Supplementary Appendix

The FAIRification of research in real-world evidence: A practical introduction to reproducible analytic workflows using Git and R

Table of contents

1 Supplementary material

1.1 Example for an end-to-end workflow cycle using Git

This section aims to summarize a minimally commented end-to-end workflow for incorporating and tracking changes for an analysis step of an exemplary RWE study. To that end, we use the same example as in the main manuscript.

Step 1: In case we work with multiple branches (see Section 1.2 below), we can first check if we are on the correct branch where we want to implement our changes.

```
# check the current branch
git branch
```

Step 2: Before initiating any new changes, it can be a good idea to check if a collaborator, who may work on the same branch, made any changes.

```
# git fetch to see changes without integrating them locally
git fetch

# git merge to integrate changes to local repository
get merge

# or directly git pull changes
git pull
```

Step 3: We make changes to the analysis script `scripts/03_propensity_score_analysis.R` by adding a title to a plot illustrating the propensity score overlap.

```
# load libraries
library(tidyverse)

# [...]

# add a title to propensity score plot
ps_plot <- exposure_plot %>%
  ggplot2::ggplot(ggplot2::aes(x = ps, fill = factor(exposure))) +
  ggplot2::geom_density(alpha = .5) +
  ggplot2::theme_bw() +
  ggplot2::labs(
    # added the title here
    title = "Propensity score overlap",
    x = "Pr(exposure)",
    y = "Density",
    fill = "Exposed"
    )

ps_plot

# [...]
```

Step 4: After having all changes made, we stage the files we want to commit next.

```
# stage command
git add 'scripts/03_propensity_score_analysis.R'
```

Step 5: Next, we commit all changes.

```
# commit command
git commit -m 'added title to plot illustrating propensity score overlap'
```

Step 6: Synchronize the changes with the remote repository, so the code is up-to-date on GitHub.

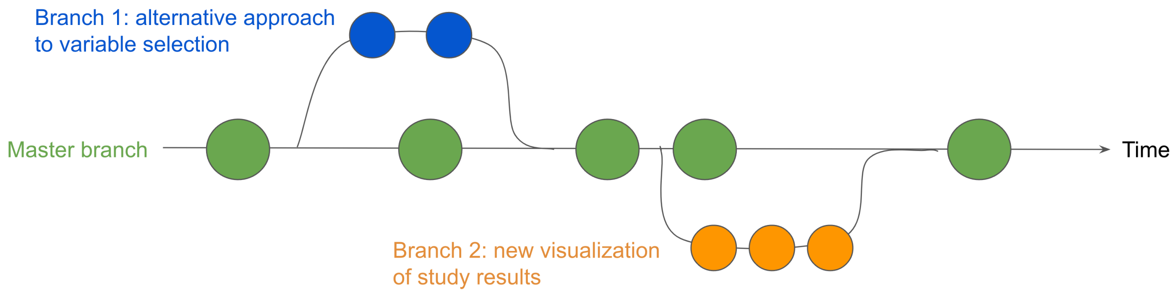
```
# push local changes to remote repository
git push
```

1.2 Branching

Sometimes it is necessary to implement changes to a document or code without wanting to interfere with a current "`production`" version of a document or code. However, a researcher may still want to be able to "`merge`" back potential changes, e.g. to integrate new analyses into a script or to develop a new set of features or functions to an application. To do this in a safe, transparent and efficient manner, `Git` provides the option to **branch** our repository. The default branch of any repository is the `main` branch.

Branching is a concept that is highly adapted in software development where the `main` branch reflects the software version that is deployed for public use (i.e., the production version) and a `development` branch that is the version that implements new features (i.e., the beta version). Hence, `branches` are used in order to have the flexibility to develop new features without jeopardizing the current `main` and `development` versions.

Branching can have two very useful use cases for adaption in real-world evidence (RWE) studies. First, new analyses or improvements of existing analyses can be implemented without having to "mess" with existing code which works perfectly. Once a new analysis or visualization of results was established in a new `named branch` and works without errors, this branch can be `merged` back to the `main` branch (as depicted in the illustration below).


