Learning Lab: Software Quality at RFF

Presented by Penny Liao & McKenna Peplinski

Data Governance Working Group

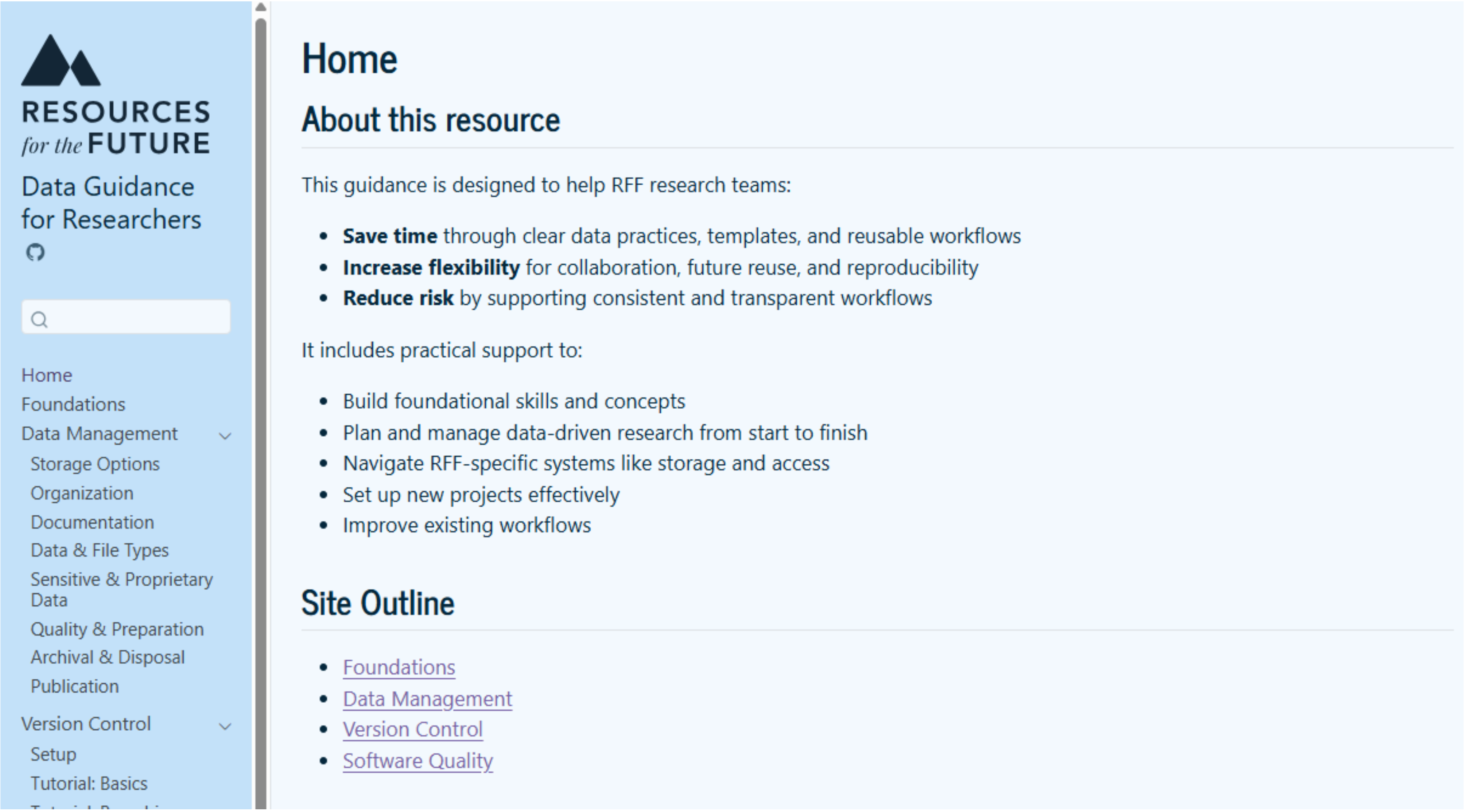
February 24, 2026

Data Governance Working Group

- Aris Awang
- Penny Liao
- McKenna Peplinski
- Alexandra Thompson
- Matthew Wibbenmeyer

