# Introduction

This page describes the general RA guidelines we employ for our projects. The purpose of the guidelines is to make it easier to focus on doing replicable and properly documented work. The guidelines draw heavily on

- *Code and Data for the Social Sciences: A Practitioners Guide.* Gentzkow and Shapiro (2014). Link
  - The document contains a general introduction to reproducibility challenges that many social science researchers have faced and how the two authors have tried to solve them in their lab. Emphasizes coding conventions, directories, and version control.

- *Lab Manual.* Gentzkow and Shapiro. Link
  - This supplements their (2014) document. It includes information on workflows, coding, data handling, and paper and slide production. We generally follow their workflow approaches (e.g., using GitHub issues to assign and resolve tasks).

How to get started

- Read this document.

- Read Gentzkow and Shapiro (2014)
- Set up git on your computer and a GitHub account as described under `Version Control`.
- Install the necessary software on your local computer. Find the list and installation guides under `Software/Required`.
- Ask your supervisor to add you to the relevant projects on GitHub and Zotero.

# Workflow

## Tasks

- We follow Gentzkow and Shapiro's workflow approach.
- Tasks will be specified on GitHub under the relevant project as `issues`. We do this to keep track of open tasks, task notes and questions, and outputs. You can familiarize yourself with GitHub issues here.
  - A supervisor will add you to the relevant project(s) on GitHub.
- Each task will contain
  - A description.

  - A set of outcomes.
  - A task supervisor.
  - One or more task assignees.

- When working on a task

  - Keep documentation of your work. We suggest having a `running_notes_(your initials).md` document where you store your thoughts. A good practice is to add headlines with dates so that it becomes easier to go back to find thoughts related to a task you previously worked on.

- Asking questions

  - We encourage you to work independently but ask questions when you realize you are stuck. We all tend to get stuck very often when we start working with administrative data, so do come around and ask if something comes up!
  - If you ask about clarifications or questions related to a task in person, add a note about the questions and answers to the GitHub issue so we can track the progress.

- A task is closed by the task supervisor when

  - The relevant outcomes have been created/reached.
  - The task assignee has written a reply to the GitHub issue on how they completed the task and where relevant outcome files are located (e.g., the code file cleaning a bit of data or the note summarizing results).
  - The task supervisor agrees that the task is completed.

### Reporting and notes

- We write notes and documents in markdown format with files ending in `.md` unless another format is required.

  - Markdown documents can easily be compiled into Word, PDF (via LaTeX), HTML, beamer PDF slides, or other formats using `pandoc` or `quarto`.
  - Searching (and replacing) across multiple markdown files for content using typical text editors is easy.
  - Markdown files can be edited using most text editors. We suggest VSCode and Obsidian.
  - Introduction to basic markdown syntax written by the developers behind the original markdown language (it comes in many flavors).

- We keep personal running notes documents. These typically contain thoughts and drafts for notes and tasks. The document will typically be named `running_notes_` and end in our initials, e.g., `running_notes_je.md`.

### References

- We use Zotero to maintain shared libraries with project references.
  - A supervisor will add you to the relevant project(s) on Zotero.
- We use the `betterbiblatex` extension for Zotero to export `.bib` files to projects or to Overleaf.
- When adding a new reference to a Zotero project collection, pin the bibtex key.
  - Right-click the reference and select `better bibtex` -> `pin citekey`.
  - Remember to set up the `better bibtex` extension to use the correct citekey structure. See `Software / Required / Zotero`.

### Writing papers

- We use Overleaf to write papers and paper drafts in LaTeX format unless otherwise specified. Overleaf allows us to
  - Work on the paper at the same time
  - Integrate references from shared Zotero libraries

## Version Control

- We use the Git version control system to track changes to our notes and code. Git allows you to add, delete, or modify files, mark them as changed (committing), and finally add them to the project (pushing to the remote repository). All changes to files will then appear in the project Git history.
- We use GitHub to host our remote repositories for local, *non-sensitive* project files.
  - You can keep the local non-sensitive project files where you prefer.
  - You will pull and push changes to this remote directory to update general project files.

