04 - Reproducibility and Version Control

ml4econ, HUJI 2025

Itamar Caspi

April 20, 2025 (updated: 2025-04-20)

Replicating this Presentation

R packages used to produce this presentation

```
library(tidyverse)  # for data wrangling and plotting
library(tidymodels) # for modeling the tidy way
library(knitr)      # for presenting tables
library(xaringan)   # for rendering xaringan presentations
```

From Best Practices to Research Methodology

Good research isn't just about the right model or dataset—it's about building a system where your entire process is transparent, reproducible, and scalable. We're not just replacing tools; we're redefining methodology

Traditional Best Practice	**Modern Methodological Approach**
High-dimensional statistics	Machine learning
Inline code comments	Literate programming (Notebooks: Rmd, Jupyter)
`mydoc_1_3_new_final_23.docx`	Version control (Git, GitHub/GitLab)
Manually curated Excel tables	Programmatically generated tables (SQL, dplyr, pandas)
Ad hoc scripts and file sharing	Fully reproducible workflows (e.g., Make, Snakemake, R targets)
Stata, SAS, EViews	R, Python, Julia
Working in isolation	Interdisciplinary, collaborative teams

Why This Matters

Modern research demands more than good analysis.

- ✅ Transparent
- ✅ Reproducible
- ✅ Collaborative
- ✅ Scalable

We're moving from ad hoc habits to engineered workflows.
Not just new tools — a new research methodology.

Outline

1. Reproducibility
2. The Tidyverse
3. Version Control
4. GitHub

RStudio Projects

Reproducibility: From Principle to Practice

Reproducible research

Enables *anyone* — including future-you — to re-run your code and get the same results.

What it requires:

- Clear documentation (what you did *and* why)
- All dependencies listed (packages, versions, data sources)
- Environment details (e.g., R version, OS, random seeds)

Mindset shift:

Write for the *next person* who reads your code — especially if that person is you, six months from now.

Robust Packages Use: renv

- renv package: Create reproducible environments for R projects
- Key benefits:
  - *Isolated:* Private package library for each project
  - *Portable:* Easily transfer projects across computers and platforms
  - *Reproducible:* Records exact package versions for consistent installations
- Learn more: Introduction to renv

RStudio Project-Oriented Workflow

Avoid common anti-patterns:

- `setwd()` — hardcodes paths and breaks portability
- `rm(list = ls())` — hides underlying issues in code structure

Recommended practices:

1. Use RStudio Projects
    - Keeps all files relative to a self-contained working directory
    - Supports portability and collaboration
2. Configure your R environment for reproducibility
    - Go to: `Tools → Global Options → General`
    - Set "Save workspace to .RData on exit" to NEVER
    - Uncheck "Restore .RData into workspace at startup"

Why Project-Oriented Workflows Matter

1. Reproducibility
Your code should run from start to finish on any machine, with no manual tweaks.

2. Transparency
Avoid hidden state and side effects — make your logic explicit and inspectable.

3. Portability
Relative paths and project directories ensure your work is easily shared or moved.

4. Collaboration
Team members (and your future self) can understand and run your project without reverse-engineering your environment.

5. Version control compatibility
Project-based organization plays well with Git, enabling clean tracking of changes and smooth collaboration.

R Markdown: A Tool for Reproducible Research

- Integrates text, code, figures, tables, and citations in one document
- Always starts from a clean R session when “knitted” — ensures reproducibility
- Outputs to multiple formats:
  - Documents: HTML, PDF, Word
  - Presentations: Beamer, PowerPoint, Xaringan
  - Websites, books, articles, dashboards (`blogdown`, `bookdown`, `pagedown`, `flexdashboard`)
- Ideal for communicating analysis, sharing workflows, and creating dynamic documents
- A modern alternative: **Quarto**
  - Supports R, Python, Julia; more flexible and extensible

The Tidyverse

This is Not a Pipe

%>% Is a Pipe

- `%>%` is the *pipe operator* from the `magrittr` package (core to the `tidyverse`)
- It passes the result of one function into the next — like saying "and then"

Example (equivalent code):

Traditional: `y <- h(g(f(x), z))`

Piped version: `y <- x %>% f() %>% g(z) %>% h()`

Read as: "Take `x`, then apply `f()`, then apply `g()` with `z`, then apply `h()` — save as `y`"

This expresses a series of transformations in the order you think about them, improving both readability and debugging.