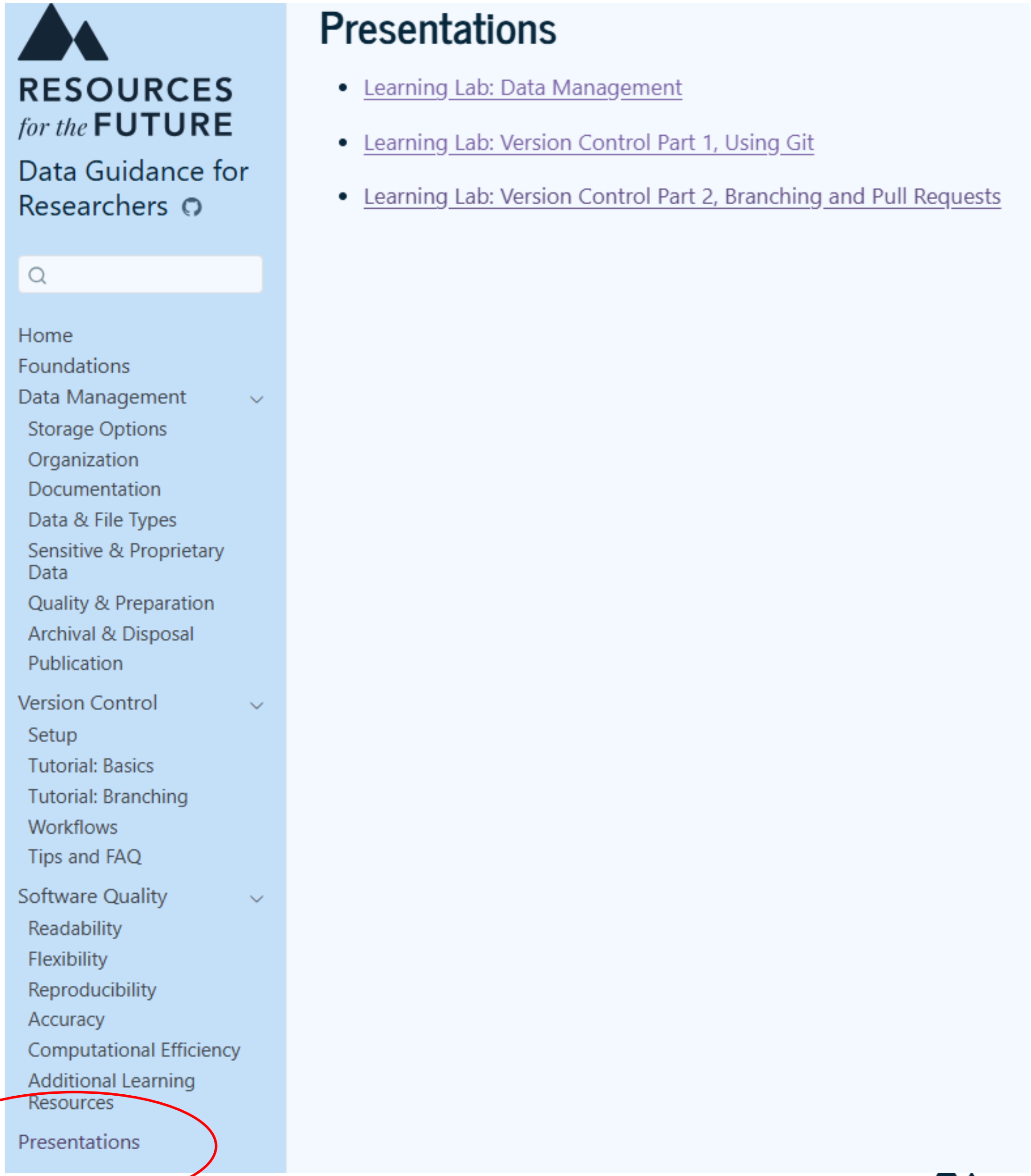
Guidance Website

- [Home – Data Guidance for Researchers](https://rff-data-projects.github.io/rff-data-gov-guidance/index.html) (https://rff-data-projects.github.io/rff-data-gov-guidance/index.html)



Previous Learning Labs

- Learning Lab: Data Management
- Learning Lab: Version Control Part 1, Using Git
- Learning Lab: Version Control Part 2, Branching and Pull Requests



Overview: Software Quality

Due to the computational nature of our research, software development is a primary component of everyday work at RFF. Best practices in scientific coding are designed to:

- ensure **accuracy** and minimize errors,
- enhance code **readability** and **flexibility**,
- improve **reproducibility**, and
- improve **efficiency** in computation.

Open-ended discussion: examples, challenges, tools, solutions

Readability

Readable code reduces the time collaborators and future developers spend deciphering complex code and ensures continuity even if the original author is unavailable. To accomplish this, we recommend:

1. **Modularizing code**
2. **Using consistent code style**
3. **Providing clear in-code documentation**

"Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread."
— Tidyverse Style Guide

Readability: *Modularizing Code*

Modularizing code: breaking script into small, self-contained, and logically organized blocks, each with a clear and focused purpose.

→ Each code block should be accompanied by **descriptive in-line comments** explaining its role and expected behavior.

