for Windows called Chocolatey, which you can install from: <https://chocolatey.org/install>. Chocolatey provides all software packages needed for our workflow in one place. Having installed Chocolatey (and restarted the computer), all dependencies can be installed in an **admin terminal** (Windows key, then type cmd, right-click *Run as administrator*) via:

```

# -- type this on the command line --

# install Docker, Make, and Git using Chocolatey
choco install -y git make docker-desktop

```

To use `docker` you need to start Docker Desktop. In the settings of Docker Desktop, you have to allow the sharing of your drive. Docker on Windows requires an unusual path (e.g., `C:\Users\aarons\Documents\reproducible-research` becomes `/c/Users/aarons/Documents/reproducible-research`). Therefore, you currently need to hand-edit the Makefile and set `current_path` to the project directory and use `make all DOCKER=TRUE WINDOWS=TRUE`. We hope that future releases of Docker for Windows will not require that workaround.

macOS

As Make already ships with macOS, you only need Git and Docker. We suggest using the package manager Homebrew, which you can install from <https://docs.brew.sh/Installation>, to install Docker (Git will be installed during the installation of Homebrew):

```
# -- type this on the command line --  
  
# install Docker via Homebrew  
brew cask install docker
```

To use `docker`, you need to start Docker Desktop. In the settings of Docker you have to allow the sharing of your drive.

Linux

There is a host of different Linux distributions and almost as many package managers. Still, to our knowledge, there is no (recent) Linux edition, that does not include Git, Make and Docker. For example, in Ubuntu Linux, installation is straightforward using the shipped package manager:

```
# -- type this on the command line --  
  
# install Docker via advanced package tool  
apt install git make docker
```

For other distributions, replace `apt install` with your package manager's equivalent. You may need elevated rights for the installation; in this case, add `sudo` before the installation command. `docker` also needs elevated rights to run; therefore, we recommend adding the local user to the `docker` group, following the [documentation of Docker](#).

An alternative to Docker on Linux is Singularity (Kurtzer, Sochat, & Bauer, 2017). To use it, just replace any `docker` calls with `singularity docker` because Singularity fully supports `docker` images. A possible advantage is that Singularity works well in high-performance computing environments and on old Linux versions, the downside is that Singularity is currently only available on Linux.

Project Organization

Finally, we conclude with some notes on project organization, which we think makes migrating projects to a reproducible workflow easier. The first step towards reproducibility is to create an R script or R Markdown file as the primary entry point for the analysis that runs on a local computer without error and performs the main statistical analyses. Next, one needs to make sure that all files relevant to the analysis can be moved to another computer. To this end, it is recommended that all files reside within one folder (or enclosed subfolders within it) and all paths are relative to that folder because absolute paths are specific to a given computer. A robust solution to the problem of making sure that file access does not break across computing platforms are [RStudio projects](#) and the **here** package ([Müller, 2017](#)) to manage file access. The **here** package solves two common issues with relative paths. First, it takes care of the fact that path separator characters vary across operating systems (typically, slash or backslash). Second, it solves the issue that anchor points of relative paths may differ depending on context. For example, knitr interprets paths relative to the dynamic document, whereas R has a current working directory that may change over the course of an R session. The **here** package provides consistent paths relative to the project directory. The following three examples refer to local files ranging from absolute paths with system-specific path separators (bad) to relative paths using the **here** package:

```
# -- R code --

# BAD because the path is specific to the computer/user
iris <- read.csv("/home/aaron/reproducible-research/data/iris.csv")
# GOOD because it is a relative path, but slash depends on OS
iris <- read.csv("data/iris.csv")
# BETTER because truly compatible across OS
iris <- read.csv(here("data", "iris.csv"))
```

The folder where all the files reside that you need for an analysis (code and data), is referred to as a “project” (or sometimes as a “research compendium”). Working with projects is particularly convenient with RStudio. It is useful to organize a data analysis project in a way that strictly segregates (raw) data and code by placing them in directories called data and R (see Section 4 in [Marwick, Boettiger, & Mullen, 2018](#)); there are also tools that automatize the standardized creation of folder structures such as [workflowr](#) ([Blischak, Carbonetto, & Stephens, 2019](#)).