Morning Routine

```
leave_house(get_dressed(get_out_of_bed(wake_up(me, time =
"8:00"), side = "correct"), pants = TRUE, shirt = TRUE), car
= TRUE, bike = FALSE)
```

```
me %>%
  wake_up(time = "8:00") %>%
  get_out_of_bed(side = "correct") %>%
  get_dressed(pants = TRUE, shirt = TRUE) %>%
  leave_house(car = TRUE, bike = FALSE)
```

Source: https://twitter.com/andrewheiss/status/1359583543509348356?s=20

Base R vs. the Tidyverse

Given this data frame:

```
df <- data.frame(
  x = rnorm(10),
  y = rnorm(10),
  z = rnorm(10)
)
```

What does this base R code do?

```
df_new <- df[df$x > 0, c("x", "y")]
df_new$xx <- df_new$x^2
```

Next: Let's rewrite this with `dplyr` for clarity and readability.

"Pipped" code?

How about this one?

```
df_new <- df %>%
  select(x, y) %>%
  filter(x > 0) %>%
  mutate(xx = x^2)
```

The above code chunk should be read as:

> "Take `df`, then select `x` and `y`, then keep rows where `x > 0`, then add a new column `xx = x^2`."

- The native pipe operator `|>` is available in R 4.1.0 and later

```
df_new <- df |>
  select(x, y) |>
  filter(x > 0) |>
  mutate(xx = x^2)
```

Pros & Cons

- Adopting a tidyverse approach improves readability ⇒ and with it, reproducibility
- There's a growing consensus in the R community that we should **learn the tidyverse first**
- But: the tidyverse is utopian by design - It evolves rapidly in pursuit of elegance and consistency
- Base R, by contrast, is stable and enduring - Built to last — sometimes clunky, but reliable
- Bottom line: Be fluent in both — tidy tools for clarity, base tools for durability

Enter GenAI

Write R code that takes df, then select x and y, then keep rows where x > 0, then add a new column xx = x^2.

Here's a concise way to do this with **dplyr**:

r Copy Edit

```r
library(dplyr)

df2 <- df %>%
  select(x, y) %>%    # keep only x and y
  filter(x > 0) %>%   # keep rows where x > 0
  mutate(xx = x^2)    # add xx = x^2
```

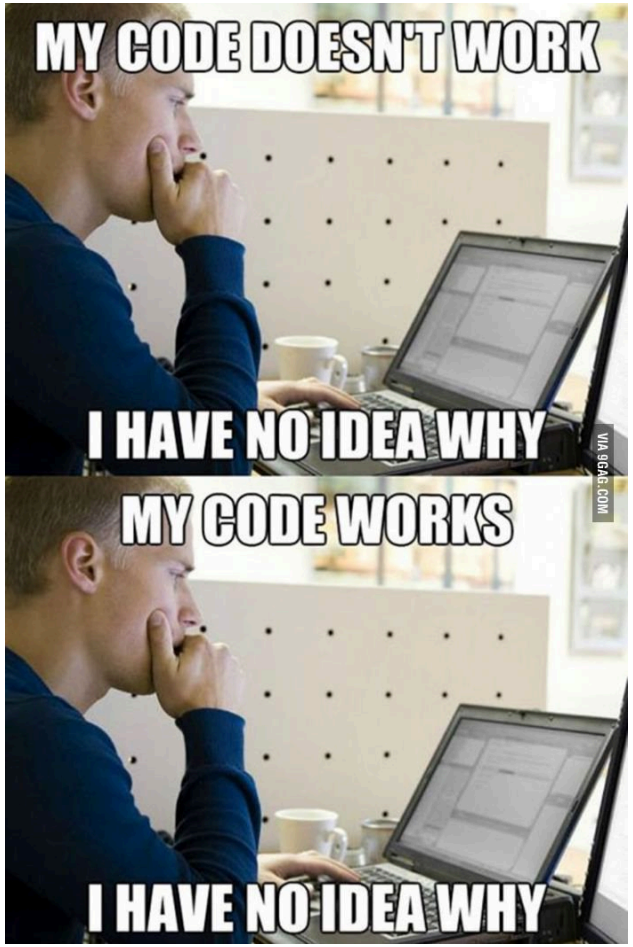
If you prefer **base R**, you can achieve the same in two steps:

r Copy Edit

```r
# 1. subset to x, y and x > 0
df_sub <- subset(df, x > 0, select = c(x, y))

# 2. add xx column
df_sub$xx <- df_sub$x^2
```

