psych

Tutorial

Reproducible Research in R: A Tutorial on How to Do the Same Thing More Than Once

Aaron Peikert [1,2,*], Caspar J. van Lissa [3,4] and Andreas M. Brandmaier [1,5,6]

[1] Center for Lifespan Psychology—Max Planck Institute for Human Development, Lentzeallee 94, 14195 Berlin, Germany; brandmaier@mpib-berlin.mpg.de
[2] Humboldt-Universität zu Berlin, Unter den Linden 6, 10117 Berlin, Germany
[3] Department of Methodology & Statistics—Utrecht University, Faculty of Social and Behavioral Sciences, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands; C.J.vanLissa@uu.nl
[4] Open Science Community Utrecht, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
[5] Max Planck UCL Centre for Computational Psychiatry and Ageing Research, Lentzeallee 94, 14195 Berlin, Germany, and London WC1B 5EH, UK
[6] MSB Medical School Berlin, Rüdesheimer Str. 50, 14197 Berlin, Germany
* Correspondence: peikert@mpib-berlin.mpg.de

Simple Summary: Reproducibility has long been considered integral to the scientific method. An analysis is considered reproducible if an independent person can obtain the same results from the same data. Until recently, detailed descriptions of methods and analyses were the primary instrument for ensuring scientific reproducibility. Technological advancements now enable scientists to achieve a more comprehensive standard that allows anyone to access a digital research repository and reproduce all computational steps from raw data to final report, including all relevant statistical analyses, with a single command. This method has far-reaching implications for scientific archiving, reproducibility and replication, scientific productivity, and the credibility and reliability of scientific knowledge. One obstacle to the widespread use of this method is that the underlying tools are complex and not part of most researchers' basic training. This paper introduces `repro`, an R package that guides researchers through installation and use of the tools required to make a research project reproducible. We also suggest using the proposed workflow for the preregistration of study plans as reproducible computer code (Preregistration as Code; PAC). Since computer code represents the planned analyses exactly as they will be executed, it is more precise than natural language descriptions. PAC circumvents the shortcomings of ambiguous preregistrations that may result in undisclosed use of researcher degrees of freedom. Reproducibility, facilitated by automation, has a wide range of applications and could potentially accelerate scientific progress.

Abstract: Computational reproducibility is the ability to obtain identical results from the *same* data with the *same* computer code. It is a building block for transparent and cumulative science because it enables the originator and other researchers, on other computers and later in time, to reproduce and thus understand how results came about, while avoiding a variety of errors that may lead to erroneous reporting of statistical and computational results. In this tutorial, we demonstrate how the R package `repro` supports researchers in creating fully computationally reproducible research projects with tools from the software engineering community. Building upon this notion of fully automated reproducibility, we present several applications including the preregistration of research plans with code (Preregistration as Code, PAC). PAC eschews all ambiguity of traditional preregistration and offers several more advantages. Making technical advancements that serve reproducibility more widely accessible for researchers holds the potential to innovate the research process and to help it become more productive, credible, and reliable.

Keywords: open science; computational reproducibility; preregistration; R; R Markdown; Make; GitHub; Docker

check for updates

Citation: Peikert, A.; van Lissa, J.; Brandmaier, A.M. Reproducible Research in R: A Tutorial on How to Do the Same Thing More Than Once. *Psych* **2021**, *3*, 836–867. https://doi.org/10.3390/psych3040053

Academic Editor: Alexander Robitzsch

Received: 11 October 2021
Accepted: 25 November 2021
Published: 9 December 2021

1. Introduction

Scientists increasingly strive to make research data, materials, and analysis code openly available. Sharing these digital research products can increase scientific efficiency by enabling researchers to learn from each other, reuse materials, and increase scientific reliability by facilitating the review and replication of published research results. To some extent, these potential benefits are contingent on whether these digital research products are reproducible. Reproducibility can be defined as the ability of anyone to obtain identical results from the *same* data with the *same* computer code (see [1] for details). The credibility of empirical research results hinges on their objectivity. Objectivity in this context means, "in principle it [the research finding] can be tested and understood by anybody." ([2] p. 22). Only a reproducible result meets these requirements. Therefore, reproducibility has long been considered integral to empirical research. Unfortunately, despite increasing commitment to open science practices and good will, many projects can not yet be reproduced by other research teams [3]. This is because there are various challenges to making a research project reproducible (e.g., missing software dependencies or ambiguous documentation of the exact computational steps taken), and there is a lack of best practices for overcoming these challenges (but see [4]). With technological advancement, however, it is now possible to make all digital products related to a research project available in a manner that enables automatic reproduction of the research project with minimal effort.

With this paper, we pursue two aims. First, we want to introduce researchers to a notion of automated reproducibility that requires no manual steps, apart from the initial setup of the software environment. Secondly, we discuss the implications of automated reproducibility for changing the general approach to research. With regard to the first goal, we discuss how to address four common threats to reproducibility, using tools originating from software engineering (see [1] for details), and present a tutorial on how to employ these tools to achieve automated reproducibility. A single tutorial cannot comprehensively introduce the reader to the detail of individual tools, but this tutorial is intended to help readers get started with a basic workflow. The tutorial is aimed at researchers who regularly write code to analyze their data and are willing to make relevant code, data, and materials available, either publicly or on request. Ideally, the reader has already created a dynamic document at some point in time (e.g., with R Markdown or Jupyter) and used some form of version control (e.g., Git). The R package `repro` supports researchers in setting up the required software and in adopting this workflow. We present automated reproducibility as a best practice; a goal that is not always fully achieved due to limited resources, technical restrictions, or practical considerations, but is worth striving for nonetheless.

In pursuit of the second aim, we present a strictly reproducible and unambiguous form of preregistration [5] that builds upon implementing this reproducible workflow, the so-called *Preregistration as Code* (PAC). PAC involves preregistering the intended analysis code and the major part of the final scientific report as a dynamic document, including typical sections like introduction, methods, and results. The resulting dynamic document closely resembles the final manuscript but uses simulated data to generate placeholder results (e.g., figures, tables, and statistics). Simulated data serve two functions, they allow to test the code for the planned analyses and for preregistering the exact presentation of the results. Once the empirical data are available, these replace the simulated data; the results are then updated automatically, and the discussion can be written to finalize the report.

Scientific organizations and funding bodies increasingly demand transparent sharing of digital research products, and researchers are increasingly willing to do so. However, although the sharing of such digital research products is a necessary condition for reproducibility, it is not a sufficient one. This was illustrated by an attempt to reproduce results from open materials in the journal *Cognition* [6]. Out of 35 published articles with open code and data, the results of 22 articles could be reproduced, but further assistance from the original authors was required in 11 of these cases. For 13 articles, at least one outcome could not be reproduced—even with the original authors' assistance. Another study of 62 registered reports found that only 41 had data available, and 37 had analysis scripts available [3]. The authors could execute only 31 of the scripts without error and reproduce the results of only 21 articles (within a reasonable time). These failed attempts to reproduce findings highlight the need for widely accepted reproducibility standards because open repositories do not routinely provide sufficient information to reproduce relevant computational and statistical results. If digital research products are available but not reproducible, their added value is limited.

This tutorial demonstrates how R users can make digital research products more reproducible, while striking a balance between rigor and ease-of-use. A rigorous standard increases the likelihood that a project will remain reproducible as long as possible. An easy-to-use standard, on the other hand, is more likely to be adopted. Our approach is to promote broad adoption of such practices by ensuring a "low threshold", by making it easy to get started, while enabling a "high ceiling" by ensuring that they are compatible with more complex rigorous solutions. As researchers become more proficient in using the tools involved, they can thus further improve the reproducibility of their work.

We have structured the tutorial with a *learning-by-doing* approach in mind, such that readers can follow along on their own computers. We explicitly encourage readers to try out all R commands for themselves. Unless stated otherwise, all code blocks are meant to be run in the statistical programming language R ([7] tested with version 4.0.4).

2. Threats to Reproducibility and Appropriate Remedies

From our own experience with various research projects, we have identified the following common threats to reproducibility:

1. *Multiple inconsistent versions of code, data, or both*; for example, the data set may have changed over time because outliers were removed at a later stage or an item was later recoded; or, the analysis code may have been modified during the writing of a paper because a bug was removed at some point in time. It may then be unclear which version of code and data was used to produce some reported set of results.
2. *Copy-and-paste errors*; for example, results are often manually copied from a statistical computing language into a text processor; if a given analysis is re-run and results are manually updated in the text processor, this may inadvertently lead to inconsistencies between the reported result and the reproduced result.
3. *Undocumented or ambiguous order of computation*; for example, with multiple data and code files, it may be unclear which scripts should be executed in what order; or, some of the computational steps are documented (e.g., final analysis), but other steps were conducted manually without documentation (e.g., executing a command manually rather than in a script; copy-and-pasting results from one program to another).
4. *Ambiguous software dependencies*; for example, a given analysis may depend on a specific version of a specific software package, or rely on software that might not be available on a different computer, or no longer exist at all; or a different version of the same software may produce different results.

We have developed a workflow that achieves long-term and cross-platform computational reproducibility of scientific data analyses. It leverages established tools and practices from software engineering and rests on four pillars that address the aforementioned causes of non-reproducibility [1]:

1. Version control
2. Dynamic document generation
3. Dependency tracking
4. Software management

The remainder of this section briefly explains why each of these four building blocks is needed and details their role in ensuring reproducibility. A more extensive treatment of these tools is given in Peikert and Brandmaier [1].

Version control prevents the ambiguity that arises when multiple versions of code and data are created in parallel during the lifetime of a research project. Version control allows a clear link between which results were generated by which version of code and data. This addresses the first threat to reproducibility, because results can only be said to be reproducible if it is clear which version of data and code produced them. We recommend using `Git` for version control, because of its widespread adoption in the R community.

`Git` tracks changes to all project-related files (e.g., materials, data, and code) over time. At any stage, individual files or the entire project can be compared to, or reverted to, an earlier version. Moreover, contributions (e.g., from collaborators) can be compared to and incorporated in the main version of the project. Version control thus reduces the risk of losing work and facilitates collaboration. `Git` is built around snapshots that represent the project state at a given point in time. These snapshots are called *commits* and work like a "save" action. Ideally, each commit has a message that succinctly describes these changes. It is good practice to make commits for concrete milestones (e.g., "Commented on Introduction", "Added SES as a covariate", "Address Reviewer 2's comment 3"). This makes it easier to revert specific changes than when multiple milestones are joined in one commit, e.g., "Changes made on 19/07/2021". Each commit refers back to its ancestor, and all commits are thus linked in a timeline. The entirety of commits (i.e., the version-controlled project) is called a repository. In `Git`, specific snapshots of a repository can be tagged, such that the user can clearly label which version of the project was used to create a preregistration, preprint, or final version of the manuscript as accepted by a journal. `Git` has additional features beyond basic version control, such as "branches" (parallel versions of a project that can later be merged again) to facilitate simultaneous collaboration. Vuorre and Curley [8] provide a more extensive treatment of how `Git` functions and how to use `Git` for research. Bryan [9] provides additional information on how to track `R Markdown` documents. Collaborating via `Git` is facilitated by uploading the repository to a cloud-based service. We recommend GitHub as a host for `Git` repositories because of its popularity among R users. GitHub has many tools that facilitate working with `Git`— particularly in project management and collaboration—but these are not central to achieving reproducibility.

Second, we rely on *dynamic document generation*. The traditional way of writing a scientific report based on a statistical data analysis uses two separate steps conducted in two different programs. The researcher writes text in a word processor, and conducts the analysis in another program. Results are then (manually) copied and pasted from one program to another, a process that often produces inconsistencies [10].

Dynamic document generation integrates both steps. Through dynamic document generation, code becomes an integral, although usually hidden, part of the manuscript, complementing the verbal description and allowing interested readers to gain a deeper understanding of the contents [11,12]. `R Markdown` uses `Markdown` for text formatting and R (or other programming languages) for writing the statistical analysis. `Markdown` is a lightweight text format in plain text with a minimal set of reserved symbols for formatting instructions. This way, `Markdown` does not need any specialized software for editing. It is userfriendly (unlike, for example, `LaTeX` [13]), works well with version control systems, and can be exported to various document formats, such as HTML websites, a Microsoft Word document, a typeset PDF file (for example, via LaTeX journal templates), or a Powerpoint presentation. `Markdown` can be used for all sorts of academic documents, ranging from simple sketches of ideas to scientific manuscripts [14] and presentations [15], or even résumés [16]. `R Markdown` extends regular `Markdown` by allowing users to include R code chunks (in fact, arbitrary computer code ([17] Chapter 15, Chapter 15, Other Languages)) into a `Markdown` document. Upon rendering the document, the code blocks are executed, and their output is dynamically inserted into the document. This allows the creation of (conditionally) formatted text, statistical results, and figures that are guaranteed to be up-to-date because they are created anew every time the document is rendered to its output format (e.g., presentation slides or a journal article). Xie et al. [17] provides an extensive yet practical introduction to most features of `R Markdown`.

