The FAIRification of pharmacoepidemiological research: A practical introduction to reproducible analytic workflows using git and R

**Authors**: Janick Weberpals1, Shirley V. Wang1

**Author affiliation**: 1Division of Pharmacoepidemiology and Pharmacoeconomics, Department of Medicine, Brigham and Women’s Hospital, Harvard Medical School, Boston, MA

**Correspondence:**

Janick Weberpals, RPh, PhD

Division of Pharmacoepidemiology and Pharmacoeconomics,

Department of Medicine, Brigham and Women’s Hospital, Harvard Medical School,

1620 Tremont Street, Suite 3030-R, Boston, MA 02120, USA

Phone: 617-278-0932

Fax: 617-232-8602

Email: [jweberpals@bwh.harvard.edu](mailto:jweberpals@bwh.harvard.edu)

**Word count:** xxxx words

**Tables:** 1

**Figures:** 5

**Supplementary material:**

**Keywords:** Transparency, Reproducibility, Git, R, Version control, Real-world evidence

**Funding:** There was no specific funding for this manuscript.

**Disclosures/COI:** Janick Weberpals reports no conflicts of interest.

Analytical code sharing statement: This manuscript was written using Quarto version 1.3.353 (<https://quarto.org/>) under R version 4.1.2. All materials can be found at <https://gitlab-scm.partners.org/drugepi/fair-epi>. Detailed information on packages and versions can be found in the renv.lock file in the linked repository.

**Acknowledgments:** The authors would like to thank …

# Abstract

**Objective**

**Study Design and Settings**

**Results**

**Conclusions**

# 1. Introduction

Transparency and reproducibility in conducting healthcare database studies in pharmacoepidemiology are critical scientific requirements for meaningful research. While many advances have been made in the documentation and reporting of study protocols and results1–4, the transparency around the actual implementation and analytical steps is still significantly lacking, especially when analyses are implemented via line programming steps in statistical programming languages like R5 or Python. Although in other quantitative disciplines such as computational biology6–8 or health informatics9, the sharing of programming code is rather the rule than the exception, this practice is not as common in the (pharmaco-)epidemiological community.

Version control systems (VCS), such as *Git*, can thereby provide easy and powerful tools to keep track of the versioning of important files and documents such as protocols, analytical code, tables and figures. Thereby, it can extend the principles of *FAIR* *data*10 to the implementation of pharmacoepidemiological study workflows by making all components of a study **F**indable, **A**ccessible, **I**nteroperable, and **R**eproducible (**FAIR**).

VCS enable researchers to track and resolve errors, collaborate with peers, and share resources instantaneously, for example, upon submission or publication of a manuscript. By design, VCS workflows automatically empower users to comment, track and compare changes made to files and hence, increase the ability to comprehend the evolution of a project over time while maintaining a complete audit trail of all documents.11 Without VCS, this is often naturally done by assigning uninformative file names such as “*final\_analysis\_1\_rev\_more\_changes.R*” which is bad practice as this is highly error prone, lacking transparency for collaborating project members and outside personnel, and changes are difficult to track and reconcile. Particularly in disciplines like pharmacoepidemiology, where analyses can have far reaching impact on healthcare decisions, analytical code needs to be accessible and reproducible by anyone, not only the analyst who originally carried out the analysis.

To increase the adoption of transparent and reproducible workflows in real-world evidence (RWE) studies using VCS, this tutorial aims to give a practical introduction tailored for pharmacoepidemiologists on how to set up, structure, and implement workflows using *Git*, which is the most frequently used VCS to date.12 We will first provide a brief introduction to Git and how it can be used for collaboration and dissemination of study results using project repositories. Then we will provide a technical step-by-step guidance on how to integrate Git in an analytical workflow using line commands. We will additionally showcase examples using the R open-source programming language, although the basic principles can be applied to any major coding language.