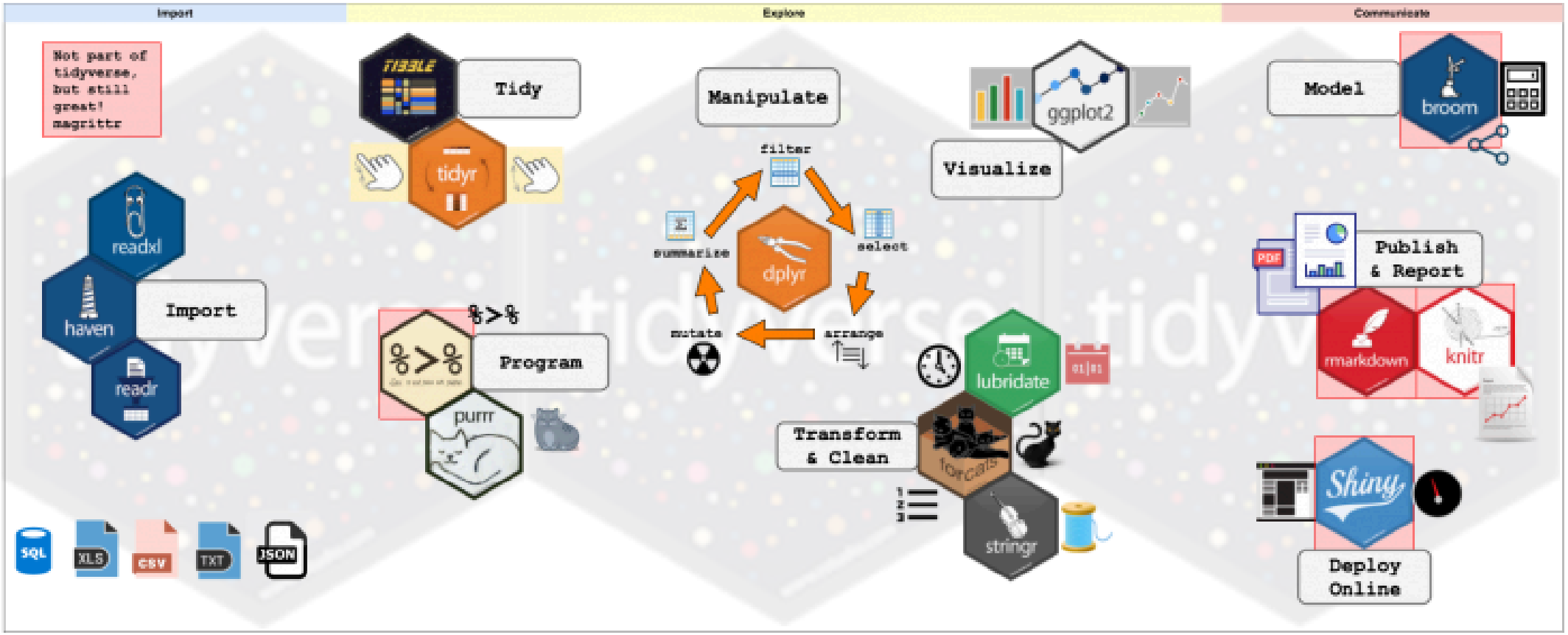
Let me know if you'd like any variations or explanations!



Vibe Coding

- What is Vibe Coding? A term popularized by Andrej Karpathy, *vibe coding* is the practice of using natural language to prompt AI tools to generate code — letting you focus on "what" rather than "how."
- How it works:
  - Describe what you want in plain language
  - Let an AI assistant (like ChatGPT) generate the code
  - Test, refine, and adapt the output to fit your goals
- Connection to Reproducibility:
  - Pros: Accelerates the creation of reproducible scripts and pipelines
  - Cons: Risks introducing opaque, unvalidated code if you're not critically engaged
- Takeaway: Vibe coding is powerful — but only if paired with a solid understanding of workflows, data, and reproducibility principles. *Don’t just vibe — validate.*

The Tidyverse



Tidyverse Packages

Which packages come with `tidyverse`?

```
tidyverse_packages()
```

```
##  [1] "broom"        "conflicted"   "cli"          "dbplyr"       "dplyr"
##  [6] "dtplyr"       "forcats"      "ggplot2"      "googledrive"  "googlesheets4"
## [11] "haven"        "hms"          "httr"         "jsonlite"     "lubridate"
## [16] "magrittr"     "modelr"       "pillar"       "purrr"        "ragg"
## [21] "readr"        "readxl"       "reprex"       "rlang"        "rstudioapi"
## [26] "rvest"        "stringr"      "tibble"       "tidyr"        "xml2"
## [31] "tidyverse"
```

Note that not all these packages are loaded by default.

We now briefly introduce the tidyverse's flagship: `dplyr`.