  - A supervisor will add you to the relevant project(s) on GitHub.
- We use a local Git repository for content stored on secure servers.
- Here, we all work from the same directory containing all relevant files simultaneously.
- A basic workflow for using Git is to Pull the latest version of project files from the remote repository.
  1. Make changes to the files you are working on.
  2. Commit your changes.
  3. Pull the latest updates from the remote repository and resolve any conflicts.

4. Push your changes to the remote repository.

- Guides
  - Basic (takes about 20 minutes total and well worth it): Git Handbook, Understanding the GitHub Flow, Mastering Issues, Mastering Markdown
  - Detailed: Pro Git, Version Control with Git, Chapters 4-9
- Setup: (based on https://github.com/gslab-econ/lab-manual/wiki/Setup)
  - Create a GitHub account and install the Git desktop/command line clients.

## Project structure

### Project storage

- We generally store non-sensitive project files in private GitHub project repository.
  - This ensures that all project participants have access to the relevant files.
  - It also makes it easy to keep track of changes to files and revert to previous versions if necessary. We use the `git` version control system to do this.
  - Supervisors will add you to the relevant project(s) on GitHub.
- Sensitive project files are stored on a secure KU server.
  - Supervisors can give you access to relevant folders.
- GDPR sensitive administrative data that we access via Statistics Denmark is stored on a secure Statistics Denmark server.
  - Documentation

  - Getting access requires the written permission from the project supervisor and the head of department.
  - Before getting access, you are required read the internal UCPH ECON and Statistics Denmark guidelines on working with sensitive data.
  - We follow the internal UCPH ECON guidelines on working with sensitive data, including what information can be downloaded from the secure server.

### Directories

- Our projects are typically distributed locally and on a secure server. The local project might contain notes, literature reviews, code that doesn't need to run on a secure server, and paper drafts. The secure server project location will exist when the project requires restricted access data (e.g., from Statistics Denmark).

- We generally apply the rules from Gentzkow and Shapiro (2014), chapter 4:
    1. Separate directories (folders) by function.
    2. Separate files into inputs and outputs (and temporary files)
    3. Make directories (folders) portable.
- Each (sub)component of a project should have its separate folder. A project with a literature review, presentation files, and a paper (draft) should contain at least those folders.

```
lit_review/
presentations/
paper/
```

- All components containing code should have at least a `code`, `temp`, and `out` folder. For example, assume that the simple project contains a simulation exercise written in R showing the consistency of an econometric estimator. The code file outputs the graph `simulate_estimator_consistency_distribution.PDF`. The folders could look like

```
consistency_simulation/
    code/
        simulate_estimator_consistency.R
    temp/
    out/
        simulate_estimator_consistency_distribution.PDF
lit_review/
presentations/
paper/
```

- We store raw data in a separate folder. Suppose we have more than one raw data set, for example, from Statistics Denmark and the Ministry of Education. We then separate them into subfolders with meaningful names and possibly a date of compilation so we can keep track of versions.

```
buildraw/
    dst/
        dat1.sas7bdat
        dat2.sas7bdat
consistency_simulation/
    code/
        simulate_estimator_consistency.R
    temp/
    out/
```

```
        simulate_estimator_consistency_distribution.PDF
lit_review/
presentations/
paper/
```

### Data formats

- When possible, store data in `.parquet` format.
  - We often work with large administrative data files that can take up many GB of space. We prefer the parquet format to reduce our server footprint and increase IO speed. This is a so-called columnar format, meaning that it is possible to load These are generally highly compressed
- We can import and export `.parquet` files using `R` and `python`. Hadley Wiggins, the developer behind the large parts of the `tidyverse` packages, has written an online book chapter on using `parquet` files with R.
- In R, we will often use the following syntaxes to import or export a dataset:

```
library(rio)
dat = import("builddata/out/clean_bef.parquet")
dat |> export("builddata/out/clean_bef.parquet")
```