While version control and dynamic document generation are becoming more common, we have argued that two more components are required and that each component alone is unlikely to guarantee reproducibility [1,4]. In practice, dependencies between project files (e.g., information on what script uses which data file and what script needs to be run first) or on external software (e.g., system libraries or components of the programming language, such as other R packages) are frequently unmentioned or not exhaustively and unambiguously documented.

Dependency tracking helps automatically resolve dependencies between project files. In essence, researchers provide a collection of *computational recipes*. A computational recipe describes how inputs are processed to deterministically create a specific output in a way that is automatically executable. The concept of computational recipes is central to our understanding of reproducibility because it enables a unified way to reproduce a project automatically. Similar to a collection of cooking recipes, we can have multiple products (*targets*) with different ingredients (*requirements*) and different steps of preparation (*recipes*). In the context of scientific data analysis, targets are typically the final scientific report (e.g., the one to be submitted to a journal) and possibly intermediate results (such as preprocessed data files, simulation results, and analysis results). A workflow that involves renaming variable names by hand in a graphical spreadsheet application, for example, is therefore incompatible with automated reproducibility. Another property of a computational recipe is that the same inputs should always result in the same outputs. For most computer code (given the same software is used), this property is fulfilled. However, one noteworthy exception is the generation of pseudo-random numbers. Whenever random numbers are used in a computation, it is only reproducible if the random number generator generates the same numbers. To ensure identical random numbers, users may fix the state of the random number generated with a so-called seed (e.g., `set.seed()` in R), but they also need to guarantee that the pseudo-random number generator is unchanged (see [1]).

We recommend using `Make` for dependency tracking because it is language independent. The following hypothetical example illustrates the utility of `Make` and a suitable `Makefile`. Consider a research project that contains a script to simulate data (`simulate.R`) and a scientific report of the simulation results written in `R Markdown` (`manuscript.Rmd`). A `Makefile` for this project could look like this:

```
1 manuscript.pdf: manuscript.Rmd simulated_data.csv
2 Rscript -e 'rmarkdown::render("manuscript.Rmd")'
3
4 simulated_data.csv: simulate.R
5 Rscript -e 'source("simulate.R")'
```

There are two targets, the final rendered report (`manuscript.pdf`, l. 1) and the simulation results (`simulation_results.csv`, l. 4). Each target is followed by a colon and a list of requirements. If a requirement is newer than the target, the recipe will be executed to rebuild the target. If a requirement does not exist, `Make` uses a recipe to build the requirement before building the target. Here, if one were to build the final `manuscript.pdf` by rendering the `R Markdown` with the command shown in l. 2, `Make` would check whether the file `simulation_results.csv` exists; if not, it would issue the command in l. 5 to run the simulation before rendering the manuscript. This ensures that the simulated data are present before the manuscript is built, and that the simulation is re-run and the manuscript is rebuilt if the simulation code was changed. `Make` therefore offers a standardized process to reproduce projects, regardless of the complexity or configuration of the project. Note that the Workflow for Open Reproducible Code in Science (WORCS) we presented elsewhere [4] does not explicitly contain this dependency tracking element, but its strict structure of only containing one definite `R Markdown` still makes dependencies between files unambiguous.

A version-controlled dynamic document with dependency tracking still relies on external software. Troubleshooting issues specific to a particular programming language or dependent tool typically requires considerable expertise and threatens reproducibility. *Software management* refers to the act of providing records of, or access to, all software packages and system libraries a project depends on. One comprehensive approach to software management is containerization. The central idea is that by "[..] packaging the key elements of the computational environment needed to run the desired software [makes] the software much easier to use, and the results easier to reproduce [. . .]" ([18] p. 174).

`Docker` is a popular tool for containerization. It manages software dependencies by constructing a virtual software environment independent of the host software environment. These so-called "`Docker` images" function like a virtual computer (i.e., a "sand box" a computational environment seperated from the host). A `Docker` image contains *all* software dependencies used in an analysis—not just R packages, but also R and Rstudio, and even the operating system. This is important because low level functionality may impact the workings of higher-order software like R, such as calls to random number generators or linear algebra libraries. All of the differences in computational results that could be caused by variations in the software used are hence eliminated.

Note that the software environment of the Docker image is completely separate from the software installed on your computer. This separation is excellent for reproducibility but takes some getting used to. For example, it is important to realize that software available on your local computer will *not* be accessible within the confines of the `Docker` image. Each dependency that you want to use within the `Docker` image must be explicitly added as a dependency. Furthermore, using `Docker` may require you to install software on an operating system that may not be familiar to you. The images supplied by the rocker project [19], for example, are based on Linux.

There are two ways to build a `Docker` image. First, users can manually install whatever software they like from within the virtual environment. Such a manually build environment can still be ported to all computers that support `Docker`. However, we prefer the second way of building images automatically from a textual description called `Dockerfile`. Because the `Dockerfile` clearly describes how which software is installed, the installation process can be repeated automatically. Users can therefore quickly change the software environment, for example, update to another R version or given package version. Packaging all required software in such an image requires considerable amounts of storage space. Two major strategies help to keep the storage requirements reasonable. One is to rely on pre-made images that are maintained by a community for particular purposes. For example, there are pre-made images that only include what is necessary for R, based on Ubuntu containers [19]. Users can then install whatever they need in addition to what is provided by these pre-compiled images. The image that was used for this article uses 1.35 GiB of disk space. The image for this project includes Ubuntu, R, RStudio, LaTeX as well as a variety of R packages like tidyverse [20] and all its dependent packages, amounting to 192 R packages.

A second strategy is to save a so-called `Dockerfile`, which contains only a textual description of all commands that need to be executed to recreate the software environment. `Dockerfiles` are tiny (the `Dockerfile` (https://github.com/aaronpeikert/repro-tutorial/blob/main/Dockerfile accessed on 11 October 2021) for this project has a size of only 1.55 KiB). However, they rely on the assumption that all software repositories that provide the dependent operating systems, pieces of software, and R packages will continue to remain accessible and provide historic software versions. For proper archiving, we therefore recommend storing a complete image of the software environment, in addition to the `Dockerfile`. A more comprehensive overview of the use of containerization in research projects is given by Wiebels and Moreau [21]. Note that WORCS, which we presented elsewhere [4] relies on the R package `renv` [22] for software management. Although `renv` is more lightweight and easier to use than `Docker`, it is not as comprehensive because it only takes snapshots of the R packages instead of all software used.

To summarize, the workflow by Peikert and Brandmaier [1] requires four components (see Figure 1) dynamic document generation (using `R Markdown`), version control (using `Git`), dependency tracking (using `Make`), and software management (using `Docker`). While

R Markdown and Git are well integrated into the R environment through RStudio, Make and Docker require a level of expertise that is often beyond the training of scholars outside the field of information technology. This presents a considerable obstacle to the acceptance and implementation of the workflow. To overcome this obstacle, we have developed the R package repro that supports scholars in setting up, maintaining, and reproducing research projects in R. Importantly, a reproducible research project created with repro does not have the repro package itself as a dependency. These projects will remain reproducible irrespective of whether repro remains accessible in future. Users do not need to have repro installed to reproduce a project; in fact, they do not even need to have R installed because the entire project can be rebuilt inside a container with R installed. In the remainder, we will walk you through the creation of a reproducible research project with the package repro.

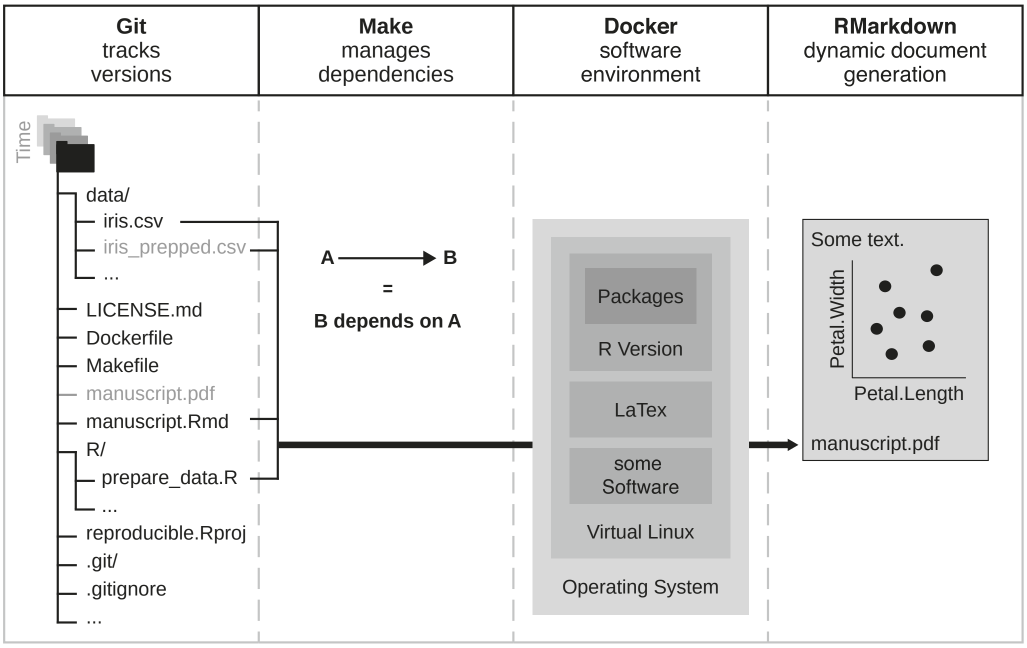


Figure 1. Schematic illustration of the interplay of the four components (in dashed columns) central to the reproducible workflow: version control (Git), dependency tracking (Make), software management (Docker), and dynamic document generation (R Markdown). Git tracks changes to the project over time. Make manages dependencies among the files. Docker provides a container in which the final report is built using dynamic document generation in R Markdown. Adapted from Peikert and Brandmaier [1] licensed under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0 accessed on 4 December 2021).

3. Creating Reproducible Research Projects

One impediment to the widespread adoption of a standard for reproducible research is that learning to use the required tools can be quite time-intensive. To lower the threshold, the R package repro introduces helper functions that simplify the use of complicated and powerful tools. The repro package follows the format of theusethis (https://usethis.r-lib.org) [23] package, which provides helper functions to simplify the development of R packages. The repro package provides similar helper functions, but focuses on reproducibility-specific utilities. These helper functions guide end-users in the use of reproducibility tools, provide feedback about what the computer is doing and suggest what the user should do next. We hope this makes reproducibility tools more accessible by enabling beginner-level users to detect their system's state accurately and act correspondingly ([24] Chapter 8: "Automation and Situation Awareness"). These wrappers are merely a support system; as users learn to use the underlying tools, they can rely less on repro and use these tools directly to solve more complex problems.

This tutorial assumes that the user will be working predominantly in R with the help of RStudio. It describes basic steps that we expect to be relevant for small-scale psychological research projects that do not rely on external software or multistage data processing (for those requirements see Section 4). Of course, your specific situation might involve additional, more specialized steps. After completing the tutorial, you should be able to customize your workflow accordingly.

The first step is to install the required software. We assume that you have installed R ([7] version 4.0.4) and RStudio ([25] version 1.4) already but the tutorial will guide you in detail through the installation of other necessary software with the help of the R package `repro` (https://github.com/aaronpeikert/repro accessed on 4 December 2021) [26]. In case you have either not installed R and RStudio or are unsure if they are up-to-date, you might want to consult our installation advice in the Online Supplementary Material (https://github.com/aaronpeikert/repro-tutorial/blob/main/install.md accessed on 4 December 2021) that covers the installation of all software necessary for this tutorial in three steps. The installation advice (https://github.com/aaronpeikert/repro-tutorial/blob/main/install.md accessed on 11 October 2021) may also help Windows users who have problems installing `Docker`.

Unfortunately, Docker requires administrator rights to run, which may not be available to all researchers. We recommend `renv` [22] in cases where no administrator rights can be obtained but can not detail its use in this document. `renv` tracks which R package is installed from which source in which version in a so-called `lockfile`. This lockfile is then used to reinstall the same packages on other computers or later in time. For a more thorough discussion, see Lissa et al. [4].