`dplyr`: The Grammar of Data Manipulation

`dplyr` is the tidyverse toolkit for transforming data frames
Readable, consistent, and designed for pipelines.

Core verbs:

- `filter()` – keep rows that meet a condition
- `select()` – pick columns by name
- `mutate()` – add or transform columns
- `arrange()` – sort rows
- `summarise()` – reduce values to summaries

Additional verbs:

- `group_by()` – perform operations by group
- `sample_n()` – randomly sample rows

Complete `dplyr` reference

The `tidymodels` package

- Tidymodels extends the tidyverse's "grammar" to modeling tasks.

```
tidymodels::tidymodels_packages()
```

```
##  [1] "broom"       "cli"         "conflicted"  "dials"       "dplyr"
##  [6] "ggplot2"     "hardhat"     "infer"       "modeldata"   "parsnip"
## [11] "purrr"       "recipes"     "rlang"       "rsample"     "rstudioapi"
## [16] "tibble"      "tidyr"       "tune"        "workflows"   "workflowsets"
## [21] "yardstick"   "tidymodels"
```

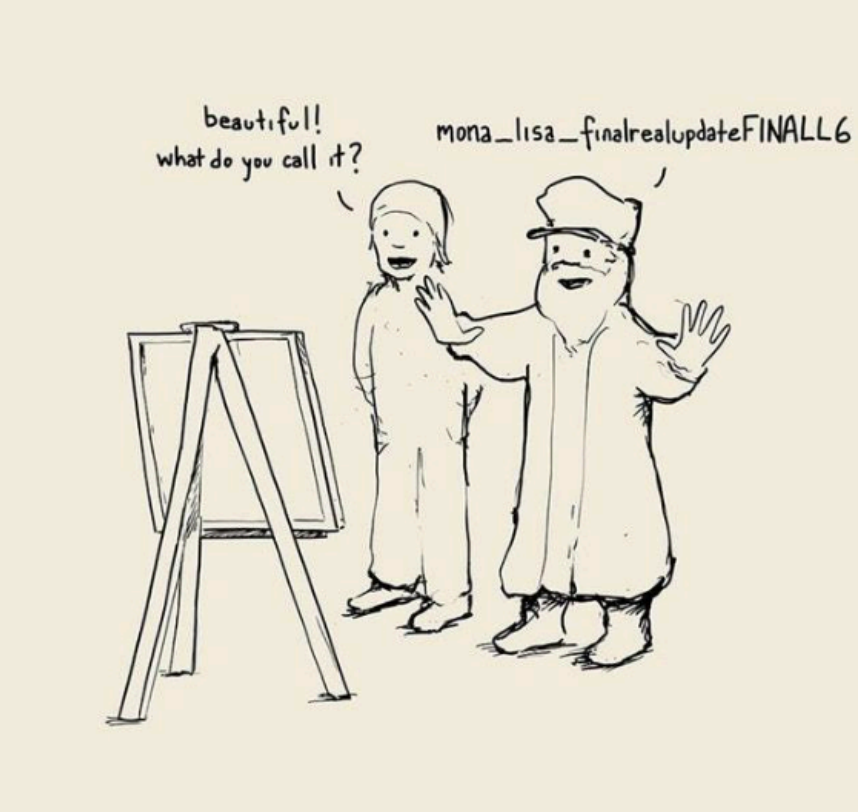
For more information, visit the tidymodels GitHub repo.

Resources

1. R for Data Science (r4ds, 2e) by Garrett Grolemund and Hadley Wickham.
2. Data Visualiztion: A practical introduction by Kieran Healy.

Version Control

Version Control



The "*_FINAL_v2_REALLY_FINAL_THIS_TIME.xlsx*" problem:

- What exactly changed?
- Where in the file?
- When was it edited?
- Who made the change?
- Why was it changed?

Without version control, your data and code become a mystery.

Let's fix that.

Git

- Git is a *distributed version control system*.
- Sounds fancy? Think "track changes", but for entire code projects.
- Git tracks every change, by every contributor, across time.
- It's the de facto standard for managing code, collaborating, and ensuring reproducibility.
- Git isn't just for code — it's used to track any plain text files. It's a foundational tool for reproducible research

GitHub

- GitHub is a web-based platform for hosting Git repositories.
- It’s like Dropbox for Git projects — but with features for collaboration, issue tracking, and code review.
- GitHub builds on Git by making it easier to work with others and manage projects online.
- It's a key hub for open-source development, including many R packages and research tools.

Git vs GitHub

> Git is the engine. GitHub is the garage.