## Code Conventions

### General

- Name scripts by what they do.
  - *Example*: Assume we have written some code that cleans the raw `BEF` register data for use in subsequent analyses. The file (sh)could be named `clean_bef.R`.
- Script outputs must contain the name of the producing script and be informative about the content.
  - *Example*: Assume the file `descriptives_main_sample.R` outputs two summary tables in LaTeX format. One is a balance table with means and differences in means between treatment and control groups, and one contains general summary statistics for the full sample. These (sh)could be named `descriptives_main_sample_balance.tex` and `descriptives_main_sample_summary.tex`.
- No line of code should be more than 100 characters long. All languages we work in allow you to break a logical line across multiple lines on the page (e.g., using `///` in Stata or `...` in Matlab). You may want your editor to show a “margin” of 100 characters.

- Functions should not typically be longer than 200 lines.

### R

- We follow Google's R Style Guide
- Exceptions to style guide:
  - Do not use dots, separate using underscores, and keep lowercase. Example: `.CalcMeans()` should be `calc_mean()`.

- Use the `rio` package for data import/export.
  - It supports many file types and generally uses the most efficient IO tool for importing/exporting the file format.
- Use the `data.table` package for data wrangling when possible.
  - Introduction to data.table from the authors of the package.
  - Introduction for Stata users
  - data.table chapter in Introduction to Data Science.
- Use the `fixest` package by Laurent Berge for estimating most types of statistical models, particularly linear and IV models with fixed effects, when possible.
  - It provides estimation tools typically much faster than other options in R and Stata.
  - Linear, fixed effects, and 2SLS models can be estimated via `feols()`.

  - Documentation.
- We typically use the `modelsummary` package for summarizing regression results and creating summary statistics tables outputted to latex or markdown format.
  - An introduction is available here.
  - When adding footnotes to `modelsummary()` tables, use the `footnote()` function from the `kableExtra` package with the options `escape = F` and `threeparttable = T`.

## Automation

- We automate everything that can be automated. This implies writing scripts to do all data cleaning, analysis, and table formatting, and using build tools to run these scripts in the correct order.
- We use the build tool `make` to run all project code.
  - Generally, we want to be able to delete all files in output folders, type `make all` on the command line, and have all output files reappear just as they were before.

### Build tool - `make`

- Build tools generally help run your project code in the correct order based on a set of rules that specify output targets and inputs, including code files. Some examples of more advanced build tools include `snakemake`, and `cmake`. I have generally found that these modern build tools are unavailable on the secure servers hosted at Statistics Denmark, where a substantial part of my project code resides.
- We, therefore, use the slightly more archaic `make`.
- Documentation for `make` is available here.
- Helpful introductions to `make` for data analysis
  - Automation and Make by Software Carpentry.
  - Makefiles for R/Latex projects by Rob Hyndman.
  - Minimal make by Karl Broman. Runs a couple of R scripts and creates a latex compiled PDF paper with the resulting figures.

  - GNU Make for Reproducible Data Analysis by Zachary Jones.
- An important feature of `make` is that it compiles or runs project code based on a general recipe, the `makefile`. The `makefile` consists of *targets*, *dependencies*, and *commands*, which together defines *rules*. A `makefile` can contain multiple rules. These rules can be linked, for example, if a rule uses the target of another rule as a dependency.
  - *Target*: The output file or goal you want to achieve. E.g., `builddata/out/clean_data.parquet`
  - *Dependencies*: Files or targets that must be up-to-date before executing the target's commands. This will typically include the code file you want to run, and the data it uses. E.g., `builddata/code/clean_data.R`, and `buildraw/out/rawdata.csv`.
  - *Command*: The command(s) that `make` will execute to create or update the target. `make` basically runs from the command line meaning that you must either use a command that is available from the CLI or specify the full path to the program. If, for example, you want to run an r script, you can use the CLI command `rscript` when this is installed.

```
#target: dependencies
#   command

builddata/out/clean_data.parquet: builddata/code/clean_data.R buildraw/out/rawdata.csv
    rscript builddata/code/clean_data.R
```

- Make `make` run your command
  1. Navigate a command line tool, such as `cmd` to the project folder where the makefile is located.
  2. In the command line, type `make target`, where target is the file you want to build. E.g., `make builddata/out/clean_data.parquet`.

