Best Practices in Computing

Francisco Villamil

Applied Quantitative Methods II

MA in Social Sciences, Spring 2026

Today's goals

- Understand why computing practices matter for research
- Learn to organize a research project with a clear folder structure
- Write better R code: functions, checks, and style
- Understand the role of plain text and command line tools
- Deepen your knowledge of version control with Git

This is a practical session — a breather between the heavy methods content of the first four weeks and the panel data material coming next. The skills covered today are not statistical, but they are essential for doing good empirical work. Students who have struggled with organizing their assignments or tracking down bugs in their code will find this session especially useful. The goal is not to master all of these tools in one session, but to establish good habits early that will pay off for the rest of the course and beyond.

Roadmap

Why computing practices matter

- **Reproducibility**: can you (or someone else) re-run your analysis?
- **Avoiding errors**: manual steps introduce mistakes
  - → Copy-pasting results into Word
  - → Renaming files by hand
  - → Running scripts in the wrong order
- **Efficiency**: time invested in workflow saves time later
- On errors: Bisbee et al. (2022)

Start by motivating why we are spending a whole session on computing practices. The reproducibility crisis in social science is partly a computing problem: researchers cannot reproduce their own results because they lost track of which script produced which output, or because they made manual changes that were never recorded. The link at the bottom is a paper documenting how common coding errors are in published research, including errors introduced by tools like stargazer that silently reorder variables. Emphasize that these are not hypothetical problems — they happen to experienced researchers regularly.

You come back to a project after 6 months.

You need to update one figure.

How long does it take you?

Discussion prompt. Ask students to think about their own experience. Most will admit that going back to an old project is painful: they do not remember which script produces which output, what the file names mean, or what order to run things in. The goal of this session is to make the answer to this question "a few minutes" instead of "a whole day." A well-organized project with a Makefile and clear folder structure can be re-run with a single command. A poorly organized project requires archaeology.

Roadmap

The key principle: separate tasks into folders

- Each **task** gets its own folder with its own script and output
- A task is a self-contained step in the pipeline:
  - → Data collection / cleaning
  - → Analysis (models, tables)
  - → Plots and figures
  - → Final document (paper, slides)
- Each folder has:
  - → A script (e.g., `analyze.R`)
  - → An `output/` subfolder for results
- **Never** one giant script that does everything

The single most important organizational principle: break the project into discrete tasks, each in its own folder. This makes the project navigable (you know where to look for each piece), modular (you can re-run one step without re-running everything), and debuggable (when something breaks, you know which script to check). The output subfolder is key: it creates a clear separation between code and results, and makes it obvious which files are generated (and therefore reproducible) versus which are source files. Contrast this with the common student pattern of having one enormous R script with 500 lines that loads data, cleans it, runs models, and makes plots all in one go.

Example: `workflow_example` project

- Full example: `github.com/franvillamil/workflow_example`

Walk through the folder structure. Each folder is a task: create_data generates the dataset, analyses runs models and produces a LaTeX table, plots creates figures, and document compiles the final paper. Each task folder has a script and an output subfolder. The Makefile ties everything together (we will see it shortly). The .gitignore tells Git which files to ignore. The key insight: the document folder does not contain any R code, and the analysis folders do not contain any LaTeX. Each piece does one thing. This is the structure students should aim for in their assignments and final projects.

File naming conventions

Bad	Good
`Final Data.csv`	`data_cleaned.csv`
`Datos educación.csv`	`datos_educacion.csv`
`analysis.R (which one?)`	`01_clean_data.R`
`figure1final2.pdf`	`fig_scatter_income.pdf`
`My Thesis Draft (3).docx`	`thesis_draft.tex`

- **No spaces** in file names (use underscores or hyphens)
- **No special characters** (accents, symbols)
- Use **numbered prefixes** for scripts that run in order
- Use **descriptive names**: what is in the file, not when you made it

File naming seems trivial but causes real problems. Spaces in file names break command-line tools and require quoting in scripts. Special characters cause encoding issues across operating systems. Numbered prefixes (01_, 02_) make the execution order obvious. Descriptive names mean you can find the right file without opening it. The "Final Data (3).csv" pattern is a sign of a disorganized project. If you need version control, use Git, not file names. The goal is that anyone (including your future self) can look at the file listing and understand what each file does.