→ Modularization also helps with **debugging**: when something goes wrong, you can **isolate and fix the issue**

Readability: *Using Consistent Style Code*

Code style: conventions that govern how code is written and formatted

Use **consistent, distinctive, and meaningful names** for variables, functions, datasets, and files:

- **Variable or object** names should be descriptive of their content (e.g., discount_rate instead of val)
- **Function** names should describe their action or output (e.g., calculate_average_price()).
- **Datasets and files** should follow a naming pattern with relevant identifiers (e.g., county_population_2022.csv instead of data_final.csv)

Readability: *Using Consistent Style Code*

"There are only two hard things in Computer Science: cache invalidation and naming things."
— Phil Karlton

Established style guides:

- R: *Tidyverse Style Guide* by Hadley Wickham
- Python: *Google Python Style Guide*
- Stata: *Suggestions on Stata programming style* by Nicholas Fox
- Julia: Blue Style Guide

Object names

```
# Good
day_one
day_1

# Bad
DayOne
dayone
```

Calling functions

```
# Good
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
                              )
```

Readability: *In-Code Documentation*

Three types of in-code documentation are recommended:

1. **Script headers**
   Outline key metadata such as the script's objective, author, and start date.

2. **Block-level comments**
   Use comments to describe the intent and logic of each major code block.

3. **Inline comments**
   Comments on individual lines of code are useful when functionality is not obvious or there are potential limitations.

Readability: *Example*

```r
d<-read.csv("data.csv")
d$co2[d$co2<0]<-0
d$hr<-d$heatRate/1000
x=d[d$state=="CA",]
m=mean(x$co2,na.rm=T)
s=sd(x$co2,na.rm=T)
z=(x$co2-m)/s
x$o=abs(z)>2
write.csv(x,"out.csv")
print(mean(x$o))
```

```r
# Clean emissions data and flag CO2 outliers

read_input <- function(path) {
  read.csv(path, stringsAsFactors = FALSE)   # load raw data
}

clean_emissions <- function(df) {
  df$co2[df$co2 < 0] <- 0                   # fix invalid values
  df$heat_rate_mmbtu <- df$heatRate / 1000   # unit conversion
  df
}

flag_outliers <- function(df, state, z = 2) {
  x <- df[df$state == state, ]               # filter to state
  m <- mean(x$co2, na.rm = TRUE)             # mean CO2
  s <- sd(x$co2, na.rm = TRUE)               # std dev CO2
  x$outlier <- abs((x$co2 - m) / s) > z      # z-score rule
  x
}

all_cleaning_steps <- function() {
  df  <- read_input("data.csv")              # read input
  df  <- clean_emissions(df)                 # clean data
  res <- flag_outliers(df, "CA")             # analyze
  write.csv(res, "out.csv", row.names = FALSE) # save output
}

all_cleaning_steps()                         # run pipeline
```

Flexibility

Flexibility: how easily code can be **adapted and modified** to accommodate changes in external factors, data, or methodology. We recommend three practices to enhance program flexibility:

1. **Use functions**

2. **Use parameters instead of hard coding**

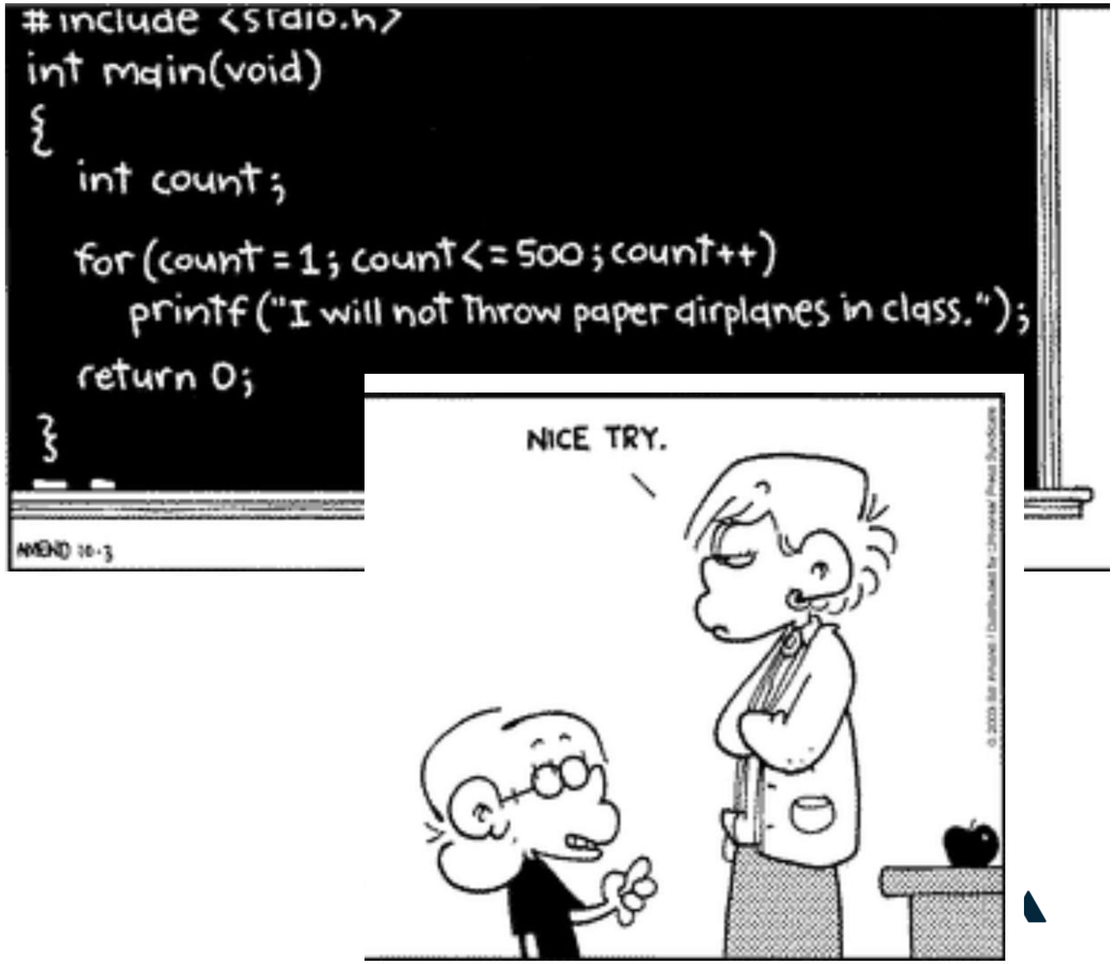
3. **Use relative file paths**

Flexibility: *Functions*

Most operations can be written as functions, which can be applied to different datasets and variables.

Tips to use functions effectively:

1. **Define a clear purpose**
2. **Document thoroughly**
3. **Make function configurable**
4. **Generalize operations using loops and lists**



Flexibility: *Functions*

```r
library(tidyverse)

# ----------------------------
# Function
# ----------------------------

# Function to calculate the mean of a numeric vector
# Args:
#   x: A numeric vector
#   na_as_zero: Logical; treat NA as zero? Default FALSE.
#   na.rm: Logical; remove NA values? Default TRUE.
# Returns:
#   The mean of the numeric vector.

calculate_mean <- function(x, na_as_zero = FALSE, na.rm = TRUE) {

  # Replace NAs with 0 if specified
  if (na_as_zero) {
    x <- replace_na(x, 0)
  }

  # Calculate the mean
  mean_value <- mean(x, na.rm = na.rm)
  return(mean_value)
}
```

1

2

3

```r
# ----------------------------
# Example data
# ----------------------------
monthly_sales <- list(
  January   = c(100, 200, 150, NA, 300),
  February  = c(250, 300, NA, 400),
  March     = c(200, 180, 220, 210)
)

# ----------------------------
# Result
# ----------------------------
# Apply with na_as_zero = TRUE
mean_sales_with_zero <- lapply(monthly_sales, calculate_mean, na_as_zero = TRUE)

# Apply with na_as_zero = FALSE
mean_sales_without_zero <- lapply(monthly_sales, calculate_mean, na_as_zero = FALSE)

# Print the results
print(mean_sales_with_zero)
print(mean_sales_without_zero)
```

4

Flexibility: *Parameters*

*Use **variables, configuration files, or external resources** to store data so that it can be adjusted without changing the code structure.*

```
# Parameters
min_hp       <- 100   # Minimum horsepower required
selected_cyl <- 6     # Number of cylinders to filter on

# Filter the data using the parameters
filtered_data <- subset(mtcars, hp >= min_hp & cyl == selected_cyl)

# Compute the average MPG
avg_mpg <- mean(filtered_data$mpg)

# Report results using the parameters
cat(
  "Average MPG for cars with at least", min_hp, "horsepower",
  "and", selected_cyl, "cylinders:", avg_mpg, "\n"
)
```

Flexibility: *Relative file paths*

Absolute file paths *specify the complete location of a file or folder*: L:/my_project/data/input_data.csv
Relative file paths *refer to files or directories in relation to the current working directory*.

```r
# Set the working directory to the script's location (optional, recommended)
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))  # Temporarily set WD to where the script is stored

# Move up two levels to the project root ('my_project/')
setwd("../../")

# Define relative paths
input_path  <- "data/input_data.csv"
output_path <- "data/output_data.csv"

# Read data using a relative path
data <- read.csv(input_path)

# Write data using a relative path
write.csv(data, output_path, row.names = FALSE)
```

```
my_project/
├── data/
│   ├── input_data.csv
│   └── output_data.csv
└── scripts/
└── wilson/
└── analysis.R
```