Branch 1: alternative approach to variable selection
Master branch
Time
Branch 2: new visualization of study results

Second, branches can be re-purposed as backup versions of the repository at project landmarks. Examples could range from the first round of review among co-authors to the initial submission of a manuscript draft to a journal, corresponding rounds of revisions and the final version. This way it can be ensured that also prior versions of analyses can be reproduced without problems.

To create a new branch and switch to it from the main branch, you can run the `git checkout` command with the -b switch. Let's say we call the new branch `lasso_variable_selection`.

```
git checkout -b lasso_variable_selection
```

This is shorthand for:

```
git branch lasso_variable_selection
git checkout lasso_variable_selection
```

It is possible to switch between multiple branches back and forth and it's important to keep in mind that changes need to be committed before switch to the new branch branch. A user can e.g. switch back to the master branch by running the following command:

```
git checkout master
```

Suppose we have decided that the `lasso_variable_selection` approach led to a better balancing of the two new user cohorts after propensity score matching and we are ready to merge the implemented changes back into our main branch. This can be achieved by checking out the branch you wish to merge into (in this case `main` and then run the git merge command:

```
git checkout main
git merge lasso_variable_selection
```

If the branch is merged and there is no desire anymore to use this branch any longer, the `lasso_variable_selection` branch can be deleted with the following command:

```
git branch -d lasso_variable_selection
```

These steps can also easily be done on remote repositories like GitHub or GitLab. The only difference to keep in mind is that GitHub counterintuitively calls the procedure of merging two branches `pull request` while GitLab refers to it as `merge request`.

In case `Git` recognizes any clashes or incompatibilities of the new code with the existing code it will notify the user and give us the chance to reconcile inconsitencies.

1.3 Advanced readings on `Git` and other resources

- Main `Git` website to download and learn more about `Git:` https://git-scm.com
- Pro Git by Scott Chacon and Ben Straub.
- Comprehensive guide on the use of Git in combination with R: https://happygitwithr.com/
- Cheatsheet on most important `Git` commands: https://education.github.com/git-cheat-sheet-education.pdf
- GitHub Desktop App: https://desktop.github.com/
- Git manual by *Atlassian* (provider of the `Bitbucket` platform): https://www.atlassian.com/git
- Coursera courses:
  - Introduction to Git and GitHub (Google)
  - Version Control with Git (Atlassian)
  - Getting Started with Git and GitHub (IBM)
  - Making Data Science Work for Clinical Reporting (Genentech)
- Quarto (open-source scientific and technical publishing system for reproducible research): https://quarto.org
- Choosing a suitable license for a remote repository: Public repositories need to be licensed to be truly open source. If no license is provided, the work is by default under exclusive copyright. The website https://choosealicense.com provides an excellent overview of the different types of licenses.
- Useful Git repositories for observational healthcare research:
  - Division of Pharmacoepidemiology and Pharmacoeconomics: https://gitlab-scm.partners.org/drugepi
  - Observational Health Data Sciences and Informatics (OHDSI): https://github.com/OHDSI
  - Darwin EU® Private Development Repository: https://github.com/darwin-eu-dev
  - Pharmaverse (Network of developers of curated open source R packages for clinical reporting): https://pharmaverse.org & https://github.com/pharmaverse/

2 Reproducible computation environments in R

Besides the usage of `Git` and remote repositories as a version control systems, there are further relevant aspects that determine the reproducibility of analytic code when using open-source software such a R. To complement the introduction and discussion on `Git` in the main manuscript, we here want to elaborate on a few further considerations of computational reproducibility when using R.

2.1 Keeping track of R software versions and paths

A major advantage of modern programming languages is that they provide a large pool of open-source packages. Packages are compilations of functions that automate frequently used procedures (e.g., querying new user cohorts[1] or propensity score matching[2]) which expedites the implementation of RWE studies, mitigates copy-paste errors and makes code much more friendly to read and review.

A further step to improve reproducibility when using the R programming language is to ensure that anyone, intending to reproduce analytic results from a `Git` repository, is able to work off the same R package versions and dependencies without having to perform manual installations or path adjustments. These requirements are nothing exclusive to R, but apply to any type of programming language.