Integrating R and LaTeX

- R produces output files (tables, figures)
- LaTeX `\input{}` or `\includegraphics{}` reads them
- Example from `workflow_example`:
  - → `analyze.R` → `analyses/output/table_models.tex`
  - → `plots.R` → `plots/output/scatter.pdf`
  - → `doc.tex`: `\input{../analyses/output/table_models}`
  - → `doc.tex`: `\includegraphics{../plots/output/scatter}`
- **No copy-pasting**: update the analysis, recompile the document
- The document always reflects the latest results

This is one of the biggest practical payoffs of good project organization. When your LaTeX document reads tables and figures directly from the output folders, you never have to manually update them. Change a model specification? Re-run the R script, recompile the document, and everything updates automatically. This eliminates a major source of errors: the table in the paper not matching the actual analysis because you forgot to update it after a change. Students who are used to R Markdown get some of this automatically, but the folder-based approach is more flexible for larger projects where you want to separate analysis from writing.

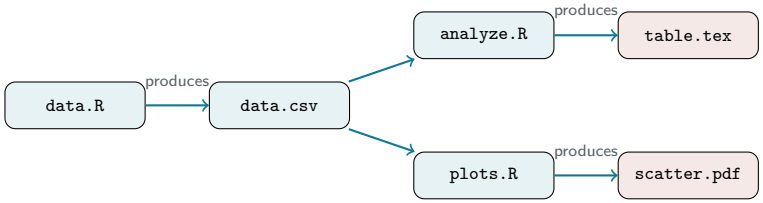
Automating the pipeline: Makefiles

```
all:  create_data/output/data.csv analyses/output/table_models.tex \
    plots/output/scatter.pdf
create_data/output/data.csv:  create_data/data.R
    Rscript --no-save create_data/data.R
analyses/output/table_models.tex:  analyses/analyze.R \
        create_data/output/data.csv
    Rscript --no-save analyses/analyze.R
plots/output/scatter.pdf:  plots/plots.R \
        create_data/output/data.csv
    Rscript --no-save plots/plots.R
```

Note: Makefiles require **tabs** (not spaces) for indentation

Walk through the Makefile line by line. The first rule (all) lists all the final outputs. Each subsequent rule has a target (the output file), dependencies (the script and any input files), and a recipe (the command to run). The key feature: Make only re-runs a step if its dependencies have changed. If you modify plots.R but not analyze.R, only the plots step re-runs. This saves time on large projects and ensures consistency. To run everything: type `make` in the terminal. To run just one step: `make plots/output/scatter.pdf`. Makefiles require tabs (not spaces) for indentation — this is a common source of errors. Reference: `makefiletutorial.com`.

Makefiles: the logic



- `make` only re-runs what has **changed**
- The dependency graph ensures correct ordering

This diagram shows the dependency graph that Make constructs from the Makefile. Each arrow represents a dependency: the analysis script depends on the data, and the table depends on the analysis script. If you change data.R, Make knows it needs to re-run data.R, then analyze.R, then plots.R (because they all depend on data.csv). If you only change plots.R, only the plots step re-runs. This is the power of Make: it tracks dependencies automatically. For small projects, running everything manually is fine. For larger projects (a dissertation with 10 scripts), Make saves hours and prevents errors from running things in the wrong order.

Roadmap

DRY: Don't Repeat Yourself

- If you write the same code **twice**, write a **function**
- Why?
  - → Fix a bug in one place, not ten
  - → Easier to test and debug
  - → Shorter, cleaner scripts
- Common candidates for functions:
  - → Cleaning steps applied to multiple datasets
  - → Running the same model with different variables
  - → Producing plots with consistent formatting

The DRY principle is the single most important coding habit. Whenever you find yourself copying and pasting code and changing one or two things, you should write a function instead. The reason is not just aesthetics: duplicated code means duplicated bugs. If you find an error in the cleaning step for one dataset, you have to remember to fix it in all the copies. With a function, you fix it once. Functions also make your code self-documenting: `clean_survey(df)` is more readable than 20 lines of dplyr operations. Encourage students to start small: even a function that just wraps a few lines is better than copy-pasting.