Sometimes external requirements make it impossible for the data to be stored and shared with the scripts. In most of the cases we have seen, these are either space constraints or privacy considerations. In these cases, unrestricted reproducibility is not guaranteed. If splitting data and scripts is unavoidable, we recommend validating all data files using checksums (also called a “hash,” e.g., using the functions provided in package **digest**; [Eddelbuettel et al., 2019](#)) before analyzing them. A checksum is a short

fixed-length fingerprint (often displayed in the hexadecimal system) of a file with the purpose of verifying the integrity of a digital object. Fingerprints are computed from digital objects such that they change with high probability if data is changed only a little. To use checksum validation, checksums for all data files must be created and stored at the time of the original analysis. At the time of reproduction, the current checksum must be compared with the stored checksum to ensure data integrity.

```
# -- R code --

# create a dummy data.frame with two columns
x <- data.frame(VAR1=c(1,2,3,4),VAR2=c(0,4,6,9) )
# compute checksum using md5
checksum <- digest::digest(x, "md5")
if (checksum != "5ba412f5a26f43842971dd74954fcdeb"){
  warning("Mismatch between original and current data file!")
}
```

Use Case: Reproducing an Analysis

We provide a reproducible analysis as a working example via [GitHub](#). We encourage interested readers to try to reproduce this example as a practical exercise. The example shows a minimalistic analysis of the Considerations of Future Consequences (CFC) Scale. The analysis demonstrates a complete implementation of our workflow including downloads of external data, comparison of their integrity using a checksum, and a confirmatory factor analysis on the first few items using the R package **lavaan** (Rosseel, 2012). Once all required tools are installed on a computer, the following four command-line commands are sufficient to reproduce our demo analysis:

```
# -- type this on the command line --

# (1) obtain a local copy of the remote repository
git clone https://github.com/aaronpeikert/workflow-showcase.git
# (2) enter the project directory
cd workflow-showcase
# (3) build the docker container
make build
# (4) run the analysis and produce the final PDF inside the container
make all DOCKER=TRUE
```

Summary

The overarching goal of this paper was to provide a complete workflow that allows confidence in the reproducibility of R-based data analyses. Analyses following our workflow can be reproduced with four commands (here shown for this manuscript):

```
# -- type this on the command line --  
  
# (1) obtain a local copy of the remote repository  
git clone https://github.com/aaronpeikert/reproducible-research.git  
# (2) enter the project directory  
cd reproducible-research  
# (3) build the docker container  
make build  
# (4) run the analysis and produce the final PDF inside the container  
make all DOCKER=TRUE
```

The workflow enables the reproduction of a scientific report exactly without regard to the local operating system, locally installed software, time, or interim changes to the project files. To that end, the proposed workflow relies on tools that have been the foundation of reliable software development for years or even decades. As a by-product, it makes transparent how statistical results depend on the software that created them and, by virtue of this transparency, facilitates later reuse by other researchers.

Each tool in the workflow reduces the chances of non-reproducibility. Dynamic reporting with R Markdown guarantees consistency between computational results and their reporting; version control with Git ensures permanence and consistency across multiple versions of data and code; dependency management with Make provides defined entry-points while mapping out dependencies between all components of a project; containerization with Docker guarantees full computational reproducibility. We believe that the proposed combination of tools does not limit researchers but enables them to operate on a solid basis to deliver transparent and sustainable research.