2.2 Reproducible R computing environments

One way to achieve this is through self-contained computing environments via Docker images which are isolated and standardized environments which include everything needed to run and deploy applications and code regardless of a user's local operating system.[3,4] Another more lightweight, yet effective way is by using the **`renv`** R package[5], which helps creating **re**producible **env**ironments and manages R package dependencies for a given project. Through a lockfile, the `renv` package keeps track of the underlying version of R that is being used as well as all R packages, their versions and the installation source (e.g., https://cran.r-project.org/). To synchronize the R environment with that of a collaborator, e.g., after cloning a repository, the only step necessary is running the following command in R.

```r
renv::restore()
```

Appending the session info to the end of an R script can provide further context about the dependencies and versions of an analytic script.

```r
sessionInfo()
```

2.3 Absolute versus relative paths

Another common issue when trying to reproduce analytic code is the use of absolute paths such as in the following example, where we try to call an `icd_code` file from the project's `protocol` sub-directory.

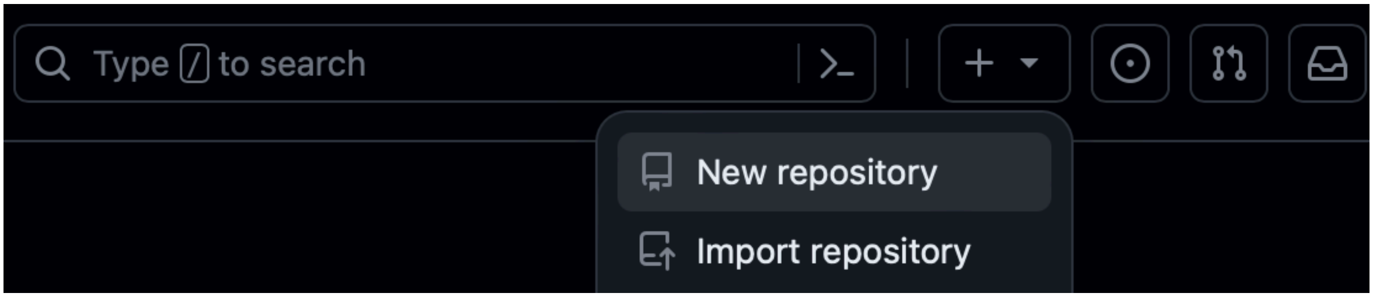
```
icd_codes <- read.csv('C:/User/Projects/rwe_study/protocol/icd_codes.csv')
```

The use of absolute paths will make it impossible for any collaborator to run the code without having to manually adapt every single path throughout a repository which is tedious and time consuming. A better solution to this is the use of relative paths. A popular R package to implement this is the `here` package[6] which enables file referencing in project-oriented workflows. The main function "`here"` creates paths relative to the top-level of the project directory (in our example the `rwe_study` directory). Applied to our example project, the following command would automatically reference the correct path and could be used without modification by any collaborator re-running this script.