# 2. Git in a nutshell

Git is a free and open source distributed VCS software which was developed in 2005 by the LINUX developer community primarily with the intent to handle large software projects efficiently.13 The way to conceptually think about how Git works is that it makes a *snapshot* of a research project repository every time the state of a project is saved. For our purposes, a repository can be seen as an isolated project directory which includes all neccessary files and documents for a given research study ( e.g., protocol, programming code, manuscript, tables, figures, etc.).

In its basic form, Git is a local software and does not need any internet connectivity or connection to a remote server to store *local* changes made to files in a repository. However, to leverage the full potential of Git, it is almost always used in combination with web-based remote repository hosting services such as *GitHub*, *GitLab*, *Bitbucket* or other platforms (henceforth referred to as remote repositories, **Overview Box 1**).

|  |
| --- |
| **Overview Box 1**. Differences between Git and remote repository platforms (GitHub, GitLab, etc.) |
| **Git**   * Is an open-source software and distributed version control system * Generally operated through command line tools, graphical user interfaces are available * Installed locally on a system to track local changes of a Git repository   **Remote repository platforms (GitHub, GitLab, etc.)**   * Web-based services, usually provided by a commercial entity * Usually free for basic functions, paid services for more advanced features * Needed to collaborate on shared repositories and dissemination of code * Offer a graphical user interface and other functionalities that complement basic Git functions |

These platforms make it possible to synchronize local Git repositories across multiple project members and thereby improve collaboration. They further complement Git functionalities by providing a graphical user interface (GUI) to visualize changes made to files and often offer advanced features such as the automation of workflows using continuous integration and deployment (CI/CD) or, most recently, AI-powered virtual coding assistants (e.g., GitHub co-pilot). Generally speaking, it’s possible to use Git without a remote repository, while it’s unusual to use a remote repository without Git.

# 3. A step-by-step introduction on how to use Git in a pharmacoepidemiology research study

In this section, we want to give a technical introduction on how Git and remote repositories can be used in RWE studies. For the following examples, we focus on workflows utilizing R/RStudio ([https://posit.co/download/rstudio-desktop/](https://posit.co/download/rstudio-desktop/https://posit.co/download/rstudio-desktop/)) and GitHub (<https://github.com/>) as remote repository, although the general concepts are fairly similar and easily transferable to other programming languages and remote repository providers, respectively. We will show examples for both command line interface (CLI) prompts as well as for RStudio’s integrated GUI. For more details, we refer to the book *Pro Git* by Chacon and Straub.13

## 3.1 Installing Git

Git is available for every contemporary operating system and can be downloaded on the official Git website: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

To check if the installation succeeded and show the available version, the following prompt in the CLI can be used.

git --version

## 3.2 Configuration and first time use

For Git to be able to track who made changes, the first step is to provide a name and email address as shown below. This needs to be done only once if the --global parameter is set.

git config --global user.name "First Last"  
git config --global user.email "name@domain.edu"

## 3.3 Initialization of a new Git-controlled repository

### 3.3.1 Local initialization

To start a new project (e.g., a new RWE study), the next step is to create and navigate to a new directory where all documents that are intended to be tracked via Git will be stored (in the following referred to as *repository*). In this example, the study repository is called rwe\_study.

mkdir ~/rwe\_study  
cd ~/rwe\_study

The command to initialize this directory as a Git repository is

git init

This initiates the creation of the .git sub-directory within the study repository which will contain the entire version history of the rwe\_study project. Typically, users don’t directly interact with this sub-directory and the only thing to keep in mind is that this folder keeps all files needed to track changes.

### 3.3.2 GUI initialization and git clone

An alternative to creating a new repository locally is through creating a new project in the remote repository (in this case GitHub) and then creating a local copy, a process which is usually referred to as cloning.

To do so, users will need to log into their remote repository account and navigate to the section New repository (**Supplementary Figure 1**). This will take a user to a form to populate basic information of the new project such as the repository name (rwe\_study) and choose basic setting such as if the repository should be private or public and if a README.md (see [Section 3.4](#sec-repository-structure)) and .gitignore (see [Section 3.5.1](#sec-stage-and-commit-changes)) file should be automatically created (**Supplementary Figure 2**).