Start RStudio and install the package `repro`[26]. It will assist you while you follow the tutorial.

```r
# repro is not on CRAN yet
options(
repos = c(aaronpeikert = 'https://aaronpeikert.r-universe.dev',
CRAN = 'https://cloud.r-project.org')
)
install.packages('repro')
```

To verify that you have indeed installed and set up the required software for this workflow, you can use the “check functions”. These also illlustrate how `repro` assists the user in setting up a reproducible workflow. In the example below, we use the double-colon operator to explicitly indicate which functions originate in the `repro` package. If the package is loaded (using `library("repro")`), it is not necessary to use this double-colon notation.

```r
# `package::function()` → use function from package without `library(package)`
repro::check_git()
```

```
## v Git is installed, don’t worry.
```

```r
repro::check_make()
```

```
## v Make is installed, don’t worry.
```

```r
repro::check_docker()
```

```
## v Docker is installed, don’t worry.
```

These functions check whether specific dependencies are available on the user’s system, and if not, explain what further action is needed to obtain it. Sometimes they ask the user to take action; for example, the following happens if you are a Windows user who does not have Git installed:

```
repro::check_git()
```

```
## x Git is not installed.

## i We recommend Chocolately for Windows users.

## x Chocolately is not installed.

## * To install it, follow directions on:
##   'https://chocolatey.org/docs/installation'

## i Use an administrator terminal to install chocolately.

## * Restart your computer.

## * Run `choco install -y git` in an admin terminal to install Git.
```

The messages from repro try to help the user solve problems. They are adjusted to your specific operating system and installed dependencies. Before you continue, we ask you to run the above commands to check Git, Make, and Docker—both to become familiar with the functionality of the check_*() functions and to make sure your system is prepared for the remainder of this tutorial.

After you have installed the necessary software, we suggest that you set up a secure connection to GitHub:

```
repro::check_github()
```

```
## v You and GitHub are on good terms, don't worry.
```

If you know what Secure Shell (SSH) is and want to use it, you may alternatively use:

```
# only an alternative: DO NOT USE if you are unsure what SSH means
repro::check_github(auth_method = "ssh")
```

```
## v You and GitHub are on good terms, don't worry.
```

If necessary, follow any instructions presented until all checks are passed.

3.1. Creating an RStudio Project

We start by creating a project folder with RStudio by clicking the menu item:

File → New Project… → New Directory → Example Repro Template

This creates a project with a sample analysis. This sample analysis consists of a single R Markdown document and a single data file. The only special thing about the R Markdown document is the repro metadata that we will learn about later. However, you may turn any other template or existing R project into a reproducible research project by adding those repro metadata there.

3.2. Implementing Version Control

Now that your project is set up, we will introduce you to version control with Git. Git does not automatically track all files in your project folder; rather, you must manually add files to the Git repository. To make sure you do not accidentally add files that you do not wish to share (e.g., privacy-sensitive data), you can list specific files that you do not want to track in the .gitignore file. You can also block specific filetypes; for example, to prevent accidentally sharing raw data. You can add something to the .gitignore file directly or with this command:

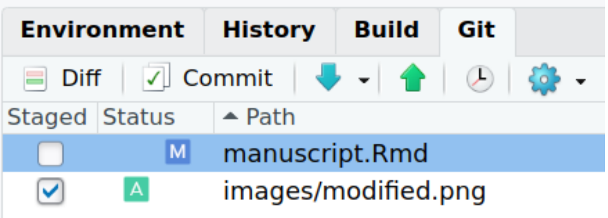
```
usethis::use_git_ignore("private.md")
```

Now the file `private.md` will not be added to the `Git` repository, and hence also not be made public if you push the repository to a remote service like GitHub. Please also consider carefully whether you can include data in the repository without violating privacy rights. If you are not allowed to share your data publicly, add the data file(s) to the `.gitignore` file and only share them on request.

New users are advised to explicitly exclude any sensitive files before proceeding. When you are ready, you can begin tracking your remaining files using `Git` by running:

```
usethis::use_git()
```

For `Git` to recognize changes to a given file, you have to stage and then commit these changes (this is the basic save action for a project snapshot). One way to do this is through the visual user interface in the RStudio `Git` pane (see Figure 2). Click on the empty box next to the file you want to stage. A checkmark then indicates that the file is staged. After you have staged all of the files you want, click on the commit button, explain in the commit message why you made those changes, and then click on commit. This stores a snapshot of the current state of the project.



Staged	Status	Path
□	M	manuscript.Rmd
☑	A	images/modified.png

Figure 2. The Git pane in R Studio, showing `manuscript.Rmd` modified but unstaged and `modified.png` newly added and staged.

The files you created and the changes you made have not yet left your computer. All snapshots are stored in a local repository in your project folder. To back up and/or share the files online, you can push your local repository to a remote repository. While you can choose any `Git` service (like GitLab or BitBucket), we will use GitHub in this tutorial. Before you upload your project to GitHub, you need to decide whether you would like the project to be publicly accessible (viewable by anyone, editable by selected collaborators) or if you want to keep it private (only viewable and editable by selected collaborators). To upload the project publicly to GitHub use:

```
usethis::use_github()
```

To upload it privately:

```
usethis::use_github(private = TRUE)
```

Depending on your computer's configuration, it may ask you to set up a secure connection to GitHub. In this case, first, follow the suggestions shown on the R console.

3.3. Using Dynamic Document Generation

Now that you have created a version-controlled project, we will proceed with dynamic document generation. A dynamic document has three elements:

1. Text (prose; e.g., a scientific paper or presentation)
2. Executable code (e.g., analyses)
3. Metadata (e.g., title, authors, document format)

`R Markdown` is a type of dynamic document well-suited to the RStudio user interface. The text of an `R Markdown` is formatted by `Markdown` (see [27] for technical details and [17] for practical guidance). The code mostly consists of R code (although other programming languages are supported, like Python, C++, Fortran, etc). The following example serves to illustrate the `Markdown` syntax. It shows how to create a heading, a word in bold font, a citation, and a list of several items in `Markdown`:

```
<!--this is a Markdown file -->
# Heading (level 1)

Normal text.
Important **word** in bold.
A citation: @einstein1935 did important research on this topic.

## Subheading (level 2)

To do list:

* Do research
* Do more research
* Spend time with family and friends
```

One advantage of this type of markup for formatting is that it can be rendered to many different output formats—both in terms of file types, like `.docx`, `.html`, `.pdf`, and in terms of style, e.g., specific journal requirements. For social scientists, the `papaja` package [28] may be relevant, as it produces manuscripts that follow the American Psychological Association formatting requirements [29]. `R Markdown` files are plain text, which is more suitable for version control using `Git` than binary files generated by some word processors. Some users might find it easier to activate the "Visual Editor" of RStudio ([Ctrl] + [Shift] + [F4] or click on the icon that resembles drawing materials or a compass in the upper right corner of the `R Markdown` document), which features more graphical elements like a traditional word processor but still creates an `R Markdown` underneath with all of its flexibility. The visual editor has some additional benefits, such as promoting best practices (for example, each sentence should be written on a new line, which makes it easier to track changes across versions) and improving the generation of citations and references to tables and figures.

Now that you are familiar with `Markdown` formatting basics, we turn our attention to including code and its results in the text. Code is separated by three backticks (and the programming language in curly brackets) like this:

~~~
This is normal text, written in Markdown.

```{r}
# this is R code
1 + 1
```
~~~

The hotkey [Control] + [Alt] + [i] inserts a block of code in the file. The results of code enclosed in such backticks will be dynamically inserted into the document (depending on specific settings). This means that whenever you render the `R Markdown` to its intended output format, the code will be executed and the results updated. The resulting output document will be static, e.g., a pdf document, and can be shared wherever you like, e.g., on a preprint server.

Once the `R Markdown` file has been rendered to a static document (the output, e.g., PDF), the resulting file is decoupled from the `R Markdown` and the code that created it. This introduces a risk that multiple versions of the static document are disseminated, each with slightly different results. To avoid ambiguity, we, therefore, recommend referencing the identifier of the `Git` commit at the time of rendering in the static document. Simply put, a static document should link to the version of the code that was used to create it. The repro package comes with the function repro::current_hash() for this purpose. This document was created from the commit with the hash a8bb0d4 (view on GitHub).

Now that you know how to write text and R code in an R Markdown, you need to know about metadata (also called: YAML front matter). These metadata contain information about the document, like the title and the output format. Metadata are placed at the beginning of the document and are separated from the document body by three dashes. The following example is a full markdown document where the metadata (the "YAML front matter") are in lines 1–6. Some metadata fields are self-explanatory (like the author field), and exist across all output formats (like the title field). Others are specific to certain output formats or R packages.

```
1  ---
2  title: "A Tutorial on how to Do the Same Thing More Than Once"
3  author: Aaron Peikert, Caspar J. van Lissa, and Andreas M. Brandmaier
4  abstract: A hitchhiker's guide to reproducible research in R
5  output: html_document
6  ---
7  
8  # Introduction
9  
10 Important for reproducibility:
11 
12 1. *Version control*
13 2. *Dynamic document creation*
14 3. *Dependency tracking*
15 4. *Software management*
16 
17 ```{r}
18 # this is R code
19 t.test(extra ~ group, data = sleep)
20 ```
```

3.4. Manage Software and File Dependencies

The repro package adds fields to the metadata to list all dependencies of the research project. This includes R scripts, data files, and external packages. The format is as follows (see everything below the line *repro:*):

```
---
title: "A tutorial on how to do the same thing more than once"
author: Aaron Peikert, Caspar J. Van Lissa and Andreas M. Brandmaier
output: html_document
repro:
  scripts:
    - R/load.R
  data:
    - data/mtcars.csv
  packages:
    - tidyverse
    - usethis
    - gert
---
```

This information clarifies what dependencies (in the form of files and R packages) a project relies on. repro uses this information to construct a Makefile for the dependencies on other files and a Dockerfile that includes all required packages. Together, these two files form the basis for consistency within a research project and consistency across different systems. The function repro::automate() converts the metadata from all R Markdown files in the project (all files with the ending .Rmd) to a Makefile and a Dockerfile. These files allow users (including your future self) to reproduce every step in the analysis automatically. Please run repro::automate() in your project:

```
repro::automate()
```

It is important to re-run repro::automate() whenever you change the repro metadata, change the output format, or add a new R Markdown file to the project to keep the Makefile and Dockerfile up to date. There is no harm in running it too often. Other than the Makefile and the Dockerfile, which are created in the document root path, repro generates a few more files in the .repro directory (which we will explain in detail later), all of which you should add and commit to Git.

3.5. Reproducing a Project

If someone (including you) wants to reproduce your project, they first have to install the required software, that is Make, and Docker. Remember, you can use the check_*-functions to test if these are installed:

```
repro::check_make()
```

```
## v Make is installed, don't worry.
```

```
repro::check_docker()
```

```
## v Docker is installed, don't worry.
```

When these are set up, they can ask repro to explain how they should use Make and Docker to reproduce the project (or you could explain it to them):

```
repro::reproduce()
```

```
## * To reproduce this project, run the following code in a terminal:

##   make docker &&
##   make -B DOCKER=TRUE
```

If you feel uncomfortable using the terminal directly, you can send the command to the terminal from within R:

```
system(repro::reproduce())
```

The only "hard" software requirement for reproducing a project is Docker, assuming users know how to build a Docker image and run Make within the container. However, if they have installed Make in addition to Docker, they do not even need to know how to use Docker and can simply rely on the two Make commands "make docker" and "make -B DOCKER=TRUE".

3.6. Summary

1. Install the repro package:

```
options(
repos = c(aaronpeikert = 'https://aaronpeikert.r-universe.dev',
CRAN = 'https://cloud.r-project.org')
)
install.packages('repro')
```

2. Check the required software:

```r
repro::check_git()
```

```
## v Git is installed, don't worry.
```

```r
repro::check_github()
```

```
## v You and GitHub are on good terms, don't worry.
```

```r
repro::check_make()
```

```
## v Make is installed, don't worry.
```

```r
repro::check_docker()
```

```
## v You are inside a Docker container!
```

3. Create an R project or use an existing one. Do not forget to add `repro` metadata (i.e., packages, scripts, data).

```yaml
repro:
  scripts:
    - R/load.R
  data:
    - data/mtcars.csv
  packages:
    - tidyverse
```

The sample `repro` project already has theese metadata:

```r
repro::use_repro_template("/some/folder")
```

4. Let `repro` generate `Docker`- and `Makefile`:

```r
repro::automate()
```

5. Enjoy automated reproducibility:

```r
repro::reproduce()
```

```
## * To reproduce this project, run the following code in a terminal:

##   make docker &&
##   make -B DOCKER=TRUE
```

4. Advanced Features

This section is for advanced users who want to overcome some limitations of `repro`. If you read this paper the first time, you will probably want to skip this section and continue reading from the section "Preregistration as Code." As explained above, `repro` is merely a simplified interface to the tools that enable reproducibility. This simplified interface imposes two restrictions. Users who ask themselves either, "How can I install software dependencies outside of R in the `Docker` image?" or "How can I express complex dependencies between files (e.g., hundreds of data files are preprocessed and combined)?" need to be aware of these restrictions and require a deeper understanding of the inner workings of `repro`. Other users may safely skip this section or return to it if they encounter such challenges.