Reproducibility: *Principles of Project Script Structure*

To ensure reproducibility, the code base should be organized as a logical sequence of scripts that take the raw data and generates the complete set of outputs:

- **1 script = 1 major task**
    - e.g. pre-processing a large dataset, creating geographical overlays, merging, running a specific analysis
- **Separate out time-consuming steps and save the output**
- **Isolate key intermediate outputs**
    - If a code block produces an output that will be used in multiple subsequent steps, place it in a dedicated script
- **Extract reusable code blocks** (e.g. functions used across multiple scripts)
    - *In R:* `source("script_name.R")`
    - *In Python:* `import module_name`
    - *In Stata:* `do filename.do`

Reproducibility: *Number Scripts and Script Folders*

Encode execution order in script and folder names:

- Number the scripts:
  - 00_,
  - 01_,
  - 02_
- Parallel steps:
  - 01a_,
  - 01b_
- Same for folders:
  - 00_setup,
  - 01_processing,
  - 02_analysis

Example Script Folder

```
scripts/
|--00_tools/                        # Shared helpers, not part of pipeline
|    |--filters.R                   # Reusable filter functions
|    |--data_io.R                   # Reusable input/output wrapper functions
|
|--01_processing/                   # Data prep and integration
|    |--01_clean.R                  # Clean raw data
|    |--02_aggregate.R              # Aggregate to analysis unit
|    |--03_merge_external.R         # Merge with supplementary datasets
|
|--02_analysis/                     # Analysis and outputs
|    |--01a_summary_stats.R         # Summary statistics
|    |--01b_map.R                   # Descriptive spatial visualizations
|    |--01c_time_series_figure.R    # Descriptive time series plots
|    |--02_main_regressions.R       # Main statistical analysis
|    |--03_robustness_checks.R      # Alternative specifications
```

Reproducibility: *Create a "run_all" Script*

A script that executes all other scripts in the correct order:

- Documents the **full workflow:** reproduces the complete set of outputs starting from raw data
- Needs to be kept **up to date** on any script structure change
- **Global parameters** should be placed in a dedicated configuration script at the top

Example `run_all.R` Script

```r
# This script runs all the code in my project, from scratch

# Prepare analysis data
source("scripts/01_processing/01_clean.R")
source("scripts/01_processing/02_aggregate.R")
source("scripts/01_processing/03_merge_external.R")

# Descriptive stats and figures
source("scripts/02_analysis/01a_summary_stats.R")
source("scripts/02_analysis/01b_map.R")
source("scripts/02_analysis/01c_time_series_figure.R")

# Analysis
source("scripts/02_analysis/02_main_regressions.R")
source("scripts/02_analysis/03_robustness_checks.R")
```

Reproducibility: *Package Management*

Code may not run the same if software and package versions are different

Project-specific **virtual environments** keep the computational environment/package versions consistent over time and across machines

Potential options:

renv (R only): record and restore R package versions

conda (Python, R, etc.): cross-platform, multi-language manager

Docker (general): package the entire computational environment

Note: We have not fully tested these tools. Please contact us if you have used them within RFF's computational environment!

Accuracy

Accuracy refers to the correctness and precision of written code in executing its intended functions without errors or unintended consequences.

Three recommended practices:

- Pseudocoding
- Testing and debugging
- Code review

Accuracy: *Pseudocoding*

- **Pseudocode** is a step-by-step, high-level description of an algorithm written in plain English
- *Have you or your team used pseudocoding before? Why did you use it?*
  - Clarify the logic of a complex algorithm
  - Facilitate consensus in teams
  - Great for learning

Accuracy: *Psuedocoding Example*

```
# Goal: Download migration data for years 2011-2021 from IRS website

Load libraries
Set working directory

# Create function to download to IRS data
download_irs_data <- function(IRS data year, file destination) {
  url <- Web address to IRS data for IRS data year
  download file at url and save to file destination
}

# Execute function for years 2011-2021
dst <- selected destination
create dst folder if it doesn't already exist

for year in 2011-2021 {
  download_irs_data(IRS data year = year, file destination = dst)
}
```

```r
# Load necessary libraries
library(dplyr)
library(tidyr)
library(readr)

# Set the new working directory to path where script is saved
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))

# Function to download IRS inflow migration data
download_irs_data <- function(year1, dst) {
  year2 <- sprintf("%02d", (year1 + 1) %% 100) year1 <- sprintf("%02d", year1 %% 100)
  link <- sprintf("https://www.irs.gov/pub/irs-soi/countyinflow%s%s.csv", year1, year2)
  download.file(link, sprintf("%s/countyinflow%s%s.csv", dst, year1, year2))
}

# Download inflow data for years 2011-2021
dst <- "migration-data"
dir.create(dst, showWarnings = FALSE, recursive = TRUE)
lapply(seq(11, 21), download_irs_data, dst = dst)
```

Accuracy: *Testing and Debugging*

Test every step of the analysis, even when it runs without error – never assume!