Now we can copy the pre-populated repository to our local machine by copy-pasting the link given under Clone > Copy HTTPS into the CLI (**Supplementary Figure 3**):

mkdir ~/rwe\_study  
cd ~/rwe\_study  
git clone https://gitlab.com/USERNAME/rwe\_study.gitlab.io.git

This can also be achieved directly in the RStudio GUI via File > New Project > Version Control > Git > paste the link to Repository URL ([Figure 1](#fig-RStudio-gui)).

## 3.4 Repository structure

At this step, we now can populate the repository and start working on study-relevant documents. To enhance clarity, it is recommended to use a logical structure and informative and transparent sub-directory and file names (a minimal example is illustrated in [Figure 2](#fig-structure)). To document the structure and location of files, we can include this and other critical information, that a peer or outside personnel should know concerning the project, in the README.md file. This file can be auto-generated when a project is initialized in a remote repository (see prior step in [Section 3.3.2](#sec-gui-initialization-and-git-clone)) or can be added manually aftwerwards by creating and adding a text file with the name README.md in the root directory of the repository.

To speed up these initial steps, we provide a pre-populated template of such a structured study repository based on the HARPER protocol4 under <https://gitlab-scm.partners.org/drugepi/harper>.

## 3.5 The git workflow

The most essential steps of a git workflow are summarized in [Figure 3](#fig-workflow).

### 3.5.1 Stage and commit changes

As introduced before, Git allows researchers to keep track of changes by making snapshots of the repository every time the state of a project is saved. In Git terms, this fundamental step is referred to as a commit. Generally it is up to the user when to commit, which files to commit and how the commit should be documented in a commit message.

To determine which files should be selected (or staged) for a new commit, the git add command is used followed by the name(s) of the files that should be committed or a . to stage all changes present in the current repository.

# Examplary staging command  
git add 'scripts/03\_propensity\_score\_analysis.R'

Once all files that should be committed in the same step are staged, the git commit command will create a local snapshot of the changes. The -m suffix and quoted text after the command represent the commit message, i.e., an informative yet brief comment on what changes were performed.

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

Every time a user makes a commit to create a snapshot of the work and made changes, Git creates a unique hash (a 40-character string created by a **simple hashing algorithm** [SHA]) which can be seen as an “ID” of the snapshot which enables user to comprehend every change made and revert back to any snapshot of the repository.

If any file(s) in the study repository should not be tracked and hence not be synchronized with the remote repository (e.g., because they include confidential information that should not be visible to others like database credentials), these files can be specified in the .gitignore file which needs to be located in the root directory of the repository. To create a .gitignore file, either auto-generate it upon initialization of the repository on GitHub (**Supplementary Figure 2**) or manually add a text file called .gitignore to the root directory.

[Figure 4](#fig-stage-commit-RStudio-gui) illustrates the equivalent step by using the RStudio GUI.

## 3.6 Synchronize changes with remote repository

### 3.6.1 Synchronize local changes to remote (push)

To synchronize the local changes made to our files, the next step involves synchronizing our commits with the remote repository on GitHub via the push command.

git push

In case we initialized the local repository as described in section [Section 3.3.2](#sec-gui-initialization-and-git-clone), a remote was already automatically configured. This can be checked via the following command.

git remote -v

If there is no remote established yet, this can be configured using the following commands.

git remote add origin https://github.com/USERNAME/rwe\_study.git  
git branch -M main  
git push -u origin main

This step is not needed if the intention of a researcher is just to keep track of local changes and there is no intention to share code, collaborate with others or work across different systems. However, most of the times researchers use Git to collaborate on a project and remote repositories such as GitHub can improve collaborative work significantly ([Figure 5](#fig-collaborate)).