Related Approaches

While our approach was designed to scale well with the complexity of a computationally intense project, we realize that this flexibility may not be straightforward to integrate into researchers' everyday workflow. There are various R packages that implement parts of our workflow and, thus, lower the threshold for adoption when the full flexibility provided by our workflow is not needed. The use of R Markdown within a project, tracked with Git can be simplified with the [workflowR](#) package (Blischak et al., 2019). The [drake](#) package (Landau, 2018) is directly inspired by Make and takes an R-centric approach, making it especially suited for projects only involving R, but it can also handle external dependencies. The [liftR](#) package (Xiao, 2019) and the [holepunch](#) package (Ram, 2019) automatize the use of Docker. The former is perfectly compatible with the described workflow, and we recommend it to users who are not comfortable with command-line use of Docker. **holepunch** uses [binder](#) (Jupyter et al., 2018) to move the analysis to the cloud, so that no local installation of Docker is required. **holepunch** is well suited for simple analyses with low computational demands because binder's memory and computing time is limited. There are several alternatives to Docker that manage dependencies on R packages. [renv](#) (Ushey, 2020) is a way to freeze package

version via local copies of packages in the project, but it does not guarantee a given base R version or system dependencies beyond R. Similar approaches are taken by [jetpack](#) (Kane, 2019), [miniCRAN](#) (de Vries, 2019) and [checkpoint](#) (Microsoft Corporation, 2019). The package [reprex](#) (reproducible example, Bryan, Hester, Robinson, & Wickham, 2019) is also worth noting, but its scope is limited. A particularly noteworthy approach is the [worcs](#) package (van Lissa, Brinkman, et al., 2020; van Lissa, Peikert et al., 2020), which is an R project template that creates a standardized file structure for code and data supporting version management with Git, package management with [renv](#) and dynamic document generation with R Markdown. We acknowledge that [worcs](#) is much easier to install and provides a one-click solution for the creation of reproducible projects. It achieves a high standard of reproducibility but does not guarantee full computational reproducibility and is limited to dependency management within the R environment.

Other than these tools, which ease the process of creating workflows like ours does, we have noticed an increased interest in changing the way research is published and used (Perkel, 2018), with the emergence of *life code* (Perkel, 2019) and *continuous integration* (Beaulieu-Jones & Greene, 2017; Yenni et al., 2018). These techniques give us a glimpse of a paradigm shift from static to dynamic, interactive, and living publications that is yet to happen.

Limitations

We are aware that implementing the proposed workflow is not straightforward, and the difficulty of its implementation may vary by platform. For example, the installation of all tools is already easier on POSIX-compatible platforms such as Unix, Linux, or macOS (but not Windows). However, once a reproducible workflow is established as a default, it can be used with minimal changes for every R project.

In our own experience, it is often not possible to convince all co-authors to switch to a different document processing environment, such as R Markdown. That is, we have experienced the case that after writing up the first draft in R Markdown, we eventually had to generate a Word file that, from then on, was used as static file serving as a basis for multiple iterations among the co-authors. Retaining reproducibility in such situations requires tedious manual synchronization of files across formats. This annoyance may be reduced with the [redoc](#) package (Ross, 2019), which enables a bidirectional synchronization between Word and R Markdown. Conversions between R Markdown and Word retain all changes and support Word's track-changes feature. Hence, R Markdown users can share a Word file with their collaborators, receive their changes in this file and transform it back to R Markdown.

Sharing Reproducible Workflows

How can one best share a reproducible workflow? We believe that, ideally, a non-commercial public service provider should be found that guarantees permanent and reliable hosting of reproducible workflows, such as the Open Science Framework (Foster & Deardorff, 2017). An independent provider mirroring and complementing the services offered by GitHub, Docker Hub, and MRAN would be desirable. Second, to ensure that other users are legally able to benefit from the shared materials, authors must choose an appropriate license. Typically, there is no single license that works for code, data, and media (such as text or figures). We encourage authors to choose appropriate license forms that do not hinder others from freely downloading, using, and modifying the shared workflows and materials while, at the same time, ensuring recognition for the time and effort invested in creating the workflow in the first place. In our experience, the Creative Commons—Attribution license (CC-BY) is often appropriate for sharing texts, R Markdown files, generated figures, and other media, whereas scripts and any other computer code are often best shared under the MIT license (or similar permissive licenses). Both licenses assure maximal freedom for future users while requiring the attribution of the original authors in derivative work. These licenses are also in line with the recommendations by the Reproducible Research Standard (Stodden, 2009; Stodden et al., 2016). A great resource to choose a license is choosealicense.com, however, no resource, including our recommendation, replaces legal advice. To facilitate an inclusive environment, we recommend naming all contributors and including a Code of Conduct⁴ in your project.