- How make runs your command:
  - When you tell make to create a target, make first checks the dependencies for that rule. If a dependency is itself a target from another rule, makemoves back to this previous rule. This continuous until make finds the antecedent dependencies.
  - make then checks the timestamps of antecedent dependencies. If the dependencies have not changed since the target was last created, make won't re-run the commands for that target. If the target does not exist, the command is always run.
  - make will the move through the linked set of dependencies, running the commands from rules where dependencies have changed since the target was last created.
  - make will let you know before it tries to run if a dependency does not exist and there exists no rule to create it.
- Installation:
  - make is usually pre-installed on Unix-based systems (like macOS and Linux). For Windows, you can install it through tools like Cygwin, GNUWin32, or rtools.
- Important syntax notes:
  - *Indentation*: Make sure to use a tab, not spaces, for indentation in the Makefile.
  - *Multiple Languages*: If your workflow involves Stata or SAS scripts, you can include them in the Makefile just like R scripts. For instance, `stata -b do my_analysis.do` for a Stata script.
- Line splitting with many dependencies
  - If you file contains many dependencies it can be useful to split the dependency list over multiple lines. You can do this by writing \ and continuing the content on the next line after an indentation. If you include anything, include a space, after the backslash, make will throw an error.

```
builddata/out/clean_data.parquet: \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    rscript builddata/code/clean_data.R
```

- Many targets
  - A file may create multiple outputs, such as regression tables. To specify this, you simple add all targets to the left of :

```
builddata/out/clean_data1.parquet builddata/out/clean_data2.parquet : \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    rscript builddata/code/clean_data.R
```

- Automatic variables
  - `make` allows you to use automatic variables in rules.
    - `$@`: The name of the target
    - `$<`: The name of the first prerequisite
    - `$^`: The names of all the prerequisites, with spaces between them

```
builddata/out/clean_data.parquet: \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    rscript $<
```

- Creating variables
  - Creating variables can be useful, for example, for creating build rules or listing all targets you want to build.
  - Variables can be assigned with `:=`. E.g., `var := name1`
  - Variables can be called using `$(variablename)`
  - You can append to a variable by adding `$(variablename)` on the right hand side of the assignment name; `var := $(var) name2`.

```
R := C:\Users\bxn825\scoop\shims\rscript.exe

builddata/out/clean_data.parquet: \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    $(R) $<
```

- Creating build rules with many targets (phony targets)
  - The rule `all: $(targets)` is often used to run all rules you want in a project, when the variable `$(targets)` contains a set of targets in your project.
  - You can use `.PHONY` to explicitly declare `all` (and other non-file targets) as a phony target. This tells `make` that this target isn't a file but rather a label for a recipe to be executed.

```
R := C:\Users\bxn825\scoop\shims\rscript.exe

$(targets) := builddata/out/clean_data.parquet
builddata/out/clean_data.parquet: \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    $(R) $<
```

```
$(targets) := $(targets) builddata/out/clean_data.parquet
builddata/out/clean_data.parquet: \
    builddata/code/clean_data.R \
    buildraw/out/rawdata.csv
    $(R) $<

.PHONY: all

all: $(targets)
```