Define constants at the top

Bad	Good
`df = df[df$year >= 2000, ]`	`start_year = 2000`
`...`	`...`
`df2 = df2[df2$year >= 2000, ]`	`df = df[df$year >= start_year, ]`
	`df2 = df2[df2$year >= start_year, ]`

- Put parameters and thresholds at the **top of the script**
- Change once, applies everywhere
- From the example project:
  - → `n_obs = 1000`
  - → Used throughout — change it once to re-run with different sample size

This is a simple but powerful habit. Any number or string that appears more than once in your script should be defined as a constant at the top. Year cutoffs, sample size, file paths, variable names for analysis — all of these should be constants. This way, when you need to change a parameter (e.g., switch from 2000 to 1990), you change it in one place. The alternative — searching through a 200-line script for every occurrence of "2000" — is error-prone and tedious. In the workflow_example project, `n_obs = 1000` is defined once and used in the data generation step. Changing it to 5000 requires editing one line.

Write checks and assertions

```
# After merging two datasets
merged = merge(df1, df2, by = "id")
if(nrow(merged) != nrow(df1)) {
  stop("Merge changed number of rows!")
}

# Check for duplicates
if(any(duplicated(df$id))) {
  stop("Duplicate IDs found!")
}

# Sanity check on values
if(any(df$age < 0 | df$age > 120, na.rm = TRUE)) {
  warning("Suspicious age values detected")
}
```


Assertions are the best defense against silent errors. The idea: after every operation that could go wrong, add a check that verifies the result is what you expect. Merges are the classic example: a merge that unexpectedly creates duplicates can silently inflate your sample size and bias your results. The `stop()` function halts execution with an error message — use it when something is definitely wrong. The `warning()` function prints a warning but continues — use it when something is suspicious but might be okay. These checks cost nothing to write and can save you from publishing wrong results. Make it a habit to add a check after every merge, filter, or reshape operation.

Print diagnostics as you go

- Sprinkle `print()` and `cat()` throughout your scripts
- Examples:
  - → `cat("Rows after cleaning:", nrow(df), "\n")`
  - → `print(table(df$treatment))`
  - → `cat("Missing values:", sum(is.na(df$outcome)), "\n")`
- When the script runs, you get a **log** of what happened
- If something goes wrong later, the log tells you where
- This is especially valuable when running scripts via `Rscript` or `make`

Print statements are the simplest form of logging. When you run a script interactively in RStudio, you can inspect objects as you go. But when you run scripts via the command line (which is what Make does), you do not see intermediate results unless you print them. Adding print statements that show sample sizes, value distributions, and missing data counts creates a log that you can review to make sure everything went as expected. This is especially important for long-running scripts that process multiple datasets. If the final output looks wrong, the print log helps you narrow down where the problem occurred.

Code style matters

- Use **meaningful variable names**:
  - → `pop_2020` not `x1`, `income_log` not `v`
- Be **consistent**:
  - → Pick `snake_case` and stick with it
  - → Consistent indentation (2 spaces in R)
- Write **comments** for non-obvious code
  - → Why, not what: `# Drop obs before 1990 (pre-reform)`
  - → Use section dividers: `# ============`
- Your code is read more often than it is written

Code style is about making your code readable — for others and for your future self. Meaningful variable names eliminate the need for comments explaining what `x1` is. Consistent formatting (indentation, spacing, naming conventions) makes the code scannable. Comments should explain the reasoning behind non-obvious decisions, not narrate what the code does line by line. The section divider pattern (a line of equals signs or dashes) helps break long scripts into navigable chunks. The R community has adopted snake_case as the standard; avoid dots in names (data.frame style) and camelCase (Java style). Reference: Hadley Wickham's R Style Guide.

Roadmap

Why plain text?

- Plain text files: `.R`, `.tex`, `.csv`, `.md`, `.py`
- Binary files: `.docx`, `.xlsx`, `.dta`, `.pdf`
- Plain text advantages:
  - → **Portable**: works on any computer, any OS, forever
  - → **Version control**: Git tracks changes line by line
  - → **Scriptable**: can be processed by other tools
  - → **No vendor lock-in**: does not require specific software
- You are already using plain text: R scripts, LaTeX files
- The goal: use it for **as much as possible**

Plain text is the foundation of a reproducible workflow. A .tex file from 1990 still compiles today; a Word 95 .doc file may not open in modern Word. Git can show you exactly which line changed between two versions of a .R file; for a .docx file, it can only tell you the file changed. CSV files can be read by any programming language; .xlsx files require special libraries. The point is not that binary formats are bad — sometimes you need them (e.g., images, compiled PDFs). The point is that your source materials (code, text, data when possible) should be in plain text so they are portable, versionable, and scriptable. Reference: Kieran Healy's *The Plain Person's Guide to Plain Text Social Science* at `plain-text.co`.