The first restriction is that users must rely on software that is either already provided by the base `Dockerimage` "rocker/verse" or the R packages they list in the metadata. The metadata the `repro::automate()` function relies on can only express R packages as dependencies for the `Dockerfile` and only trivial dependencies (in the form of "file must exist") for the `Makefile`. Other software that users might need, like other programming languages, not yet installed LaTeX packages, etc., must be added manually. We plan to add support for commonly used ways to install software beyond R packages via the metadata and `repro::automate()`, for example, for system libraries (via `apt` the Ubuntu package manager), LaTeX packages (via `tlmgr` the Tex Live package manager), Python packages (via `pip` the python package manager). The second limitation is related to dependencies. `Make` can represent complex dependencies, for example: A depends on B, which in turn depends on C and D. If B is missing in this example, `Make` would know how to recreate it from C and D. These dependencies, and how they should be resolved, are difficult to represent in the metadata. Users, therefore, have to either "flatten" the dependency structure by simply stating that A depends on B, C, and D, thereby leaving out important information or express the dependencies directly within the `Makefile`.

The following section explains how to overcome these limitations despite reliance on the automation afforded by `repro`. Lifting these restrictions requires the user to interact more directly with `Make` or `Docker`. Users need to understand how repro utilizes `Make` and `Docker` internally to satisfy more complicated requirements.

Let us have a closer look at the command for reproducing a repro project: `make docker && make -B DOCKER=TRUE`; which consists of two processing steps. First, it recreates the virtual software environment (`Docker`), and then it executes computational recipes in the virtual software environment (`Make`). The first step is done by the command `make docker`. The command `make docker` will trigger `Make` to build the target called `docker`. The recipe for this target builds an image from the `Dockerfile` in the repository. The `&&` concatenates both commands and only runs the second command if the first is successful. Therefore, the computational steps are only executed when the software environment is set up. The second step executes the actual reproduction and is again a call to `Make` in the form of `make -B DOCKER=TRUE` with three noteworthy parts. First, a call to `make` without any explicit target will build the `Make` target `all`. Second, the flag `-B` means that `Make` will consider all dependencies as outdated and will hence rebuild everything. Third, repro constructs `Make` targets so that if you supply `DOCKER=TRUE` they are executed within the `Docker` image of the project.

The interplay between `Docker` and `Make` resembles a chicken or egg problem. We have computational steps (`Make`) that depend on the software environment (`Docker`) for which we again have computational steps that create it. Users only require a deeper understanding of this interdependence when they either want to have more complex computational recipes than rendering an `R Markdown` or require software other than R packages.

Users can have full control over the software installed within the image of the project. `repro` creates three `Dockerfiles` inside the `.repro` directory. Two `Dockerfiles` are automatically generated. The first is `.repro/Dockerfile_base`. It contains information about the base image on which all the remaining software is installed. By default we rely on the "verse" images provided by the Rocker project [19]. These contain (among other software) the packages `tidyverse`, `rmarkdown`, and a complete LaTeX installation, which makes these images ideal for the creation of scientific manuscripts. Users can choose which R version they want to have inside the container by changing the version number in line 1 to the desired R version number. By default, the R version corresponds to the locally installed version on which `repro::automate()` was called the first time. The build date is used to install packages in the version that was available on the Comprehensive R Archive Network on this specific date and can also be changed. By default, this date is set to the date on which `repro::automate()` was called the first time. This way, the call to the automate function virtually freezes the R environment to the state it was called the first time inside the container. Below, you see the Docker base file we used to create this manuscript:

```
FROM rocker/verse:4.0.4
ARG BUILD_DATE=2021-05-06
WORKDIR /home/rstudio
```

The second automatically generated Dockerfile is .repro/Dockerfile_packages. Whenever repro::automate() is called, repro gathers all R packages from all .Rmd files and determines whether they should be installed from CRAN or GitHub fixed to the date specified in Dockerfile_base. Finally, there is one manually edited Dockerfile: .repro/Dockerfile_manual. It is blank by default and can be used to add further dependencies outside of R, like system libraries or external software. Using Docker may require you to install software on an operating system that may not be familiar to you. The images supplied by [19], for example, are based on the Ubuntu operating system. The most convenient way to install software on Ubuntu is through its package manager apt. If the following snippet is added to .repro/Dockerfile_manual, the Docker image will have, for example, Python installed. Other software is installed identically, only the software name is exchanged.

```
RUN apt-get update && apt-get install -y python3
```

Docker eventually requires a single Dockerfile to run, so repro::automate() simply concatenates the three Dockerfiles and saves the result into the main Dockerfile at the top level of the R project. With this approach, users of repro can build complex software environments and implement complex file dependencies. The standard repro metadata only make sure that all dependencies are available but does not allow you to specify custom recipes for them in the metadata. If you can formulate the creation of dependencies in terms of computational steps, e.g., the file data/clean.csv is created from data/raw.csv by script R/preprocess.R, you should include these in the Makefile. The Makefile that repro creates is only a template, and you are free to change it. However, make sure you never remove the following two lines:

```
include .repro/Makefile_Rmds
include .repro/Makefile_Docker
```

The file .repro/Makefile_Rmds contains the automatically generated targets from repro::automate() for the R Markdown files. This file should not be altered manually. If you are not satisfied with the automatically generated target, simply provide an alternative target in the main Makefile. Targets in the main Makefile take precedent.

The file .repro/Makefile_Docker does again contain a rather complicated template that you could, but should usually not modify. This Makefile coordinates the interplay between Make and Docker and contains targets for building (with make docker) and saving (with make save-docker) the Docker image. Additionally, it provides facilities to execute commands within the container. If you write a computational recipe for a target, it will be evaluated using the locally installed software by default. To evaluate commands inside the Docker image instead, you should wrap them in $(RUN1) command $(RUN2), as done in this example, which is identical to the first Make example we gave above except for the addition of $(RUN1) and $(RUN2) in l. 2:

```
1 simulated_data.csv: R/simulate.R
2 $(RUN1) Rscript -e 'source("R/simulate.R")' $(RUN2)
```

If users execute this in the terminal:

```
make data/simulation_results.csv
```

It behaves exactly as in the first Make example, the script R/simulate.R is run using the locally installed R. Because this translates simply to:

```
Rscript -e 'source("R/simulation.R")'
```

But if users use

```
make DOCKER=TRUE data/simulation_results.csv
```

It is evaluated within the `Docker` container using the software within it and not the locally installed R version:

```
docker run --rm --user 1000 -v "/home/rstudio":"/home/rstudio/"
reprotutorial Rscript -e 'source("R/simulate.R")'
```

To summarize, `repro` automates dependency tracking (in the form of `Make`) and software management (using `Docker`) without the necessity to learn both tools, but users with advanced requirements can still customize all aspects of both programs.

5. Preregistration as Code

Preregistration refers to the practice of defining research questions and planning data analysis before observing the research outcomes [5]. It serves to separate a-priori planned and theory-driven (confirmatory) analyses from unplanned and post-hoc (exploratory) analyses. Researchers are faced with a myriad of choices in designing, executing, and analyzing a study, often called researchers degrees of freedom. Undisclosed researcher degrees may be used to modify planned analyses until a key finding reaches statistical significance or to inflate effect size estimates, a phenomenon referred to as *opportunistic bias* [30]. Preregistration increases transparency by clarifying when and how researchers employ their degrees of freedom. It expressly does not restrict what researchers may do to gather or analyze their data.

There are still several shortcomings to preregistration. One is that written study plans are often interpretable in multiple ways. Empirical research has shown that, even when several researchers describe their analysis with the same terms, use the same data, and investigate the same hypothesis, their results vary considerably [31]. The current best practice to ensure comprehensive and specific preregistration is to impose structure by following preregistration templates [32,33]. However, such templates cannot ensure full transparency because it is impossible to verbally describe every detail of an analysis for any but the most straightforward analysis. This ambiguity causes a second problem, namely, comparing the initial plan and the resulting publication to decide if and how researchers deviated from the preregistration. This task is difficult because it is impossible to decide without additional information weather the analysis was actually carried out differently or just described differently. Even if researchers were faithful to the preregistration, readers may reach opposite conclusions because they have to compare two different text that may be worded differently or describe the same thing in varying levels of detail. A third limitation is that preregistrations are susceptible to non-reproducibility, just like primary research. To illustrate, a review of 210 preregistrations found that, even though 174 (67%) included a formal power analysis, only 34 (20%) of these could be reproduced [34]. Even when researchers have gone to great lengths in preregistering an analysis script, they sometimes inexplicably fail to reproduce their own results. For example, Steegen et al. [35] realized after publication that part of their preregistered code resulted in different test statistics than they reported initially (see their Footnote 7). A final limitation is that written plans may turn out to be unfeasible once data are obtained and analyzed. For example, a verbal description of a statistical model may be unidentified, e.g., if it includes reciprocal paths between variables or more parameters than observed data. Conversely, a model may be misspecified in a major way; for example, by omitting direct effects when the research question is about mediation, thus leading to a model with an unacceptable fit. Many researchers would only realize that such a model cannot be estimated once the data are obtained, thus necessitating a deviation from the preregistered plans.

The workflow described in this paper facilitates a rigorous solution to this problem: Instead of describing the analysis in prose, researchers include the code required to conduct the analysis in the preregistration. We term this approach of writing and publishing code at the preregistration stage *Preregistration as Code (PAC)*. PAC has the potential to eliminate undisclosed researchers degrees of freedom to a much greater extent than, e.g., preregistration templates. Moreover, it reduces overhead by removing the need to write a separate preregistration and manuscript. For PAC, researchers can write a reproducible, dynamically generated draft of their intended manuscript at the preregistration stage. This already includes most of the typical sections, such as introduction, methods, and results. These results are initially based on simulated data with the same structure as the data the authors expect to obtain from their experiments. For guidance on how to simulate data, see Morris et al. [36], Paxton et al. [37], and Skrondal [38], as well as the R packages `simstudy` [39] and `psych` [40].

Once the preregistration is submitted and real data have been collected or made available, the document can be reproduced with a single command, thus updating the Results section to the final version. Reproducibility is of utmost importance at this stage since the preregistration must produce valid results at two points in time, once before data collection and once after data collection. As outlined before, reproducibility builds upon four pillars (version control, dynamic document generation, dependency tracking, and software management). To use PAC, the dangers to reproducibility we described must be eliminated.

The idea of submitting code as part of a preregistration is not new (e.g., [41]). A prominent preregistration platform, The Open Science Framework, suggests submitting scripts alongside the preregistration of methods. In an informal literature search (we skimmed the first 300 results of Google Scholar with the keywords `("pre registration"|"pre-registration"|preregistration)&(code|script|matlab| python|"R")`) we only found close to a dozen published articles that did include some form of script as part of their preregistration. Though the notion of preregistering code has been around for a while (cf. [35]), it has not gained much traction—perhaps because, to date, this has constituted an extra non-standard step in the research process. This tutorial integrates the preregistration of code into the reproducible research workflow by encouraging researchers to preregister the whole manuscript as a dynamic document.

5.1. Advantages of PAC over Traditional Preregistration

We believe that pairing PAC with the workflow presented above offers five advantages over classical preregistration. First, PAC is merely an intermediate stage of the final manuscript, thus sparing authors from writing, and editors and reviewers from evaluating, two separate documents. Relatedly, writing the preregistration in the form of a research article has the advantage that researchers are usually familiar with this format. By contrast, a preregistration template is a novelty for many. Second, PAC is a tool for study planning. A study can be carried out more efficiently if all steps are documented clearly than when every step is planned ad hoc. Third, PAC removes ambiguity regarding the translation of verbal analysis plans into code. PAC is more comprehensive by design because its completeness can be empirically checked with simulated data. Evaluating the intended analysis code on simulated data will help identify missing steps or ambiguous decisions. PAC, therefore, minimizes undisclosed researchers degrees of freedom more effectively than standard preregistration does [33,41]. Fourth, despite its rigor, PAC accommodates data-dependent decisions if these can be formulated as code. Researchers can, for example, formulate conditions (e.g., in the form of if-else-blocks) under which they prefer one analysis type over the other. For example, if distributional assumptions are not met, the code may branch out to employ robust methods; or an analysis may perform automated variable selection mechanisms before running the final model. Another example of data-dependent decisions are more explorative analyses, i.e., explorative factor analysis or machine learning. Decisions that do not lend themselves to formulation in code, e.g., visual inspection, must still be described verbally or be treated as noted in the next section. Fifth, deviations from the preregistration are clearly documented because they are reflected in changes to the code, which are managed and tracked with version control.