- Build and test code in small, manageable pieces (e.g. functions)
- Pay attention to error and warning messages
- Use logs and diagnostic print statements
    - In long loops, logging can help track progress, diagnose performance bottlenecks, and pinpoint where errors occur
- Inspect intermediate and final outputs
    - Check summaries, counts, and simple plots at key steps for unexpected values, extreme outliers, incorrect data types, and missing data
    - Check the number of rows and columns, especially after merges, joins, reshaping, or filtering

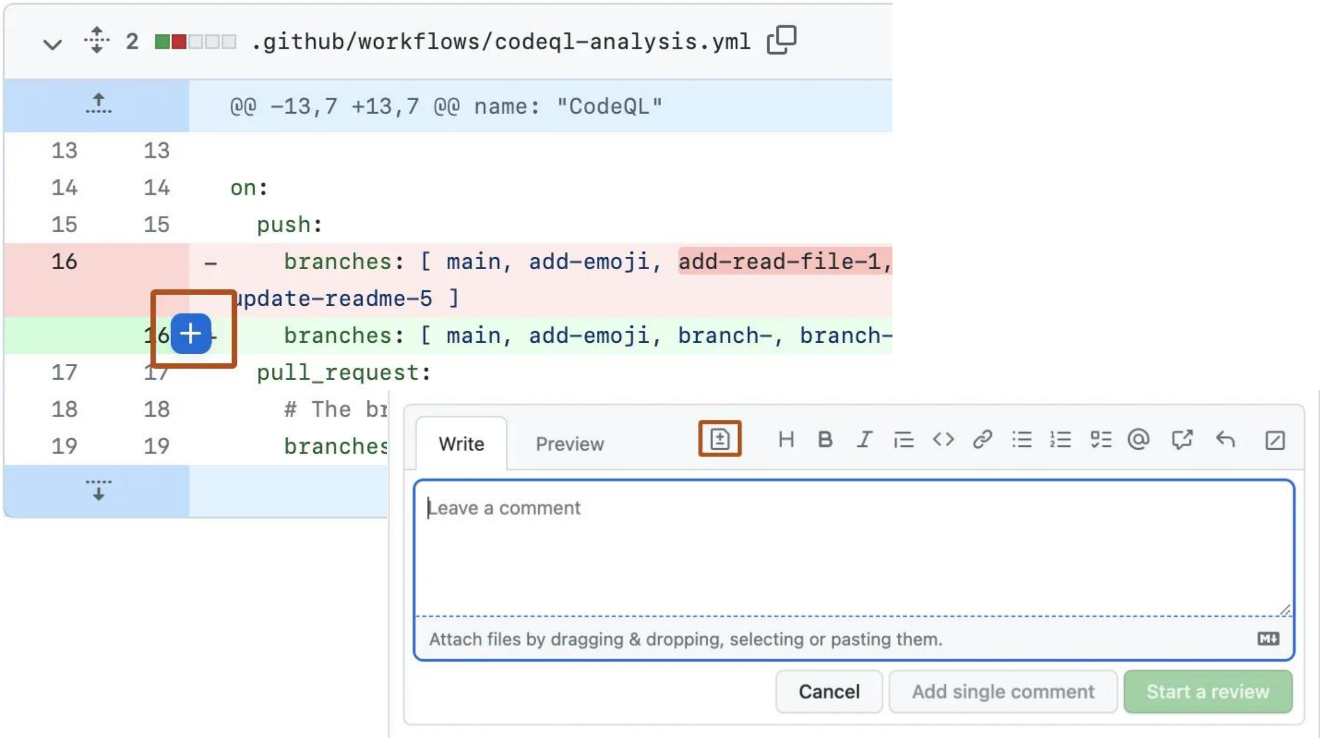
Accuracy: *Code Review*

Code review is the practice of having a colleague examine major coding components