### 3.6.2 Synchronize remote to local

In case collaborators have also made modification to their local files and pushed them to the remote repository, these changes can be easily synchronized with our local copy via the pull command.

git pull

Occasionally it can occur that two collaborators made conflicting changes to the same file or even to the same line of code. While Git is generally good at automatically integrating (merging) new changes, Git will notify the user to resolve a major conflict by editing the conflicting file if Git cannot automatically determine what is correct ((e.g., when one collaborator has made changes to a line of code and the other completely deleted it).

## 3.7 Tracking changes

One of the main reasons to adhere to this workflow is that in return we get rewarded with a detailed audit trail of changes made to our files. Probably, the best way to visualize and track changes is by browsing the commit history on the remote repository. **Supplementary Figure 4** illustrates this for the example commit we have previously performed in [Section 3.5.1](#sec-stage-and-commit-changes). A similar view for uncommitted changes is also provided in the RStudio GUI as displayed in [Figure 4](#fig-stage-commit-RStudio-gui).

For completeness, tracking uncommitted changes is also possible via the command line, but usually provides the user with a less intuitive output compared to what is visually provided by RStudio and GitHub.

git diff

A complete summary of committed changes with details on SHA hash, author, email, time and commit messages can be accessed via

git log

## 3.8 Branching

## 3.9 Keeping track of R software versions and paths

A major advantage of modern programming languages, particularly R and Python, 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 cohorts14 or propensity score matching15) 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.

### 3.9.1 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 system16,17. A more lightweight, yet effective way is by using the renv R package18, 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, 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, all that needs to be done is running the following command in R.

renv::restore()

Appending the session info to the end of an R script can give further context and information.

sessionInfo()

### 3.9.2 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.

new\_user\_cohort <- read.csv("C:\Users\Username\Projects\rwe\_study\protocol\icd\_codes.csv")

The use of absolute paths will make it impossible for anyone 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 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 rwe\_study). Applied to our example project, the following command would automatically reference the correct path and would be reproducible by any user re-running this script.

library(here)  
new\_user\_cohort <- read.csv(here("data", "icd\_codes.csv"))

# 4. Discussion

Also other GUIs to interact with Git…

Docker

Advantages of distributed VCS (Git):

* Full audit trail of which changes were made by whom, when and why
* Even if the code is not shared, Git makes it easier for an individual to go back to the code
* Git can serve as internal and external repository for code and made accessible to everyone, e.g. along with manuscript without copy-paste into word files or similar
* It is free and open-source and reliable
* Distributed VCS fully mirror entire repositories and hence every local clone is a full backup of the data

# 5. Tables

Create a table with most important git commands

# 6. Figures

|  |
| --- |
| Figure 1: Clone a remote repository using the RStudio graphical user interface. |

|  |
| --- |
| Figure 2: Minimal example of a transparent repository structure containing relevant study documents |

|  |
| --- |
| Figure 3: Overview of a basic Git workflow. |

|  |
| --- |
| Figure 4: RStudio provides a graphical user interface as an alternative to command line prompts to interact with Git and to perform command equivalent to git add, git commit and git push/pull (red circles). This view can be accessed in the right upper pane of RStudio under Git > Diff. In this example, an R script with the title ‘03\_propensity\_score\_analysis.R’ is selected and staged. All changes performed in this file can be viewed in the window below with green highlighted lines indicating additions and red highlighted lines deletions from the prior version. In this case, a title was added to a propensity score plot which is also expressed in the commit message. |

|  |
| --- |
| Figure 5: Git in connection with remote repositories (e.g., GitHub or GitLab) can signficantly improve collaboration across project members. |

# References

1. Wang SV, Pinheiro S, Hua W, *et al.* STaRT-RWE: structured template for planning and reporting on the implementation of real world evidence studies. *BMJ* 2021: m4856. doi:[10.1136/bmj.m4856](https://doi.org/10.1136/bmj.m4856).

2. Schneeweiss S, Rassen JA, Brown JS, *et al.* Graphical Depiction of Longitudinal Study Designs in Health Care Databases. *Annals of Internal Medicine* 2019; **170**: 398. doi:[10.7326/m18-3079](https://doi.org/10.7326/m18-3079).