Outlook

The proposed workflow leverages various existing tools that are partly integrated into RStudio already. Parts of the proposed workflow have been integrated into stand-alone packages (such as **worcs**, van Lissa, Peikert et al., 2020; **workflowr**, Blischak et al., 2019; or **holepunch**, Ram, 2019), which we recommend to beginners; in particular, **worcs** is a step-by-step procedure with best practices for Open Science from preregistration to publication. Still those approaches do either not guarantee full computational reproducibility or rely on proprietary service providers. We hope that as awareness of the challenges of computational reproducibility increases, the growing demand for unified and open solutions will lead to better integration of existing tools and services so that reproducible workflows become a standard in psychological research.

4) For example, https://www.contributor-covenant.org/version/2/0/code_of_conduct/

Funding: The authors have no funding to report.

Acknowledgments: We are grateful to Julia Delius for her helpful assistance in language and style editing. We thank Michèle Nuijten and the anonymous reviewer, and all contributors to our GitHub repository for helpful feedback on the paper.

Competing Interests: The authors have declared that no competing interests exist.

Supplementary Materials

This paper is fully reproducible using the workflow described here. All materials for doing that, are provide via the GitHub repository (Peikert & Brandmaier, 2020). This paper is based on the commit identified by hash "c4213f6". For adding issues, providing feedback or any other type of comment or questions regarding the workflow described in the paper please add an issue to the GitHub repository (<https://github.com/aaronpeikert/reproducible-research/issues>).

Index of Supplementary Materials

Peikert, A., & Brandmaier, A. M. (2020). *A reproducible data analysis workflow with R Markdown, Git, Make, and Docker* [Materials for reproducing the journal paper]. GitHub.
<https://github.com/aaronpeikert/reproducible-research/>

References

- Askren, M. K., McAllister-Day, T. K., Koh, N., Mestre, Z., Dines, J. N., Korman, B. A., ... Madhyastha, T. M. (2016). Using make for reproducible and parallel neuroimaging workflow and quality-assurance. *Frontiers in Neuroinformatics*, 10, Article 2. <https://doi.org/10.3389/fninf.2016.00002>
- Aust, F., & Barth, M. (2018). *papaja: Create APA manuscripts with R Markdown* (Version 0.1.0.9842). Retrieved from <https://github.com/crsh/papaja>
- Barba, L. A. (2016). The hard road to reproducibility. *Science*, 354(6308), 142.
<https://doi.org/10.1126/science.354.6308.142>
- Beaulieu-Jones, B. K., & Greene, C. S. (2017). Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology*, 35(4), 342-346.
<https://doi.org/10.1038/nbt.3780>
- Blischak, J., Carbonetto, P., & Stephens, M. (2019). *workflowr: A framework for reproducible and collaborative data science* (Version 1.4.0.9001). Retrieved from
<https://github.com/jdblischak/workflowr>
- Boettiger, C., & Eddelbuettel, D. (2017). An introduction to rocker: Docker containers for R. *The R Journal*, 9(2), 527-536. <https://doi.org/10.32614/RJ-2017-065>