5.2. Deviating from the Preregistration and Exploration

We would like to note that PAC allows explicit comparison of the preregistration and the final publication. Authors should retrospectively summarize and justify any changes made to the preregistered plan, e.g., in the discussion of the final manuscript (In section [Preregistration as Code—a Tutorial] we conducted an actual PAC and summarize the changes we make to the preregistered code in the discussion). During the analysis process, authors can additionally maintain a running changelog to explain changes in detail as they arise. Each entry in the changelog should explain the reasoning behind the changes and link to the commit id that applied the changes. This enables readers and reviewers to inspect individual changes and make an informed judgment about their validity and implications.

Deviations from the preregistration are sometimes maligned, as if encountering unexpected challenges invalidates a carefully crafted study [42]. However, we share the common view that deviation from a preregistration is not a problem [43]; rather, a failure to disclose such deviations is a problem. In fact, it is expected that most PACs will require some modification after empirical data becomes available. Often, deviations provide an opportunity to learn from the unexpected.

For example, imagine that authors preregistered their intention to include both "working memory" and "fluid intelligence" as covariates in an experimental study, examining the effect of task novelty on reaction time. When evaluating the planned analyses on the empirical data, these two covariates reveal high collinearity, thus compromising statistical inference. The authors decide to use PCA to extract common variance related to "intelligence", and include this component as a covariate instead. This change pertains to an auxiliary assumption (that working memory and fluid intelligence are distinct constructs), but does not undermine the core theory (that task novelty affects reaction time). Now imagine that a different researcher is interested in the structure of intelligence. This change to the preregistration directly relates to their theory of intelligence. That researcher might thus interpret the same result as an explorative finding, suggesting that these aspects of intelligence are unidimensional. A deviation from preregistration thus requires a judgement about what changes affect the test of the theory to what extend [44]. Only transparent reporting enables such judgment.

Another common misunderstanding is that preregistration, including PAC, precludes exploratory analyses. We differentiate between two kinds of exploration, neither of which is limited by PAC. The first, more traditional kind of exploration involves ad hoc statistical decisions and post hoc explanations of the results. Such traditional exploratory findings should be explicitly declared in the manuscript to distinguish them from confirmatory findings [5,43]. The second kind of exploration is through procedurally well defined exploration with exploratory statistical models that are standard in machine learning [45]. These models often involve dozens, if not hundreds of predictors, which makes it difficult to describe them verbally. With PAC, such models can be preregistered clearly and in comprehensive detail, and the researcher can precisely define a priori how much they want to explore. We specifically recommend PAC for such exploratory statistical models. The merit of preregistration in these cases is to communicate precisely how much exploration was done; a piece of information that is crucial to assess e.g., whether the results might be overfit ([46], p. 220f.).

5.3. Planned Analyses as Functions

Although researchers may use any form to preregister their planned analyses (e.g., scripts), we suggest writing three functions for each planned hypothesis: one to conduct the planned analysis, one to simulate the expected data, and one to report the results. Using functions makes the analysis more portable (i.e., it can easily be used for other datasets), and facilitates repeated evaluation, as is the case in a simulation study. The functions shown here do not contain executable code, but the interested reader can find working functions in the online Supplementary Materials (https://github.com/aaronpeikert/repro-tutorial/blob/main/R/simulation_funs.R accessed on 11 October 2021) that power the example below.

It is difficult to write analysis code when it is not clear what the expected data will look like. We therefore recommend first simulating a dataset that resembles the expected structure of the empirical data that will be used for the final analysis. Dedicated packages to simulate data for specific analyses exist .

The general format of a simulation function might be as follows:

```r
simulate_data <- function(n, effect_size){
# 1. warn users that the results are "fake"
# 2. draw `n` samples with `effect_size`
# 3. format and return in expected data format
}
```

For linear models, simulating data is extremely simple:

```r
simulate_data <- function(n, effect_size){
warning("This manuscript contains mock results based on simulated data.")
# Draw n samples from a normal distribution for predictor X
  x <- rnorm(n)
# Calculate dependent variable Y..
#.. as a function of population effect size and residual error
  y <- effect_size * x + rnorm(n)
# Return a data.frame
data.frame(x = x, y = y)
}
```

Next, write a function to conduct the planned analysis. This function should receive the data and compute all relevant results from it. The general format of an analysis function might be:

```r
planned_analyis <- function(data){
# 1. preprocess e.g. with `rowMeans(data)`
# 2. conduct analysis e.g. with `t.test()`
# 3. `return(results)`
}
```

In the simplest case, an analysis function might already exist in R. For the linear model above, the analysis function might be:

```r
planned_analyis <- function(data){
lm(y ~ x, data = data)
}
```

As soon as we have written `planned_anaylsis()` and `simulate_data()` we can iteratively improve both functions, e.g., until `planned_analysis()` runs without error and recovers the correct parameters from `simulate_data()`. The goal is to ensure that the output of `simulate_data()` works as input to the function `planned_analysis()`.

When the researchers are satisfied with the function `planned_analysis()`, they can think about the way the would they would like to report the analysis results via tables, plots, and text. The implementation of this reporting should be in the function `report_analysis()`.

```r
report_analysis <- function(results){
# 1. create markdown tables from results
# 2. conditionally interpret results e.g. if(p < .025)"Result is significant."
# (optional) visualize results
# 3. return results section formatted in markdown
}
```

This function should again accept the output of `planned_analysis()` as input. The output of this function should be formatted in Markdown. The idea is to automatically generate the full results section from the analysis. This way, the preregistration not only specifies the computation but also how the its results are reported. Various packages automatically generate well-formatted Markdown outputs of statistical reports or even entire tables of estimates or figures directly from R goal to help with this objective. Packages like `pander` [47], `stargazer` [48], `apaTables` [49] and `papaja` [28] help you to create dynamically generated professional looking results. The package `report` [50] is particularly noteworthy because it not only generates tables but also a straightforward interpretation of the effects as actual prose (e.g., it verbally quantifies the size of an effect).

Ideally, these three functions can be composed to create a "fake" results section, e.g., when composed to `report_analysis(planned_analysis(simulate_data()))` or `simulate_data() %>% planned_analysis() %>% report_analysis()` outputs a results section.

Turning a Dynamic Document into a Preregistration

After researchers are satisfied with their draft preregistration, they should archive a time-stamped and uneditable version of the project that serves as the preregistration. zenodo.org [51] is a publicly funded service provider that archives digital artefacts for research and provides digital object identifiers (DOI) for these archives. While the service is independent of GitHub—in terms of storage facilities and financing—you can link GitHub and zenodo.org. Please note that you can only link public GitHub repositories to zenodo.org. You may log into zenodo.org through your GitHub account. To log in with your GitHub account:

> Navigate to https://zenodo.org/login/ → Log in with GitHub

To link zenodo.org and GitHub

> Log into zenodo.org → Account → GitHub (https://zenodo.org/account/settings/github/)

Or:

> Navigate to https://zenodo.org/account/settings/github/

After you have linked a GitHub repository, you trigger the archival by creating a GitHub release. To create GitHub release, navigate to GitHub:

```r
usethis::browse_github()
```

Then click on Releases → Draft a new release. Here you can add all relevant binary files but at least a rendered version of the manuscript and the Docker image.

To summarize, researchers need to write three functions, `planned_analysis()`, `simulate_data()`, and `report_analysis()` and embed these into a manuscript that serves as a preregistration in an uneditable online repository. After they gathered the actual data, they can replace the simulated data, render the dynamic manuscript (therefore run `planned_analysis()` on the actual data), and write the discussion.

5.4. Alternatives to Simulated Data

Simulating data may prove challenging to applied researchers. In the spirit of team science and collaboration, one feasible solution is to involve a statistical co-author. However, several easy alternatives exist. The downside of these alternatives is that they all rely indirectly on the use of real data. This introduces a risk that the planned analyses may be cross-contaminated by any exploratory findings. It is crucial to disclose any exposure to the data in preparation of the preregistration (PAC or otherwise). This exposure to the data may decrease trust in the objectivity of the preregistration. Moreover, researchers should take rigorous measures to prevent exposure to exploratory findings that may unintentionally influence their decision making.

The simplest method is to collect empirical data first, but set it aside and proceed with a copy of the data that is blinded by randomly shuffling the order of rows for each variable (independently of each other). Shuffling removes any associations between variables, while retaining information about the level of measurement and marginal distribution of each variable. If the hypotheses pertain to associations between variables, this treatment should thus be sufficient to prevent cross-contamination. The researcher can still access the information about means or proportions (e.g., the number of participants belonging to group "A" are in the dataset), but remain uninformed about relations between variables (e.g., members of group "A" have a greater mean in variable "Z"). Preregistration after data collection is common for secondary data analysis of data obtained by other research groups [52] but not so much within the same research project. We argue that it is still an eligible preregistration. Guidelines for clinical trials already recommend analysis of blinded data to test the feasibility of a preregistration [53].

Another alternative to simulated data is to conduct a pilot study [54] and use the pilot data to develop the preregistration. A pilot study has obvious advantages for study planning, since it lets the researcher evaluate the feasibility of many assumptions. However, we must warn our readers that while piloting is more traditional than our approach of blinding the data before preregistration, the data from the pilot study must not enter the analysis data set.

5.5. When Is PAC Applicable?

PAC is applicable to every study that can be preregistered and ultimately uses computer code for the statistical analysis. Two types of preregistrations are particularly amenable to PAC. First, pregistrations of clinical trials (called statistical analysis plans, International Council for Harmonisation of Technical Requirements for Registration of Pharmaceuticals for Human Use [53]) typically describe analyses in exhaustive detail and typically contain a detailed description of how results will be presented, including shells of tables and graphics [55]. PAC may significantly reduce the required workload while maintaining (and exceeding) the required standards for preregistering a clinical trial.

Second, preregistering exploratory statistical models (i.e., those with large numbers of competing models or those inspired by machine learning) is hardly feasible with standard preregistrations since they are too complex to describe and depend strongly on their software implementation. PAC, however, captures the precise algorithmic model, including its software implementation, and is ideal for preregistering these models [45].

5.6. Preregistration as Code: Tutorial

We have argued that PAC has several advantages over classic preregistration and have outlined its implementation. To illustrate how PAC works in practice and to help researchers implement PAC themselves, we provide a worked example. We will use an exemplary research question that was based on openly available data:

> "Is there a mean difference in the personality trait 'Machiavellism' between self-identified females and males?"

Again, we propose a preregistration format that closely resembles a classic journal article but uses simulated data and dynamic document generation to create a document that starts out as a preregistration and eventually becomes the final report. The complete preregistration source is available in the online Supplementary Material (https://github.com/aaronpeikert/repro-tutorial/blob/main/preregistration.Rmd accessed on

11 October 2021). In this section, we show code excerpts of this preregistration (formatted in monospace) and explain the rationale behind them.

As usual, the authors state why they are interested in their research question in the "Introduction" section and provide the necessary background information and literature to understand the context and purpose of the research question. This example is drastically shortened for illustration purposes:

```
# Theoretical Background

Machiavellianism describes a personality dimension characterized by a
cynical disregard of morals in the pursuit of one's own interest, e.g.
through manipulation [@christie1970]. There is extensive literature reporting
differences in the dark triad (narcissism, machiavellianism, and psychopathy)
between self-identified males and females [@muris2017] but only few studies
focus solely on machiavellianism. We aim to replicate the finding that males
tend to have higher machiavellianism scores [@muris2017].
```

After researchers have provided the research question, they typically proceed to explain how they want to study it. For simplicity, we will use already published data that we have not yet analyzed:

```
# Method

We report how we determined our sample size, all data exclusions (if any), all
manipulations, and all measures in the study [cf. @simmons2012]. We use data
available from [openpsychometrics.org](https://openpsychometrics.org/_rawdata/)
from the online version of the MACH-IV[@christie1970] and included participants
that have responded to at least one machiavellianism item and reported their
gender as either "male" or "female".
```

We choose the following statistical procedure because many researchers are familiar with it (The *t*-test and Mann-Whitney-Wilcoxon test are arguably the most often used hypothesis tests (according to [56,57] reports that 26% of all studies employed a *t*-test and 27% employed a rank-based alternative in the *New England Journal of Medicine* in 2005). The analytical strategy presented here is, in fact, suboptimal in several respects (the assumption of measurement invariance is untested [58], the effect size is underestimated in the presence of measurement error [59], the effect size is overestimated for highly skewed distributions [60]). The interested reader can use the provided code for the simulation (https://github.com/aaronpeikert/repro-tutorial/blob/main/R/simulation.R accessed on 4 December 2021) to verify that the *t*-test provides unbiased effect sizes but the Mann-Whitney-Wilcoxon overestimates effect sizes with increasing sample size and skewness):

```
We conduct a Student's t-test [@studentProbableErrorMean1908] with Welch's
correction [@welchGeneralizationStudentProblem1947] of the average of
machiavellianism items between the binary-coded gender groups. If the skew of
this average is greater than 1.0 we conduct a supposedly more robust Mann--
Whitney--Wilcoxon test [@Wilcoxon1945] instead.
```

The methods section is the translation of the following `planned_analysis()` function:

```r
planned_analysis <- function(data, use_rank = "skew", skew_cutoff = 1){
# average over all variable supplied, except gender
  machiavellianism <- rowMeans(data["gender" != names(data)], na.rm = TRUE)
# discard rows that only contain NAs
  data <- data[!is.na(machiavellianism),]
  machiavellianism <- machiavellianism[!is.na(machiavellianism)]
# assure gender is factor
  gender <- as.factor(data$gender)
# note skewness and decide t.test vs wilcox based on it
  skew <- moments::skewness(machiavellianism)
# skewness cutoff
if(use_rank == "skew")use_rank <- abs(skew) > skew_cutoff
if(use_rank){
# t.test + rank = wilcox test
    machiavellianism <- rank(machiavellianism)
  }
  test <- t.test(machiavellianism ~ gender)
# return a bunch of information
list(test = test, skew = skew, use_rank = use_rank, n = length(gender))
}
```

This function illustrates two advantages of PAC. First, a PAC can easily include data-dependent decisions by creating different analysis branches under different conditions. Second, it highlights how difficult it is to describe a statistical analysis precisely. The same verbal descriptions may be implemented differently by different persons depending on their statistical and programming knowledge and assumptions. One example would be using the function `wilcox.test` instead of the combinations of the functions `rank` and `t.test`. Either of them is a valid implementation of the Mann–Whitney–Wilcoxon test, but the first assumes equal variance. In contrast, the second applies Welch's correction by default and hence is robust even with unequal variances across groups [61]. Mentioning every such minute implementation detail is almost impossible and would result in overly verbose preregistrations. Still, these details can make a difference in the interpretation of statistical results and, thus, represent undisclosed researchers' degrees of freedom.