Code editors

- RStudio is fine for R, but consider a **general-purpose editor**
- Options:
  - → **VS Code**: free, huge ecosystem of extensions, very popular
  - → **Positron**: new IDE by Posit (RStudio makers), for R and Python
  - → **Sublime Text**: fast, lightweight, highly customizable
- Why use a general editor?
  - → Edit R, LaTeX, Python, Makefiles in the same tool
  - → Better project navigation and search
  - → Customizable snippets and shortcuts

Most students in the course use RStudio, which is a perfectly good tool for R. But as they start working with LaTeX, Makefiles, and possibly Python, having a general-purpose editor becomes valuable. VS Code is currently the most popular editor and has excellent R support through extensions. Positron is a new option from the same company that makes RStudio, designed for both R and Python. Sublime Text is lightweight and fast. The key message is not "stop using RStudio" but "be aware that other tools exist and might serve you better as your workflow becomes more complex." Emphasize that the choice of editor is personal — what matters is being comfortable and productive with your tool.

The command line: basics

- The command line is how you talk directly to your computer
- Essential commands:
  - → `cd` — change directory
  - → `ls` — list files
  - → `pwd` — print working directory
  - → `mkdir` — create a directory
  - → `Rscript file.R` — run an R script
  - → `make` — run a Makefile
- You **already use it** for Git (`git add`, `git commit`, `git push`)
- Terminal on Mac/Linux, Git Bash or WSL on Windows

Many students are intimidated by the command line, but they have already been using it for Git since Session 1. The goal here is not to turn them into command-line experts, but to make them comfortable with basic navigation and script execution. Being able to `cd` into a project folder and type `make` is all they need for the workflow we are teaching. The six commands listed here cover 90% of what a social science researcher needs. For those who want to go deeper, point them to MIT's Missing Semester course (`missing.csail.mit.edu`) and Software Carpentry's Unix Shell lesson.

How do you currently organize your R projects?

One script or many? How do you keep track of output?

Discussion prompt. This is a good moment to pause and let students share their current practices. Some will have one giant script, others will have started organizing into folders. Use their answers to connect back to the principles covered: separating tasks, naming files well, using output subfolders. This also helps calibrate the rest of the lecture: if most students are already using good practices, you can move faster; if most are struggling, spend more time on the basics.

Roadmap

Git: recap from Session 1

- You have been using Git since the first session
- The basic workflow:
  - → `git add file.R` — stage changes
  - → `git commit -m "message"` — save a snapshot
  - → `git push` — upload to GitHub
  - → `git pull` — download from GitHub
- Today: going deeper into **good practices**

Quick recap of what students already know. By this point they should be comfortable with the basic add-commit-push cycle. Today we go deeper into the practices that make Git actually useful rather than just a backup tool. The three topics: .gitignore (what not to track), commit messages (how to write useful history), and GitHub for sharing replication materials.

The `.gitignore` file

Track (source files):

- R scripts (`.R`)
- LaTeX source (`.tex`)
- Makefiles
- Documentation (`.md`)
- Small data files (`.csv`)

Do NOT track:

- Generated output (`.pdf`, `.png`)
- LaTeX auxiliary files (`.aux`, `.log`)
- Large data files
- System files (`.DS_Store`)
- Sensitive data

`.gitignore`:

```
*.pdf
*.aux
*.log
.DS_Store
output/
```


The .gitignore file tells Git which files to ignore. The principle: track source files, not generated files. If a file can be reproduced by running a script, it does not need to be in Git. This keeps the repository small and clean. LaTeX auxiliary files (.aux, .log, .synctex.gz) are generated every time you compile and should always be ignored. Output folders (tables, figures) are generated by R scripts and should also be ignored. Large data files should not be in Git because Git is not designed for binary files and the repository will grow very quickly. System files like .DS_Store (Mac) and Thumbs.db (Windows) should always be ignored. Show students the .gitignore from the workflow_example project as a minimal example.