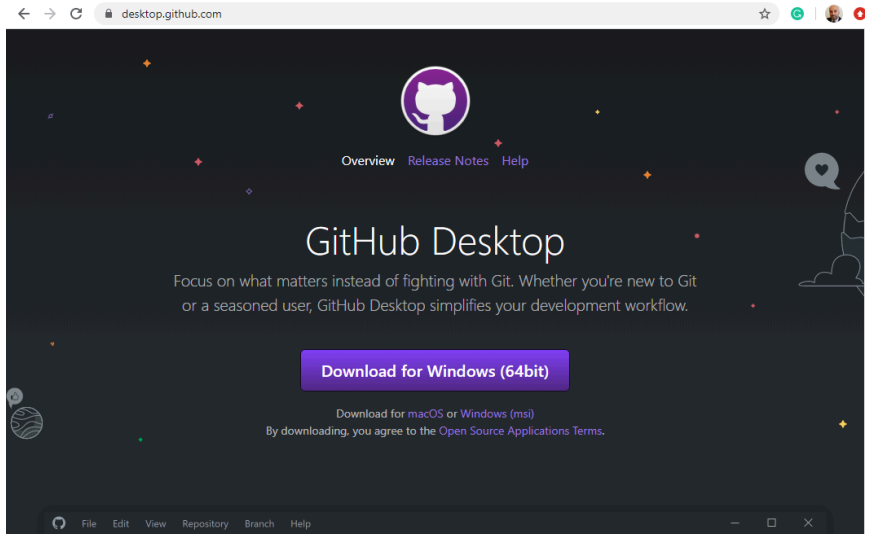
Git

- Version control system
- Tracks changes locally
- Runs in your terminal
- Works without internet
- Created by Linus Torvalds (2005)

GitHub

- Hosting platform for Git repositories
- Adds collaboration tools (issues, pull requests)
- Web-based interface
- Requires Git to function
- Launched by GitHub, Inc. (2008)

GitHub Desktop



- GitHub Desktop is a graphical interface for working with Git and GitHub.
- It lets you commit changes, manage branches, and sync with remote repositories — all without using the command line.
- It also supports pull requests, merge conflict resolution, and collaborative workflows.

Resources

1. Happy Git and GitHub for the useR by Jenny Bryan.
2. Version Control with Git(Hub) by Grant McDerrmot.
3. Pro Git (hardcore).

Workflow

Suggested Workflow: Starting a New R Project with Git

- In RStudio:

  1. Open RStudio.
  2. Go to *File → New Project → New Directory → New Project*.
  3. Enter a name for your project under Directory name.
  4. Check the box: Create a git repository.
  5. Check the box: Use renv with this project.

- In GitHub Desktop:

  1. Open GitHub Desktop.
  2. Go to *File → Add local repository*.
  3. Select your newly created RStudio project folder as the Local path.
  4. Click Publish repository to upload it to GitHub — choose public or private.

  > Optional: add a `README.md` and a `.gitignore` file before publishing.

Suggested Git Workflow (Optional)

- Pull → Stage → Commit → Push

    1. Open GitHub Desktop.
    2. Set the *Current repository* to your project.
    3. Click Fetch origin, then Pull to sync with the latest changes from GitHub.
    4. Open your project in RStudio (or your editor of choice).
    5. Make and save changes to one or more files.
    6. Review the changed files in GitHub Desktop.
    7. Select (stage) the files you want to include in the commit.
    8. Write a brief commit summary (and optional description).
    9. Click Commit to main.
    10. Click Push origin (or press `Ctrl + P`) to upload your changes to GitHub.

    > Tip: Always pull before you push — avoid merge conflicts.

Clone and Sync a Remote GitHub Repository (Optional)

- Cloning a Repository:

  1. Launch GitHub Desktop.
  2. Navigate to the remote repository.
  3. Select "Clone or download".
  4. Define the local path for your cloned repo (e.g., "C:/Documents/CLONED_REPO").

- Synchronizing a Repository:

  1. Launch GitHub Desktop.
  2. Switch "Current repository" to the cloned repo.
  3. Press the "Fetch origin" button.
  4. Pull any updates made on the remote repo.

Your Homework

- Getting Started with R and Git:

    1. Open RStudio.
    2. Create an R project.
    3. Initiate Git.[1]
    4. Create a new RMarkdown file.
    5. Commit your changes via GitHub Desktop.

[1] RStudio automatically generates a `.gitignore` file that tells Git which files to ignore. Click here for more details on configuring what to ignore.

```
slides |> end()
```

Source code