Together with the function `simulate_data()` (not shown here), the function `planned_analysis()` can be used to justify the planned sample size. To that end, `simulate_data()` is repeatedly called with increased sample sizes and the proportion of significant results (power) is recorded. The results for such a Monte Carlo simulation for this example are visualized in Figure 3. The code for this power analysis can be found in the online Supplementary Material (https://github.com/aaronpeikert/repro-tutorial/blob/main/R/simulation.R accessed on 11 October 2021). The next snippet shows how we integrated the results dynamically into the preregistration (the origin of the R-variables `minn`, `choosen_power`, and `choosen_d` is not shown).

```
A simulation we conducted indicated that with a sample size of `r minn` for
an alpha of .05 (two-sided) we achieve at least `r choosen_power*100`% power
assuming a standardized effect size of d=`r choosen_d`.
```

Monte Carlo simulations are, of course, not only applicable for this analysis method and also allow researchers to investigate further relevant properties of their analysis method beyond power [62–64].

We implemented a mechanism that only uses simulated data when the actual data are not yet available (in this example, if the file `data/data.csv` does not exist) for the results section. This mechanism also warns readers if these results are based on simulated data. The warning is colored red to avoid any confusion between mock and actual results. As soon as the actual data are available, the simulated data are no longer used, and the results represent the actual empirical results of the study.

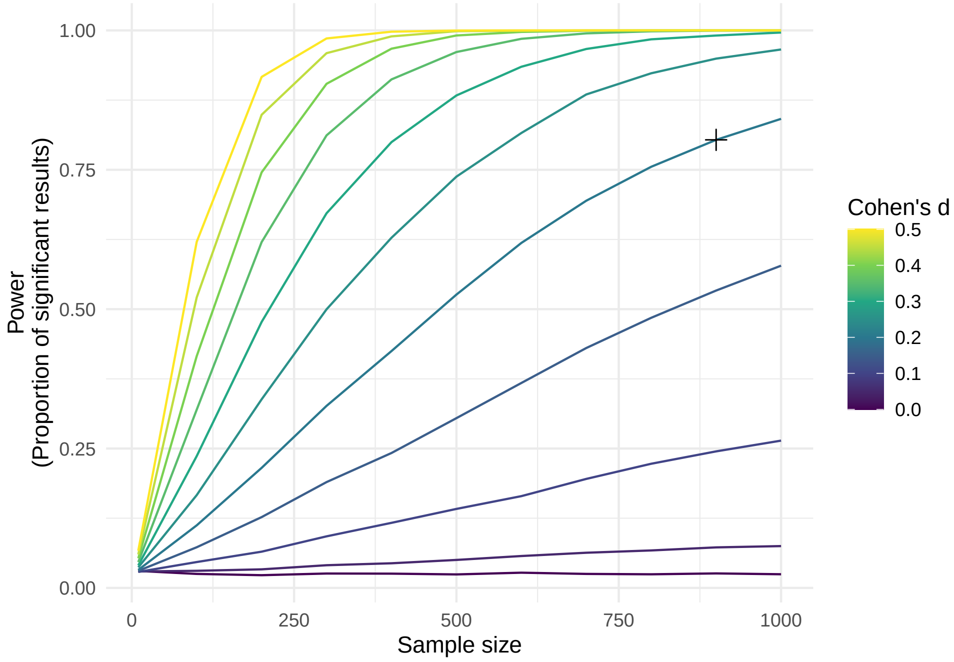


Figure 3. Results of simulation for the power analysis. The cross indicates the sample size that archives 80% assuming a Cohen's d of 0.2.

~~~
# Results

```{r, echo=FALSE, results='asis', warning=FALSE, message=FALSE}
real_data <- here::here("data", "data.csv")
simulated <- !fs::file_exists(real_data)
if(simulated){
  cat("\\textcolor{red}{The results are based on simulated data and must not be
      interpreted. They only serve to illustrate the result of the preregistered
      code.}")
  set.seed(1234)
  mach <- simulate_data(900, 8, 0.3, 10)
} else {
  mach <- readr::read_delim(real_data, delim = "\t", na = c("", "NA", "NULL"))
  # only keep MACH items + gender
  mach <- dplyr::select(mach, dplyr::matches("^Q\\d+A$"), gender)
  # code gender according to codebook (3 would be other)
  mach <-
    dplyr::mutate(mach, gender = factor(
      gender,
      levels = 1:2,
      labels = c("male", "female")
    ))
  # some items are reversed, see https://core.ac.uk/download/pdf/38810542.pdf
  reversed_nr <- c(1, 15, 2, 12, 4, 11, 14, 19)
  reversed <- stringr::str_c("Q", reversed_nr, "A")
  mach <- dplyr::mutate(mach, dplyr::across(one_of(reversed), ~ 6 - .x))
}
```
~~~

Following the recommendations outlined in this paper, we did not access the data when we initially wrote this code. We therefore did not know the exact format the data would have. This means that we did need to change our preregistration after accessing the data to include i.e., the recoding of gender (lines 17–22) and the items (lines 23–26). We invite the reader to evaluate the changes we made to the preregistered code. Either on GitHub (https://github.com/aaronpeikert/repro-tutorial/compare/v0.0.1.1-prereg..main accessed on 11 October 2021) → "Files changed" or directly in Git with `git diff v0.0.1.1-prereg preregistration.Rmd`.) This is our summary of what we changed:

```
# Discussion

This document only serves to illustrate Preregistration as Code. We, therefore,
do not discuss the results. After we have acquired the data, we realized that
we had to change the code for reading the data, including recoding gender,
missing values and reversed items (see commit [6556a93](https://github.com/
aaronpeikert/repro-tutorial/commit/6556a9395fcdd600b5b0c5358f92a2c6635ae360)
and commit [9f7ab21](https://github.com/aaronpeikert/repro-tutorial/commit/
9f7ab212dfaf84a0398752a4b80cf14c71000d00)). We do not believe that these changes
influence the results substantively.
```

Readers can inspect and judge the changes for themselves on GitHub (https://github.com/aaronpeikert/repro-tutorial/compare/v0.0.1.1-prereg..main#diff-e21a8fa2e44b297dfefef329a6ef56d283488d467c4b4ffe2a014111e52a170b accessed on 4 December 2021).

The last thing we need to preregister is the reporting of our results with the combination of the functions `planned_analysis()` and `report_analysis()`.

~~~
```{r, echo=FALSE, results='asis'}
report_analysis(planned_analysis(mach))
```
~~~

This is an example of how the results could be reported (based on simulated data):

```
report_analysis(planned_analysis(simulate_data(900, 8, 0.3, 10)))
```

```
The Welch Two Sample t-test testing the difference of machiavellianism by gender
(mean in group male = 0.96, mean in group female = 0.79) suggests that the
effect is - negative, statistically significant, and small (difference = -0.17,
95% CI [0.12, 0.22], t(887.46) = 6.38, p < .001; Cohen's d = 0.43, 95% CI [0.30,
0.56])
```

This example of a preregistration covers a single study with a single hypothesis. To organize studies with multiple hypotheses, we suggest multiple `planned_analysis()` and `report_analysis()` functions (possibly numbered in accordance with the hypotheses, e.g., 1.2, 2.3 etc.). Preregistrations that cover multiple distinct data sources may employ multiple `simulate_data()` functions. These are merely suggestions, and researchers are encouraged to find their own way of how to best organize their analysis code.

The example rendered as a PDF file with real data (https://github.com/aaronpeikert/repro-tutorial/files/7309455/preregistration.pdf accessed on 11 October 2021) is available in the online Supplementary Material (https://github.com/aaronpeikert/repro-tutorial/releases/tag/v0.0.3.1-results accessed on 11 October 2021). The changes we made since preregistering it can be inspected on this GitHub page (https://github.com/aaronpeikert/repro-tutorial/compare/v0.0.1.1-prereg..main#diff-e21a8fa2e44b297dfefef329a6ef56d283488d467c4b4ffe2a014111e52a170b accessed on 11 October 2021).

6. Discussion

Increased automation is increasingly recognized as a means to improve the research process [65], and therefore this workflow fits nicely together with other innovations that employ automation, like machine-readable hypothesis tests [66] or automated data documentation [67]. Automated research projects promise a wide range of applications, among them PAC ([68,69] potentially to be submitted as a registered report), direct replication [70], fully automated living metanalysis [71], executable research articles [72], and other innovations such as the live analysis of born open data [73,74].

Central to these innovations is a property we call "reusability", fully promoted by the present workflow. Reusable code can run on different inputs from a similar context and produce valid outputs. This property is based on reproducibility but requires the researcher to more carefully write the software [75] such that it is *built-for-reuse* [76]. The reproducible workflow we present here is heavily automated and hence promotes reusability. Furthermore, adhering to principles of reusability typically removes errors in the code and thus increases the likelihood that the statistical analysis is correct. Therefore reproducibility facilitates traditional good scientific practices and provides the foundation for promising innovations.

6.1. Summary

This paper demonstrated how the R package `repro` supports researchers in creating reproducible research projects, including reproducible manuscripts. These are important building blocks for transparent and cumulative science because they enable others to reproduce statistical and computational results and reports later in time and on different computers. The workflow we present here rests on four software solutions, (1) version control, (2) dynamic document generation, (3) dependency tracking, and (4) software management to guarantee reproducibility. We first demonstrated how to create a reproducible research project. Then, we illustrated how such a project could be reproduced—either by the original author and/or collaborators or by a third party.

We finally presented an example of how the rigorous and automated reproducibility workflow introduced by `repro` may enable other innovations, such as Preregistration as Code (PAC). In PAC the entire reproducible manuscript, including planned analyses and results based on simulated data, is preregistered. This way, every use of a researchers' degree of freedom is disclosed. Once real data is gathered, the reproducible manuscript is (re-)created with the real data. PAC only becomes possible because reproducibility is ensured and leverages version control and dynamic document generation as key features of the workflow.

6.2. Limitations

We realize that the workflow outlined in this paper, and its application in PAC, remains challenging despite our efforts to simplify the procedure by means of the `repro` package. This paper should be considered as a starting point for those seeking to improve the reproducibility of their research. Two kinds of limitations can be distinguished. The first kind are limitations by design, which are unlikely to change. Our workflow inherits these from the software it relies on and the fundamental design principles these share with the workflow and `repro`. The second kind are limitations in `repro` and its dependencies that may be overcome by our future efforts and those of the open-source development community.

With regard to limitations by design, the workflow outlined in this paper is fundamentally incompatible with steps that cannot be automated. This principle may be at odds with some ingrained habits of researchers to mix and match manual and automated steps in data analysis. To allow for automation, many researchers will have to search for alternative software.

The automation-friendly software we present here has several technical but critical limitations. For example, `Git` can track any filetype, but tracked changes are only meaningful for text files (with endings like, `.txt`, `.csv`, `.R`, `.py`, or `.Rmd`), not for binary files (with endings like `.docx`, `.exe`, or `.zip`). Furthermore, tables and graphics dynamically generated from code are difficult to edit by hand. `Make` can automate any programmable software, but not software that is exclusively controlled through a point-and-click user interface. Finally, `Docker` can ship software that runs on Linux and can be automatically installed, which precludes much commercial or closed-source software.