Good commit habits

- Commit **often**, in small logical units
  - → Not: one commit per day with everything
  - → Yes: one commit per completed task or fix
- Write **meaningful commit messages**:
  - → Bad: `"update"`, `"changes"`, `"asdf"`
  - → Good: `"Add robust SE to main models"`
  - → Good: `"Fix merge bug in cleaning script"`
- Your commit history is a **lab notebook**
  - → You should be able to reconstruct what you did and why

The commit message is the most underappreciated part of Git. Good messages make the history useful: you can look back and see when you added a feature, when you fixed a bug, and what the reasoning was. Bad messages ("update", "stuff", "final") make the history useless. The rule of thumb: the message should complete the sentence "This commit will..." So "Add robust SE to main models" reads as "This commit will add robust SE to main models." Committing often in small units means each commit is easy to understand and easy to revert if something goes wrong. One giant commit that changes 15 files is hard to review and hard to undo.

GitHub for replication materials

- Journals increasingly require **replication packages**
- GitHub is the standard platform for sharing code:
  - → Public repository with all scripts
  - → README explaining how to reproduce results
  - → Data (if shareable) or instructions to obtain it
- Good examples in political science:
  - → Most published papers now have a GitHub repo
  - → Look at how others structure their replication packages
- Your `workflow_example`-style project is already a replication package

The connection between project organization and replication: if your project is well-organized with separate task folders, a Makefile, and clear naming, it is already a replication package. You just need to push it to GitHub and add a README. Journals like the APSR, AJPS, and JOP now require replication materials as a condition of publication. Having good workflow habits from the start means you do not have to scramble to create a replication package after the paper is accepted. The README should explain: what software is needed, what data to download, and how to run the scripts (ideally just `make`). Point students to the workflow_example repo as a minimal template they can adapt.

Collaboration with Git

- Git was designed for collaboration
- Basic collaboration workflow:
  - → Both collaborators clone the same repo
  - → Each works on their own computer
  - → `push` and `pull` to sync
- **Merge conflicts**: when two people edit the same line
  - → Git asks you to resolve manually
  - → Rare if you communicate and divide tasks well
- Even for solo work: syncing between laptop and desktop

Briefly cover collaboration because some students will co-author papers or work on group projects. The basic model is simple: both people push to and pull from the same GitHub repository. Merge conflicts happen when two people edit the same part of the same file, which Git cannot resolve automatically. In practice, merge conflicts are rare if collaborators divide the work by files (one person works on the analysis script, another on the plots script). For solo researchers, Git is still useful for syncing between multiple computers — much more reliable than Dropbox for code projects, because Dropbox can create conflicting copies of files. Do not go deep into branches and pull requests — that is beyond the scope of this course.

Roadmap

Key resources

- Kieran Healy, *The Plain Person's Guide to Plain Text Social Science*:
  → `plain-text.co`
- MIT, *The Missing Semester of Your CS Education*:
  → `missing.csail.mit.edu`
- Software Carpentry lessons (Unix Shell, Git):
  → `software-carpentry.org/lessons`
- Bruno Rodrigues, *Building Reproducible Analytical Pipelines with R*:
  → `raps-with-r.dev`
- Example project: `github.com/franvillamil/workflow_example`

These are all free online resources. Healy's guide is the most directly relevant: it covers plain text, LaTeX, R, Git, and project organization from a social science perspective. The MIT Missing Semester is broader (covers the command line, editors, scripting, and more) but extremely well taught. Software Carpentry has hands-on lessons designed for researchers with no programming background. Rodrigues' book covers reproducible pipelines in R specifically, including Makefiles, Docker, and functional programming. The workflow_example repo is a minimal but complete example they can fork and adapt for their own projects.

Summary

- **Organize**: separate tasks into folders, each with script + output
- **Automate**: use Makefiles to run the pipeline
- **Integrate**: R output feeds directly into LaTeX documents
- **Code well**: DRY, constants at the top, checks and assertions
- **Print diagnostics**: log what your scripts do
- **Use plain text**: portable, versionable, scriptable
- **Use Git**: commit often, write good messages, share via GitHub

Recap the main messages. The overarching theme is: invest a little time in your workflow now, save a lot of time (and errors) later. None of these practices are difficult on their own; the challenge is building them into habits. Encourage students to start with one change (e.g., separate their next assignment into task folders) and gradually adopt more practices. By the end of the course, their final projects should follow the workflow_example structure.

For next week

- Complete Assignment 5
- Next session: Panel Data
  - → Fixed effects and within-group variation
  - → The `fixest` package in R

Brief logistics. Assignment 5 will ask students to apply the computing practices from today: organize a small project with proper folder structure, write a Makefile, and use Git. Next week returns to methods content with panel data, which is one of the most important tools in applied social science. Students should review the basics of panel/longitudinal data structure before the session.

Questions?