## Software

### Required

- Git
  - Git is a command-line version control software. See more under `Version Control`
  - Installation
    - Create a GitHub account and install the Git desktop/command line clients.
    - Installation via Scoop on Windows machines: `scoop install git`
- VSCode
  - VSCode is a general text editor you can use to edit markdown, Python, R, LaTeX, and many other file types.
  - It has excellent Git version control features included
  - Installation options:
    - Follow Jeppe Druedahl's guide to install VSCode.
    - Install using Scoop on a Windows machine with `scoop bucket add extras; scoop install vscode` (see `Software/Suggestions` on using ).
  - You can add functionality by installing extensions. Some extensions you'll likely want to install:
    - Git History
    - GitHub Copilot (free for academic users)
    - LaTeX workshop
    - Markdown All in One
    - Python
    - Quarto
    - R (consider using this together with `radian`)
    - Stata Enhanced
    - vscode-pandoc

  - Guides:
    - Markdown and Visual Studio Code
  - We use `pandoc` or `quarto` to convert markdown files into Word, PDF (via LaTeX), beamer slides, or HTML files.

- R, RStudio, rtools
  - R is an open-source statistical programming language that has gotten a lot of traction in the Econometrics community. Most new developments in Econometrics are likely to arrive to R at the same time or prior to, e.g., Stata.
  - Rstudio is an editor specialized for R coding. We typically use RStudio whenever we edit R code.

  - rtools is a set of software tools R will need to compile some packages, including the `arrow` package we use for parquet format IO.
  - Installation options:
    - [Installation guide for R and RStudio] by Posit (the developers behind RStudio).
    - Installation guide for rtools via CRAN.
    - Installation via Scoop on Windows machines: `Scoop bucket add r-bucket https://github.com/cderv/r-bucket.git; scoop install r; scoop install rstudio; scoop install rtools`

- Pandoc
  - Pandoc is an open-source command line tool that can be used to convert between many different text formats. We will typically use it to convert between markdown and PDF/word documents.
  - Installation options:
    - Guide from the developers. Installation via Scoopon Windows machines: `scoop install pandoc`

- Quarto
  - Quarto is a piece of software developed by Posit that allows you to write quarto documents containing both markdown text and integrated R, Python, or some other relevant code. The documents can be edited in RStudio or VSCode. Quarto documents are beneficial when some code requires extensive documentation.

  - Installation options:
    - Guide from the developers.
    - Installation via Scoop on Windows machines: `scoop install quarto`

- Zotero
  - Zotero is an open-source reference manager.

  - Allows easy export of bibtex reference libraries that can cited in markdown and latex documents.

  - Installation options:
    - Guide from the developers.
    - Installation via Scoop on Windows machines: `scoop install zotero`
  - Install relevant extensions
    - better bibtex
      - Install using the guide at the link.
      - Set bbt's citation key formula to `authEtAl + year + shorttitle(3,3)` by going to `preferences --> Better BibTex --> Open Better BibTex Preferences --> Citation Key`, and copy the formula in.

## Suggestions

- Scoop
  - A Powershell tool for Windows that helps you install and update software updated.
  - Installation options:
    - Guide from the developers.
  - Working on university-provided IT equipment can give update and installation problems if you do not have administrator rights over the computer. This can be circumvented by ensuring that all (or most) programs are installed in your own user path. Scoop does this.
  - We presently use the Powershell tool `scoop` to manage the installation of most software on my system. `scoop` uses recipes created by others to install (often) the latest versions of programs.
  - Installing a program with `scoop` is as simple as `scoop install program`.
  - Updating a program with `scoop` is as simple as `Scoop update program`. Type `scoop update *` to update all installed programs.
  - Scoop searches for install recipes in buckets. Buckets can be added by typing `scoop bucket add ...` in Powershell. Examples of useful buckets include `extras`, `nerd-fonts`, and `r-bucket`. You can find an example of installing and using a `scoop` to set up a new Windows machine at `https://github.com/EriksenJ/_setup`.

- Obsidian
  - A markdown-based digital note editor with latex compilation to PDF ready out of the box.

  - Some suggested plugins
    - LaTeX Suite

        - Linter
        - Obsidian Git
        - Templater
        - Zotero Integration
    - Install with `scoop` on windows: `scoop bucket add extras; scoop install obsidian`
- Linh T. Tô has a great set of (free) resources on her website.