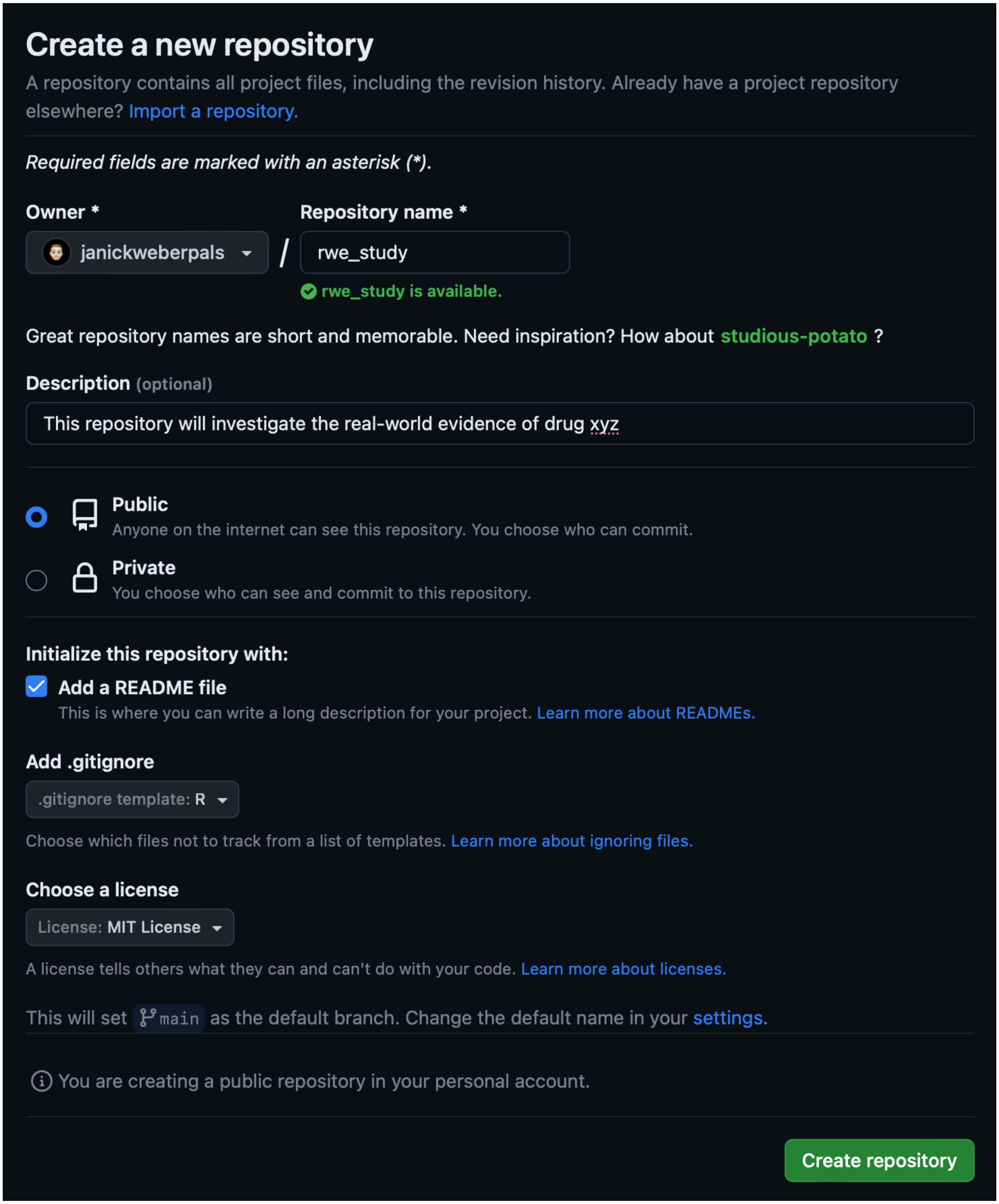
```
library(here)
icd_codes <- read.csv(here("protocol", "icd_codes.csv"))
```

3 Supplementary Figures

Supplementary Figure 1: Starting a new repository through the GitHub GUI.



Supplementary Figure 2: Populating basic information about the new repository.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Owner * janickweberpals / Repository name * rwe_study

✓ rwe_study is available.

Great repository names are short and memorable. Need inspiration? How about studious-potato ?

Description (optional)

This repository will investigate the real-world evidence of drug xyz

● Public
Anyone on the internet can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

Initialize this repository with:

☑ Add a README file
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: R

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

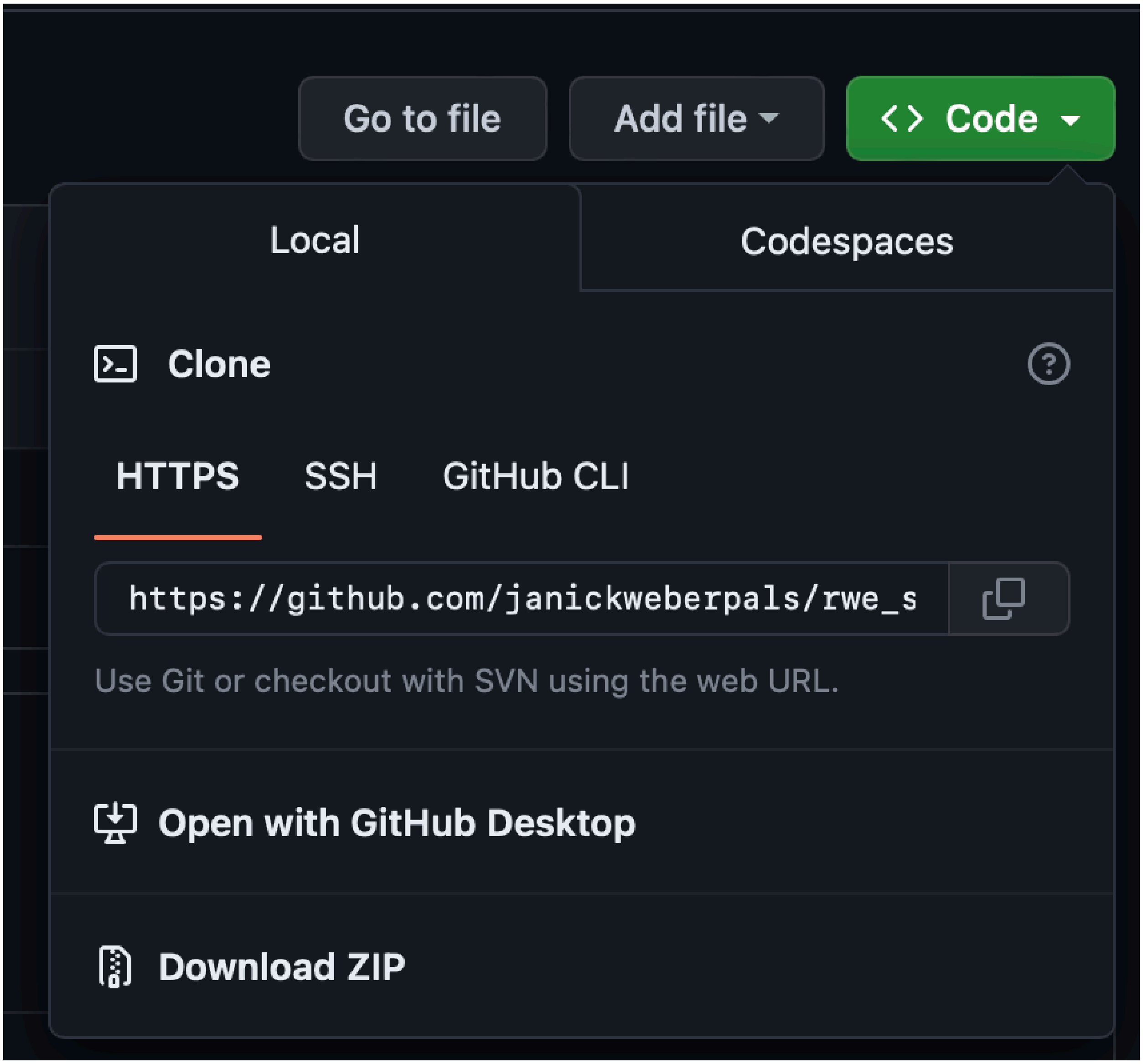
License: MIT License

A license tells others what they can and can't do with your code. Learn more about licenses.

This will set main as the default branch. Change the default name in your settings.

ⓘ You are creating a public repository in your personal account.

Supplementary Figure 3: Clone a GitHub repository using the HTTPS link and the git clone command



```
https://github.com/janickweberpals/rwe_s
```

Use Git or checkout with SVN using the web URL.

References

1. Wickham H, Averick M, Bryan J, *et al.* Welcome to the {tidyverse}. 2019; **4**: 1686. doi:10.21105/joss.01686.
2. Ho DE, Imai K, King G, Stuart EA. {MatchIt}: Nonparametric preprocessing for parametric causal inference. 2011; **42**. doi:10.18637/jss.v042.i08.
3. Boettiger C, Eddelbuettel D. An Introduction to Rocker: Docker Containers for R. *The R Journal* 2017; **9**: 527. doi:10.32614/rj-2017-065.
4. Nüst D, Eddelbuettel D, Bennett D, *et al.* The Rockerverse: Packages and Applications for Containerisation with R. *The R Journal* 2020; **12**: 437. doi:10.32614/rj-2020-007.
5. Ushey K. Renv: Project environments. 2022. Available at: https://CRAN.R-project.org/package=renv.
6. Müller K. Here: A simpler way to find your files. 2020. Available at: https://CRAN.R-project.org/package=here.