- We recommend setting up a code review system at the start of a project

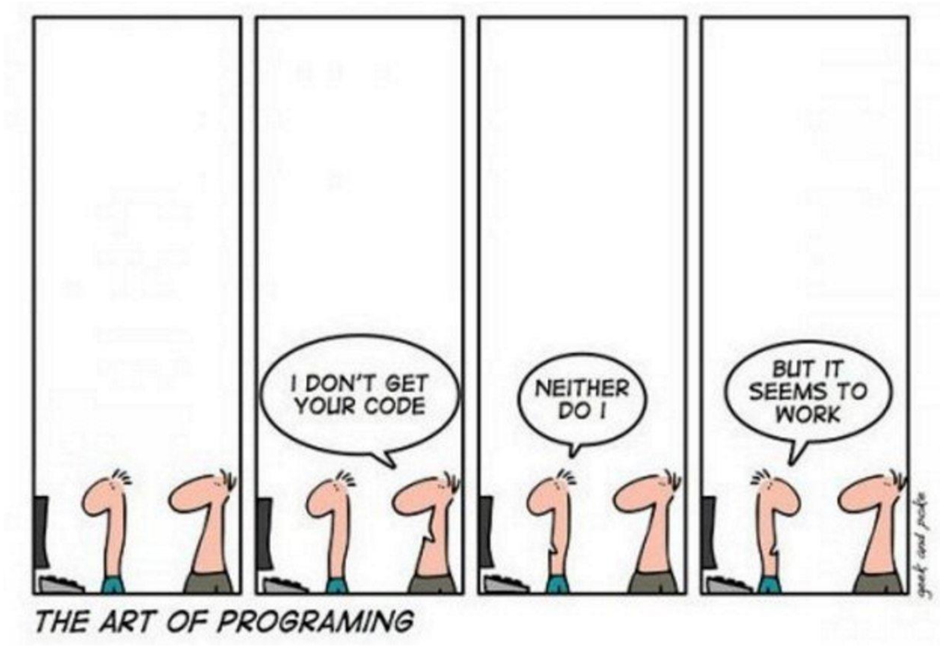
What kind of code review practices have you or your team used?

1. Github pull request
2. Written feedback
3. In-person meetings
4. Pair programming
5. Replication
6. Check output



Accuracy: *What to check for in code review*

- **Correctness:** Does the code do what it's supposed to? Are edge cases handled?
- **Clarity:** Are variable and function names clear? Is the logic easy to follow?
- **Reproducibility:** Can someone else run the code with the provided inputs and get the same result?
- **Efficiency:** Is the code written in a reasonably efficient way (avoiding unnecessary loops or redundancy)?
- **Consistency:** Does the code follow agreed formatting, naming, and documentation conventions?
- **Documentation:** Are inputs, outputs, and assumptions documented? Are comments clear and useful?