- Bryan, J., Hester, J., Robinson, D., & Wickham, H. (2019). *reprex: Prepare reproducible example code via the clipboard* (Version 0.3.0). Retrieved from <https://CRAN.R-project.org/package=reprex>
- Canty, A., & Ripley, B. D. (2019). *boot: Bootstrap r (s-plus) functions* (Version 1.3-23). Retrieved from <https://CRAN.R-project.org/package=boot>
- Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). New York, NY, USA: Apress.
- Clairbout, J. F., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. In *SEG technical program expanded abstracts 1992* (pp. 601–604). <https://doi.org/10.1190/1.1822162>
- Clyburne-Sherin, A., Fei, X., & Green, S. A. (in press). Computational reproducibility via containers in social psychology. *Meta-Psychology*. <https://doi.org/10.31234/osf.io/mf82t>
- Deutsche Forschungsgemeinschaft. (2019). *Leitlinien zur Sicherung guter wissenschaftlicher Praxis*. Retrieved from https://www.dfg.de/download/pdf/foerderung/rechtliche_rahmenbedingungen/gute_wissenschaftliche_praxis/kodex_gwp.pdf
- de Vries, A. (2019). *miniCRAN: Create a mini version of cran containing only selected packages*. Retrieved from <https://CRAN.R-project.org/package=miniCRAN>
- Eddelbuettel, D., Lucas, A., Tuszynski, J., Bengtsson, H., Urbanek, S., Frasca, M., ... Denney, B. (2019). *Digest: Create compact hash digests of r objects* (Version 0.6.21). Retrieved from <https://CRAN.R-project.org/package=digest>
- Epskamp, S. (2019). Reproducibility and replicability in a fast-paced methodological world. *Advances in Methods and Practices in Psychological Science*, 2(2), 145–155. <https://doi.org/10.1177/2515245919847421>
- Feldman, S. I. (1979). Make — A program for maintaining computer programs. *Software: Practice and Experience*, 9(4), 255–265. <https://doi.org/10.1002/spe.4380090402>
- Foster, E. D., & Deardorff, A. (2017). Open Science Framework (OSF). *Journal of the Medical Library Association*, 105(2), 203–206. <https://doi.org/10.5195/jmla.2017.88>
- Heroux, M. A., Barba, L., Parashar, M., Stodden, V., & Taufer, M. (2018). Toward a compatible reproducibility taxonomy for computational and computing sciences (No. SAND2018-11186). <https://doi.org/10.2172/1481626>
- Hlavac, M. (2018). *stargazer: Well-formatted regression and summary statistics tables* (Version 5.2.2). Bratislava, Slovakia: Central European Labour Studies Institute (CELSI). Retrieved from <https://CRAN.R-project.org/package=stargazer>
- Jupyter, P., Bussonnier, M., Forde, J., Freeman, J., Granger, B., Head, T., ... Willing, C. (2018). Binder 2.0—Reproducible, interactive, sharable environments for science at scale. In F. Akici, D. Lippa, D. Niederhut, & M. Pacer (Eds.), *Proceedings of the 17th Python in science conference* (pp. 113–120). <https://doi.org/10.25080/Majora-4af1f417-011>
- Kane, A. (2019). *jetpack: A friendly package manager*. Retrieved from <https://github.com/ankane/jetpack>
- Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>