This move away from software that has served researchers well for decades is understandably difficult and presents us with a conundrum. On the one hand, we firmly believe that automated reproducibility makes research more productive and collaboration easier. But, on the other hand, we expect researchers to invest considerable time in learning new tools and to persuade their collaborators to do the same. Three arguments reconcile this apparent paradox. First, this change will not happen all at once. Automated reproducibility is an ideal that we believe has many advantages, but it is not an all-or-nothing decision. Researchers can pick up one skill at a time and then help their fellow collaborators to do the same. Second, the upfront investment is required once (and efforts such as `repro` are underway to reduce it) and will pay dividends over many research projects. Third, the move towards open software for research offers several benefits beyond enabling automated reproducibility [77–80].

With regard to surmountable limitations, we acknowledge that the `repro` package is still in development. One limitation is that `repro` relies on several software dependencies, which represents a threat to long-term reproducibility in itself. For example, to benefit from automatic and convenient reproduction, researchers must use `Git`, `Make`, and `Docker`. However, `Git` and `Make` are themselves included in the `Docker` image created by `repro`. Researchers can therefore employ the `Docker` image manually to download the `Git` repository and execute `Make` for full reproduction. In other words, the only hard requirement for reproduction and therefore its Achilles' heel, is `Docker`. The `Docker` approach has two vulnerabilities. First, and more importantly, the `Docker` image for the project and the `Git` repository have to remain available. The `Dockerfile` (the plain text description to build a `Docker` image), as opposed to the image, is insufficient because it relies on too many service providers (e.g., Microsoft R Application Network, Ubuntu Package Archive). To overcome this limitation, we recommend archiving the `Git` repository and the `Docker` image with zenodo.org, a non-profit long-term storage for scientific data. The necessary steps for archival on zenodo.org are described at the end of Section [Preregistration as Code—a Tutorial].

The second vulnerability is that even if the existence of the `Docker` image and `Git` repository is guaranteed, future researchers still require software to run the image. To that end, they can either rely on `Docker` itself or `Docker`-compatible alternatives (e.g., `CoreOS rkt`, `Mesos Containerizer`, `Singularity`). The only way to remove the reliance on such external software is to turn the `Docker` image into a full operating system that subsequently can be installed and run on almost any modern computer. This process is technically possible and would guarantee reproducibility for decades without any software dependency, assuming hardware that conforms to the x86 instruction set architecture continues to be available. However, this process requires much technical knowledge and is currently not facilitated by `repro`. With regard to this vulnerability, it is worthwhile to note that the `R Markdown`, `Makefile`, and `Dockerfile` do provide information that allows researchers to trace the computational steps and recreate the computational environment manually. The `Makefile`, for example, is written in a way that researchers can manually trace the dependencies and execute commands in the right order, in case they are unable to run `Make` for some reason. Thus, hypothetically, even if `Docker` were to become unavailable one day, the `Dockerfile` still serves as unambiguous documentation of how the original system was set up, and may help future users to create a software environment that closely resembles the original.

6.3. Outlook

Open science practices are a continually evolving field where technical innovations foster changes in research practice. Open data are much more widespread today thanks to online storage facilities; preregistration is possible because there are preregistration platforms and so forth. Similarly, we hope that fully automatic reproduction, e.g., with `repro` as a technical innovation, will promote increased scientific rigor, efficiency, and productivity.

In practice, this ideal of a fully automatic reproduction of research projects can conflict with the wide range of demands for more user-friendly and powerful software. Some may find that `Make` is too complicated or that `Docker` requires too much storage space. Yet others may find that they require other programming languages or want to scale their computation across hundreds of computers, e.g., via high-performance computing clusters or cloud computing.

`repro` was designed modularly to meet many such demands. At the moment, `repro` only supports the combination of `R Markdown`, `Git`, `Make`, and `Docker`. However, there are alternatives for each of these elements that may fit better into an individual research project. `R Markdown` could be complemented or replaced by a dynamic Microsoft Word document with the help of `officer` [81] or `officedown` [82] to accommodate a wider range of journal submission standards. Instead of using formal version control with `Git`, `repro` could automatically save snapshots for increasing user-friendliness. `Make` could be replaced by the more R-centered alternative `targets` for more convenience. `Docker` could be combined with `renv` [22] to control the package versions precisely (our approach fixes the date, `renv` the exact package version). Alternatively, `Docker` could be replaced by the more lightweight `renv` if no dependencies outside of R are considered crucial. `Docker` does not satisfy the requirements of many HPC environments, but `Singularity` was designed to avoid this limitation while still being compatible with `Docker` images.

`repro`'s modular structure allows such alternative workflows, though they have not yet been implemented. Depending on the demand by users, we will implement some of them in `repro` and hope for broad adoption of computational reproducibility in the near future.

Supplementary Materials: All materials (i.e., the source code, all figures, and the data) that are neccesary for reproducing the submitted version of this article are availible at https://github.com/aaronpeikert/repro-tutorial and archived under https://doi.org/10.5281/zenodo.5724454 (accessed on 4 December 2021).

Author Contributions: A.P. took the lead in writing and provided the initial draft of the manuscript. A.M.B. and C.J.v.L. contributed further ideas, critical feedback, and revisions of the original manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We would like to thank Maximilian Stefan Ernst for his contributions to the code for the simulation study. We are grateful to Julia Delius for her helpful assistance in language and style editing. The R package `repro` was developed as part of the first author's master thesis (https://aaronpeikert.github.io/repro-thesis/ accessed on 11 October 2021) at the Humboldt-Universität zu Berlin.

Conflicts of Interest: The authors declare no conflicts of interest. We have received no financial support for the research, authorship, and/or publication of this article.

Abbreviations

The following abbreviations are used in this manuscript:

PAC	Preregistration as Code
Gb	Gigabyte
Kb	Kilobyte
CRAN	Comprehensive R Archive Network
WORCS	Workflow for Open Reproducible Code in Science

References

1. Peikert, A.; Brandmaier, A.M. A Reproducible Data Analysis Workflow with R Markdown, Git, Make, and Docker. *Quant. Comput. Methods Behav. Sci.* **2021**, *1*, e3763. [CrossRef]
2. Popper, K.R. *The Logic of Scientific Discovery*; Routledge: London, UK, 2002.
3. Obels, P.; Lakens, D.; Coles, N.A.; Gottfried, J.; Green, S.A. Analysis of Open Data and Computational Reproducibility in Registered Reports in Psychology. *Adv. Methods Pract. Psychol. Sci.* **2020**, *3*, 229–237. [CrossRef]
4. van Lissa, C.J.; Brandmaier, A.M.; Brinkman, L.; Lamprecht, A.L.; Peikert, A.; Struiksma, M.E.; Vreede, B.M. WORCS: A Workflow for Open Reproducible Code in Science. *Data Sci.* **2021**, *4*, 29–49. [CrossRef]
5. Nosek, B.A.; Ebersole, C.R.; DeHaven, A.C.; Mellor, D.T. The Preregistration Revolution. *Proc. Natl. Acad. Sci. USA* **2018**, *115*, 2600–2606. [CrossRef] [PubMed]
6. Hardwicke, T.E.; Mathur, M.B.; MacDonald, K.; Nilsonne, G.; Banks, G.C.; Kidwell, M.C.; Hofelich Mohr, A.; Clayton, E.; Yoon, E.J.; Henry Tessler, M.; et al. Data Availability, Reusability, and Analytic Reproducibility: Evaluating the Impact of a Mandatory Open Data Policy at the Journal Cognition. *R. Soc. Open Sci.* **2018**, *5*, 180448. [CrossRef]
7. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2021.
8. Vuorre, M.; Curley, J.P. Curating Research Assets: A Tutorial on the Git Version Control System. *Adv. Methods Pract. Psychol. Sci.* **2018**, *1*, 219–236. [CrossRef]
9. Bryan, J. Excuse Me, Do You Have a Moment to Talk About Version Control? *Am. Stat.* **2018**, *72*, 20–27. [CrossRef]
10. Nuijten, M.B.; Hartgerink, C.H.J.; van Assen, M.A.L.M.; Epskamp, S.; Wicherts, J.M. The Prevalence of Statistical Reporting Errors in Psychology (1985–2013). *Behav. Res. Methods* **2016**, *48*, 1205–1226. [CrossRef]
11. Knuth, D.E.; Levy, S. *The CWEB System of Structured Documentation*; Addison-Wesley Longman: Boston, MA, USA, 1994.
12. Claerbout, J.F.; Karrenbach, M. Electronic Documents Give Reproducible Research a New Meaning. *SEG Tech. Program Expand. Abstr.* **1992**, 601–604. [CrossRef]
13. Lamport, L. *LATEX: A Document Preparation System: User's Guide and Reference Manual*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 1994.
14. Allaire, J.; Xie, Y.; Foundation, R.; Wickham, H.; Journal of Statistical Software; Vaidyanathan, R.; Association for Computing Machinery; Boettiger, C.; Elsevier; Broman, K.; et al. *Articles: Article Formats for R Markdown*; R Package Version 0.19; 2021. Available online: https://pkgs.rstudio.com/rticles/ (accessed on 4 December 2021).
15. El Hattab, H.; Allaire, J. Revealjs: R Markdown Format for 'Reveal, Js' Presentations. 2017. Available online: https://bookdown.org/yihui/rmarkdown/revealjs.html (accessed on 4 December 2021).
16. O'Hara-Wild, M.; Hyndman, R. Vitae: Curriculum Vitae for r Markdown. 2021. Available online: https://cran.r-project.org/web/packages/vitae/vignettes/vitae.html (accessed on 4 December 2021).
17. Xie, Y.; Dervieux, C.; Riederer, E. *R Markdown Cookbook*, 1st ed.; The R Series; Taylor and Francis, CRC Press: Boca Raton, FL, USA, 2020.
18. Silver, A. Software Simplified. *Nature* **2017**, *546*, 173–174. [CrossRef]
19. Boettiger, C.; Eddelbuettel, D. An Introduction to Rocker: Docker Containers for R. *R J.* **2017**, *9*, 527. [CrossRef]
20. Wickham, H.; Averick, M.; Bryan, J.; Chang, W.; McGowan, L.D.; François, R.; Grolemund, G.; Hayes, A.; Henry, L.; Hester, J.; et al. Welcome to the tidyverse. *J. Open Source Softw.* **2019**, *4*, 1686. [CrossRef]
21. Wiebels, K.; Moreau, D. Leveraging Containers for Reproducible Psychological Research. Advances in Methods and Practices in Psychological. *Science* **2021**, *4*. [CrossRef]
22. Ushey, K. Renv: Project Environments. R Package Version 0.13.2. 2021. Available online: https://rstudio.github.io/renv/articles/renv.html (accessed on 4 December 2021).
23. Wickham, H.; Bryan, J. Usethis: Automate Package and Project Setup. 2021. Available online: https://usethis.r-lib.org (accessed on 4 December 2021).
24. Parasuraman, R.; Mouloua, M. *Automation and Human Performance: Theory and Applications*, 1st ed.; CRC Press: Boca Raton, FL, USA, 2019.
25. RStudio Team. *RStudio: Integrated Development Environment for R. RStudio*; PBC: Boston, MA, USA, 2021.
26. Peikert, A.; Brandmaier, A.M.; van Lissa, C.J. Repro: Automated Setup of Reproducible Workflows and Their Dependencies. R Package Version 0.1.0. 2021. Available online: https://github.com/aaronpeikert/repro (accessed on 4 December 2021).
27. Xie, Y.; Allaire, J.J.; Grolemund, G. *R Markdown: The Definitive Guide*; CRC Press: Boca Raton, FL, USA, 2019.
28. Aust, F.; Barth, M. Papaja: Create APA Manuscripts with R Markdown. 2020. Available online: http://frederikaust.com/papaja_man/ (accessed on 4 December 2021).
29. Association, A.P. (Ed.) *Publication Manual of the American Psychological Association*, 7th ed.; American Psychological Association: Washington, DC, USA, 2019.
30. DeCoster, J.; Sparks, E.A.; Sparks, J.C.; Sparks, G.G.; Sparks, C.W. Opportunistic Biases: Their Origins, Effects, and an Integrated Solution. *Am. Psychol.* **2015**, *70*, 499–514. [CrossRef]
31. Silberzahn, R.; Uhlmann, E.L.; Martin, D.P.; Anselmi, P.; Aust, F.; Awtrey, E.; Bahnik, S.; Bai, F.; Bannard, C.; Bonnier, E.; et al. Many Analysts, One Data Set: Making Transparent How Variations in Analytic Choices Affect Results. *Adv. Methods Pract. Psychol. Sci.* **2018**, *1*, 337–356. [CrossRef]