3. Berger ML, Sox H, Willke RJ, *et al.* Good practices for real-world data studies of treatment and/or comparative effectiveness: Recommendations from the joint ISPOR-ISPE Special Task Force on real-world evidence in health care decision making. *Pharmacoepidemiology and Drug Safety* 2017; **26**: 1033–1039. doi:[10.1002/pds.4297](https://doi.org/10.1002/pds.4297).

4. Wang SV, Pottegård A, Crown W, *et al.* HARmonized Protocol Template to Enhance Reproducibility of hypothesis evaluating real-world evidence studies on treatment effects: A good practices report of a joint ISPE/ISPOR task force. *Pharmacoepidemiology and Drug Safety* 2022; **32**: 44–55. doi:[10.1002/pds.5507](https://doi.org/10.1002/pds.5507).

5. R Core Team. *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing, 2023. Available at: <https://www.R-project.org/>.

6. Almugbel R, Hung L-H, Hu J, *et al.* Reproducible Bioconductor workflows using browser-based interactive notebooks and containers. *Journal of the American Medical Informatics Association* 2017; **25**: 4–12. doi:[10.1093/jamia/ocx120](https://doi.org/10.1093/jamia/ocx120).

7. Mammoliti A, Smirnov P, Nakano M, *et al.* Orchestrating and sharing large multimodal data for transparent and reproducible research. *Nature Communications* 2021; **12**. doi:[10.1038/s41467-021-25974-w](https://doi.org/10.1038/s41467-021-25974-w).

8. Russell PH, Johnson RL, Ananthan S, Harnke B, Carlson NE. A large-scale analysis of bioinformatics code on GitHub. Qin Z (ed.). *PLOS ONE* 2018; **13**: e0205898. doi:[10.1371/journal.pone.0205898](https://doi.org/10.1371/journal.pone.0205898).

9. Bakken S. The journey to transparency, reproducibility, and replicability. *Journal of the American Medical Informatics Association* 2019; **26**: 185–187. doi:[10.1093/jamia/ocz007](https://doi.org/10.1093/jamia/ocz007).

10. García-Closas M, Ahearn TU, Gaudet MM, *et al.* Moving Toward Findable, Accessible, Interoperable, Reusable Practices in Epidemiologic Research. *American Journal of Epidemiology* 2023; **192**: 995–1005. doi:[10.1093/aje/kwad040](https://doi.org/10.1093/aje/kwad040).

11. Ram K. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine* 2013; **8**. doi:[10.1186/1751-0473-8-7](https://doi.org/10.1186/1751-0473-8-7).

12. Blischak JD, Davenport ER, Wilson G. A Quick Introduction to Version Control with Git and GitHub. Ouellette F (ed.). *PLOS Computational Biology* 2016; **12**: e1004668. doi:[10.1371/journal.pcbi.1004668](https://doi.org/10.1371/journal.pcbi.1004668).

13. Chacon S, Straub B. *Pro git*. Springer Nature, 2014.

14. Wickham H, Averick M, Bryan J, *et al.* Welcome to the tidyverse. 2019; **4**: 1686. doi:[10.21105/joss.01686](https://doi.org/10.21105/joss.01686).

15. Ho DE, Imai K, King G, Stuart EA. MatchIt: Nonparametric preprocessing for parametric causal inference. 2011; **42**. doi:[10.18637/jss.v042.i08](https://doi.org/10.18637/jss.v042.i08).

16. Boettiger C, Eddelbuettel D. An Introduction to Rocker: Docker Containers for R. *The R Journal* 2017; **9**: 527. doi:[10.32614/rj-2017-065](https://doi.org/10.32614/rj-2017-065).

17. 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](https://doi.org/10.32614/rj-2020-007).

18. Ushey K. Renv: Project environments. 2022. Available at: <https://CRAN.R-project.org/package=renv>.