- Kurtzer, G. M., Sochat, V., & Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5), Article e0177459. <https://doi.org/10.1371/journal.pone.0177459>
- Landau, W. M. (2018). The drake R package: A pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 3(21), 550. <https://doi.org/10.21105/joss.00550>
- Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). *The American Statistician*, 72(1), 80-88. <https://doi.org/10.1080/00031305.2017.1375986>
- Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239). Retrieved from <http://dl.acm.org/citation.cfm?id=2600239.2600241>
- Microsoft Corporation, M. (2019). *checkpoint: Install packages from snapshots on the checkpoint server for reproducibility* (Version 0.4.6). Retrieved from <https://CRAN.R-project.org/package=checkpoint>
- Müller, K. (2017). *here: A simpler way to find your files* (Version 0.1). Retrieved from <https://CRAN.R-project.org/package=here>
- Nosek, B. A., & Bar-Anan, Y. (2012). Scientific utopia: I. Opening scientific communication. *Psychological Inquiry*, 23(3), 217-243. <https://doi.org/10.1080/1047840X.2012.692215>
- Perkel, J. M. (2018). A toolkit for data transparency takes shape. *Nature*, 560, 513-515. <https://doi.org/10.1038/d41586-018-05990-5>
- Perkel, J. M. (2019). Pioneering “live-code” article allows scientists to play with each other’s results. *Nature*, 567, 17-18. <https://doi.org/10.1038/d41586-019-00724-7>
- Piccolo, S. R., & Frampton, M. B. (2016). Tools and techniques for computational reproducibility. *GigaScience*, 5(1), Article 30. <https://doi.org/10.1186/s13742-016-0135-4>
- Ram, K. (2019). *holepunch: Configure your R project for “binderhub”*. Retrieved from <https://github.com/karthik/holepunch>
- R Core Team. (2020). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from <https://www.R-project.org/>
- Revolution Analytics. (2019). *Reproducibility: Using fixed CRAN repository snapshots*. MRAN. Microsoft R Application Network. Retrieved from <https://mran.microsoft.com/documents/rro/reproducibility>
- Ross, N. (2019). *redoc: Reversible reproducible documents*. Retrieved from <https://github.com/noamross/redoc>
- Rosseel, Y. (2012). lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, 48(2), 1-36. <http://www.jstatsoft.org/v48/i02/>
- Rule, A., Birmingham, A., Zuniga, C., Altintas, I., Huang, S.-C., Knight, R., ... Rose, P. W. (2019). Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. *PLOS Computational Biology*, 15(7), Article e1007007. <https://doi.org/10.1371/journal.pcbi.1007007>

- Stodden, V. (2009). *Enabling reproducible research: Open licensing for scientific innovation* (SSRN Scholarly Paper No. ID 1362040). Rochester, NY, USA: Social Science Research Network. Retrieved from <https://papers.ssrn.com/abstract=1362040>
- Stodden, V., McNutt, M., Bailey, D. H., Deelman, E., Gil, Y., Hanson, B., ... Taufer, M. (2016). Enhancing reproducibility for computational methods. *Science*, 354(6317), 1240-1241. <https://doi.org/10.1126/science.aah6168>
- The Turing Way Community, Arnold, B., Bowler, L., Herterich, S. G. P., Higman, R., Krystalli, A., ... Whitaker, K. (2019). *The Turing way: A handbook for reproducible data science* (v0.0.4). <https://doi.org/10.5281/zenodo.3233986>
- Ushey, K. (2020). *renv: Project environments*. Retrieved from <https://CRAN.R-project.org/package=renv>
- van Lissa, C. J., Brinkman, L., Vreede, B., Schoot, R. van de, Peikert, A., & Brandmaier, A. M. (2020). *WORCS: A workflow for open reproducible code in science*. <https://doi.org/10.17605/OSF.IO/ZCVBS>
- van Lissa, C. J., Peikert, A., & Brandmaier, A. M. (2020). *worcs: Workflow for open reproducible code in science*. Retrieved from <https://github.com/cjvanlissa/worcs>
- Xiao, N. (2019). *liftr: Containerize R markdown documents for continuous reproducibility*. Retrieved from <https://CRAN.R-project.org/package=liftr>
- Xie, Y. (2015). *Dynamic documents with R and knitr* (2nd ed.). Boca Raton, FL, USA: CRC press. Retrieved from <https://yihui.name/knitr/>
- Xie, Y. (2019). *knitr: A general-purpose package for dynamic report generation in R* (Version 1.22). Retrieved from <https://yihui.name/knitr/>
- Xie, Y., Allaire, J., & Golemund, G. (2018). *R markdown: The definitive guide*. Boca Raton, FL, USA: CRC press. Retrieved from <https://bookdown.org/yihui/rmarkdown>
- Yenni, G. M., Christensen, E. M., Bledsoe, E. K., Supp, S. R., Diaz, R. M., White, E. P., & Ernest, S. K. M. (2018). Developing a modern data workflow for living data. *bioRxiv*, Article 344804. <https://doi.org/10.1101/344804>