32. Bowman, S.; DeHaven, A.; Errington, T.; Hardwicke, T.E.; Mellor, D.T.; Nosek, B.A.; Soderberg, C.K. OSF Prereg Template 2020. *MetaArXiv* **2020**. [CrossRef]
33. Bakker, M.; Veldkamp, C.L.S.; van Assen, M.A.L.M.; Crompvoets, E.A.V.; Ong, H.H.; Nosek, B.A.; Soderberg, C.K.; Mellor, D.; Wicherts, J.M. Ensuring the Quality and Specificity of Preregistrations. *PLoS Biol.* **2020**, *18*, e3000937. [CrossRef]
34. Bakker, M.; Veldkamp, C.L.S.; van den Akker, O.R.; van Assen, M.A.L.M.; Crompvoets, E.; Ong, H.H.; Wicherts, J.M. Recommendations in Pre-Registrations and Internal Review Board Proposals Promote Formal Power Analyses but Do Not Increase Sample Size. *PLoS ONE* **2020**, *15*, e0236079. [CrossRef]
35. Steegen, S.; Dewitte, L.; Tuerlinckx, F.; Vanpaemel, W. Measuring the Crowd within Again: A Pre-Registered Replication Study. *Front. Psychol.* **2014**, *5*. [CrossRef]
36. Morris, T.P.; White, I.R.; Crowther, M.J. Using Simulation Studies to Evaluate Statistical Methods. *Stat. Med.* **2019**, *38*, 2074–2102. [CrossRef]
37. Paxton, P.; Curran, P.J.; Bollen, K.A.; Kirby, J.; Chen, F. Monte Carlo Experiments: Design and Implementation. *Struct. Equ. Model. Multidiscip. J.* **2001**, *8*, 287–312. [CrossRef]
38. Skrondal, A. Design and Analysis of Monte Carlo Experiments: Attacking the Conventional Wisdom. *Multivar. Behav. Res.* **2000**, *35*, 137–167. [CrossRef] [PubMed]
39. Goldfeld, K.; Wujciak-Jens, J. Simstudy: Illuminating Research Methods through Data Generation. *J. Open Source Softw.* **2020**, *5*, 2763. [CrossRef]
40. Revelle, W. *Psych: Procedures for Psychological, Psychometric, and Personality Research*; Northwestern University: Evanston, IL, USA, 2021.
41. Wicherts, J.M.; Veldkamp, C.L.S.; Augusteijn, H.E.M.; Bakker, M.; van Aert, R.C.M.; van Assen, M.A.L.M. Degrees of Freedom in Planning, Running, Analyzing, and Reporting Psychological Studies: A Checklist to Avoid p-Hacking. *Front. Psychol.* **2016**, *7*, 1832. [CrossRef] [PubMed]
42. Szollosi, A.; Kellen, D.; Navarro, D.J.; Shiffrin, R.; van Rooij, I.; Van Zandt, T.; Donkin, C. Is Preregistration Worthwhile? *Trends Cogn. Sci.* **2020**, *24*, 94–95. [CrossRef] [PubMed]
43. Nosek, B.A.; Beck, E.D.; Campbell, L.; Flake, J.K.; Hardwicke, T.E.; Mellor, D.T.; van 't Veer, A.E.; Vazire, S. Preregistration Is Hard, And Worthwhile. *Trends Cogn. Sci.* **2019**, *23*, 815–818. [CrossRef]
44. Meehl, P.E. Theoretical Risks and Tabular Asterisks: Sir Karl, Sir Ronald, and the Slow Progress of Soft Psychology. *J. Consult. Clin. Psychol.* **1978**, *46*, 806–834. [CrossRef]
45. Brandmaier, A.M.; Jacobucci, R. Machine-Learning Approaches to Structural Equation Modeling. In *Handbook of Structural Equation Modeling*, 2nd ed.; Hoyle, R.H., Ed.; Guilford Press: New York, NY, USA, in press.
46. Hastie, T.; Tibshirani, R.; Friedman, J.H. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition, Corrected at 12th Printing 2017 ed.*; Springer Series in Statisticsl; Springer: New York, NY, USA, 2017.
47. Daróczi, G.; Tsegelskyi, R. Pander: An R 'pandoc' Writer. 2021. Available online: https://www.r-project.org/nosvn/pandoc/pander.html (accessed on 4 December 2021).
48. Hlavac, M. *Stargazer: Well-Formatted Regression and Summary Statistics Tables*; Central European Labour Studies Institute (CELSI): Bratislava, Slovakia, 2018.
49. Stanley, D. apaTables: Create American Psychological Association (APA) Style Tables, 2021. Available online: https://dstanley4.github.io/apaTables/articles/apaTables.html (accessed on 4 December 2021).
50. Makowski, D.; Ben-Shachar, M.S.; Patil, I.; Lüdecke, D. Automated Results Reporting as a Practical Tool to Improve Reproducibility and Methodological Best Practices Adoption. CRAN 2021. Available online: https://easystats.github.io/report/ (accessed on 4 December 2021).
51. European Organization For Nuclear Research. *OpenAIRE*; Zenodo: online, 2013. [CrossRef]
52. Weston, S.J.; Ritchie, S.J.; Rohrer, J.M.; Przybylski, A.K. Recommendations for Increasing the Transparency of Analysis of Preexisting Data Sets. *Adv. Methods Pract. Psychol. Sci.* **2019**, *2*, 214–227. [CrossRef]
53. International Council for Harmonisation of Technical Requirements for Registration of Pharmaceuticals for Human Use. E 9 Statistical Principles for Clinical Trials. 1998. Available online: https://www.ema.europa.eu/en/ich-e9-statistical-principles-clinical-trials (accessed on 4 December 2021).
54. Thabane, L.; Ma, J.; Chu, R.; Cheng, J.; Ismaila, A.; Rios, L.P.; Robson, R.; Thabane, M.; Giangregorio, L.; Goldsmith, C.H. A Tutorial on Pilot Studies: The What, Why and How. *BMC Med. Res. Methodol.* **2010**, *10*, 1. [CrossRef] [PubMed]
55. Yuan, I.; Topjian, A.A.; Kurth, C.D.; Kirschen, M.P.; Ward, C.G.; Zhang, B.; Mensinger, J.L. Guide to the Statistical Analysis Plan. *Pediatric Anesth.* **2019**, *29*, 237–242. [CrossRef]
56. Fagerland, M.W. T-Tests, Non-Parametric Tests, and Large Studies—A Paradox of Statistical Practice? *BMC Med Res. Methodol.* **2012**, *12*, 78. [CrossRef]
57. Horton, N.J.; Switzer, S.S. Statistical Methods in the Journal. *New Engl. J. Med.* **2005**, *353*, 1977–1979. [CrossRef] [PubMed]
58. Putnick, D.L.; Bornstein, M.H. Measurement Invariance Conventions and Reporting: The State of the Art and Future Directions for Psychological Research. *Dev. Rev.* **2016**, *41*, 71–90. [CrossRef] [PubMed]
59. Frost, C.; Thompson, S.G. Correcting for Regression Dilution Bias: Comparison of Methods for a Single Predictor Variable. *J. R. Stat. Soc. Ser. A* **2000**, *163*, 173–189. [CrossRef]

60. Stonehouse, J.M.; Forrester, G.J. Robustness of the *t* and U Tests under Combined Assumption Violations. *J. Appl. Stat.* **1998**, *25*, 63–74. [CrossRef]
61. Zimmerman, D.W.; Zumbo, B.D. Rank Transformations and the Power of the Student *t* Test and Welch *t* Test for Non-Normal Populations with Unequal Variances. *Can. J. Exp. Psychol./Rev. Can. De Psychol. Exp.* **1993**, *47*, 523–539. [CrossRef]
62. Brandmaier, A.M.; von Oertzen, T.; Ghisletta, P.; Lindenberger, U.; Hertzog, C. Precision, Reliability, and Effect Size of Slope Variance in Latent Growth Curve Models: Implications for Statistical Power Analysis. *Front. Psychol.* **2018**, *9*, 294. [CrossRef]
63. Harrison, R.L. Introduction to Monte Carlo Simulation. *AIP Conf. Proc.* **2010**, *1204*, 17–21. [CrossRef]
64. Raychaudhuri, S. Introduction to Monte Carlo Simulation. In Proceedings of the 2008 Winter Simulation Conference, Miami, FL, USA, 7–10 December 2008; pp. 91–100. [CrossRef]
65. Rouder, J.N.; Haaf, J.M.; Snyder, H.K. Minimizing Mistakes in Psychological Science. *Adv. Methods Pract. Psychol. Sci.* **2019**, *2*, 3–11. [CrossRef]
66. Lakens, D.; DeBruine, L.M. Improving Transparency, Falsifiability, and Rigor by Making Hypothesis Tests Machine-Readable. *Adv. Methods Pract. Psychol. Sci.* **2021**, *4*. [CrossRef]
67. Arslan, R.C. How to Automatically Document Data With the Codebook Package to Facilitate Data Reuse. Advances in Methods and Practices in Psychological. *Science* **2019**, *2*, 169–187. [CrossRef]
68. Nosek, B.A.; Lakens, D. Registered Reports. *Soc. Psychol.* **2014**, *45*, 137–141. [CrossRef]
69. Chambers, C. What's next for Registered Reports? *Nature* **2019**, *573*, 187–189. [CrossRef] [PubMed]
70. Simons, D.J. The Value of Direct Replication. *Perspect. Psychol. Sci.* **2014**, *9*, 76–80. [CrossRef] [PubMed]
71. Elliott, J.H.; Turner, T.; Clavisi, O.; Thomas, J.; Higgins, J.P.T.; Mavergames, C.; Gruen, R.L. Living Systematic Reviews: An Emerging Opportunity to Narrow the Evidence-Practice Gap. *PLoS Med.* **2014**, *11*, e1001603. [CrossRef]
72. eLife Sciences Publications. eLife Launches Executable Research Articles for Publishing Computationally Reproducible Results. 2020. Available online: https://elifesciences.org/for-the-press/eb096af1/elife-launches-executable-research-articles-for-publishing-computationally-reproducible-results (accessed on 4 December 2021).
73. Rouder, J.N. The What, Why, and How of Born-Open Data. *Behav. Res. Methods* **2016**, *48*, 1062–1069. [CrossRef] [PubMed]
74. Kekecs, Z.; Aczel, B.; Palfi, B.; Szaszi, B.; Szecsi, P.; Zrubka, M.; Kovacs, M.; Bakos, B.E.; Cousineau, D.; Tressoldi, P.; et al. Raising the Value of Research Studies in Psychological Science by Increasing the Credibility of Research Reports: The Transparent Psi Project—Preprint. *PsyArXiv* **2020**. [CrossRef]
75. Lanergan, R.G.; Grasso, C.A. Software Engineering with Reusable Designs and Code. In *Software Reusability: Vol. 2, Applications and Experience*; Association for Computing Machinery: New York, NY, USA, 1989; pp. 187–195.
76. Al-Badareen, A.B.; Selamat, M.H.; Jabar, M.A.; Din, J.; Turaev, S. Reusable Software Components Framework. In Proceedings of the European Conference of Systems, and European Conference of Circuits Technology and Devices, and European Conference of Communications, and European Conference on Computer Science, Kuantan, Pahang, Malaysia, 27–29 June 2011; World Scientific and Engineering Academy and Society (WSEAS): Stevens Point, WI, USA, 2010; ECS'10/ECCTD'10/ECCOM'10/ECCS'10, pp. 126–130.
77. Schaffner, A.C. The Future of Scientific Journals: Lessons from the Past. *Inf. Technol. Libr.* **1994**, *13*, 239–247.
78. Fitzgerald, B. The Transformation of Open Source Software. *MIS Q.* **2006**, *30*, 587–598. [CrossRef]
79. Chaldecott, J.A. A History of Scientific and Technical Periodicals: The Origins and Development of the Scientific and Technological Press. *Br. J. Hist. Sci.* **1965**, *2*, 360–361. [CrossRef]
80. Sonnenburg, S.; Braun, M.L.; Ong, C.S.; Bengio, S.; Bottou, L.; Holmes, G.; LeCun, Y.; Müller, K.R.; Pereira, F.; Rasmussen, C.E.; et al. The Need for Open Source Software in Machine Learning. *J. Mach. Learn. Res.* **2007**, *8*, 2443–2466.
81. Gohel, D. Officer: Manipulation of Microsoft Word and PowerPoint Documents. 2021. Available online: https://davidgohel.github.io/officer/ (accessed on 4 December 2021).
82. Gohel, D.; Ross, N. Officedown: Enhanced 'R Markdown' Format for 'Word' and 'PowerPoint'. 2021. Available online: https://davidgohel.github.io/officedown/ (accessed on 4 December 2021).