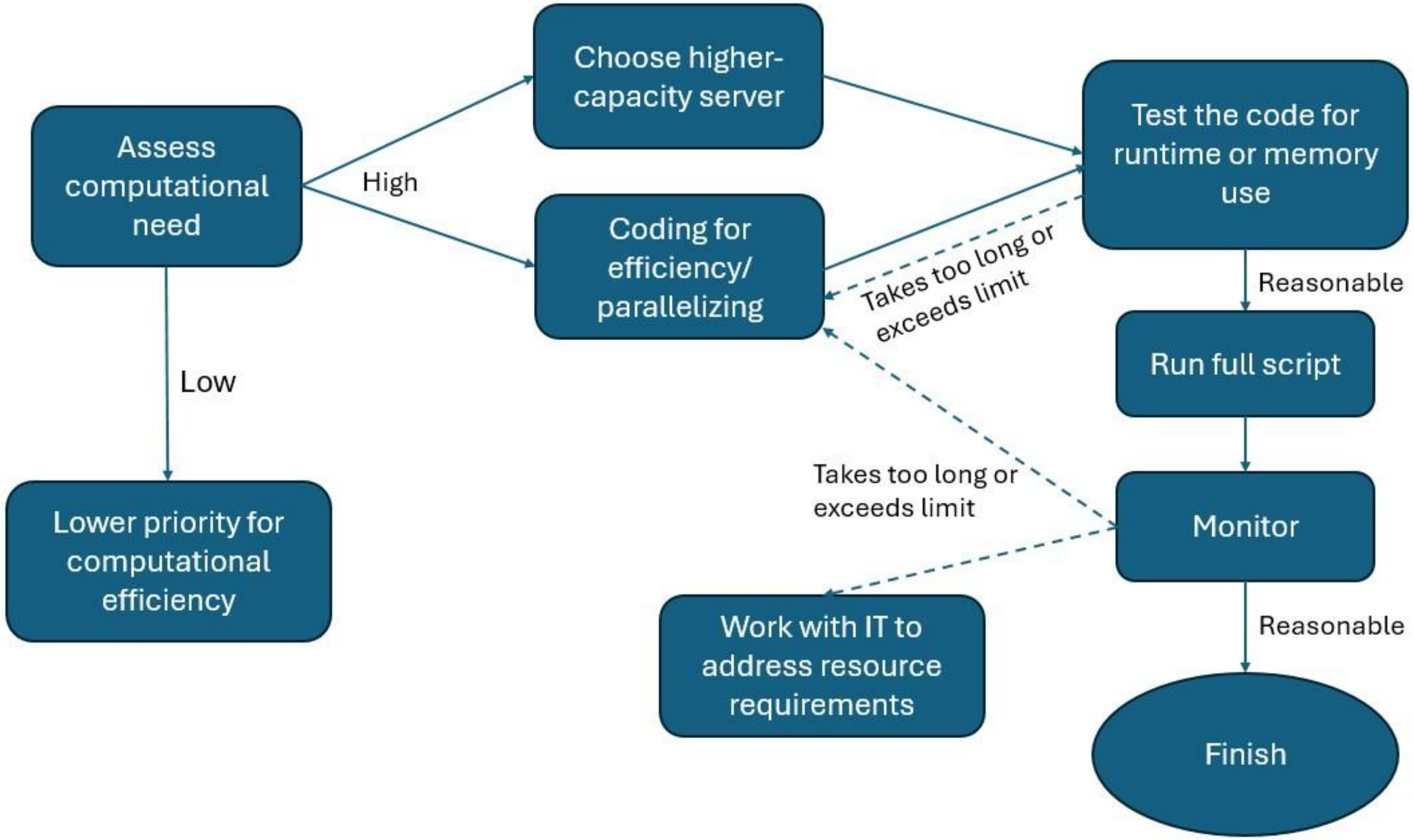
Computational Efficiency

Computational efficiency refers to how effectively code uses available resources:

- **Time** – how long a task takes to run
- **RAM (Random Access Memory)** – how much data can be processed at once
    - Fast, temporary workspace where data and objects are stored while code is running
    - RAM ≠ Disk Storage
    - Most statistical software load data into memory before performing computation
    - When memory is exceeded, it can cause sharp performance slowdown or crashes

Computational Efficiency

The need to manage computational efficiency differs by project and should be approached on a case-by-case basis:



Computational Efficiency: *Practices in R*

1. Use vectorization over loops
2. Avoid growing objects inside loops
3. Use efficient data structures
4. Profile and benchmark code
5. Use efficient input/output (I/O)
6. Manage garbage collection
7. Use parallel computation when appropriate

Small Group Discussion

- What are the biggest challenges you have faced in code development at RFF?
- What are the solutions that work well?
- What can RFF or the Data Governance Working Group do to better support your work?

Please fill out the survey!

Thank you.

- Find out more about RFF online: **www.rff.org**
- Follow us on **LinkedIn**
- Subscribe to receive updates: **rff.org/subscribe**

Computational Efficiency: *Practices in R*

Practice 1: Use vectorization over loops

- Operating on entire data structures (e.g. vectors, matrices) at once rather than iterating element by element in loops
- Many base R functions are inherently vectorized, including arithmetic operations (e.g. +, *), math functions (e.g., log(), sqrt()), and summary functions (e.g., mean(), rowSums())
- Use vectorized conditions such as ifelse() of instead of looping with if / else
- Use apply family functions (e.g., apply(), lapply(), or sapply()) rather than looping

Computational Efficiency: *Practices in R*

Example: lapply() vs. a loop to standardize multiple numeric columns

```
df <- data.frame(
  a = rnorm(1e5),
  b = rnorm(1e5),
  c = rnorm(1e5),
  group = sample(letters[1:3], 1e5, replace = TRUE)
)

num_cols <- c("a", "b", "c")

# Using a loop (inefficient)
for (col in num_cols) {
  df[[col]] <- (df[[col]] - mean(df[[col]])) / sd(df[[col]])
}

# Using lapply() (efficient)
df[num_cols] <- lapply(df[num_cols], function(x) {
  (x - mean(x)) / sd(x)
})
```

Computational Efficiency: *Practices in R*

Practice 2: Avoid growing objects inside loops

- Repeatedly expanding an object inside a loop (e.g., using rbind() or append() on each iteration) is slow and memory-intensive because R needs to allocate new memory and copy the existing object each time it grows

- Instead, preallocate objects to their final size:

```r
# Grow the vector each iteration (inefficient)
vec <- numeric(0)
for (i in 1:1000) {
  vec <- c(vec, i^2)
}

# Preallocate and fill (efficient)
vec <- numeric(1000)
for (i in 1:1000) {
  vec[i] <- i^2
}
```

Computational Efficiency: *Practices in R*

Practice 2: Avoid growing objects inside loops

- When combining many results, store them in a list and use do.call(rbind, …) or rbindlist() rather than repeatedly appending rows:

```
result_list <- vector("list", 1000)

for (i in 1:1000) {
  result_list[[i]] <- data.frame(id = i, value = i^2)
}

result <- do.call(rbind, result_list)
```

Computational Efficiency: *Practices in R*

Practice 3: Use efficient data structures

- A **data structure** is a format for organizing, retrieving, and storing data (e.g. vectors, matrices, data.frames, data.tables)
- data.table is more efficient than data.frame for large tabular data – it is optimized for fast grouping, joins, and in-place updates
- Use factors for categorical variables instead of character strings

Computational Efficiency: *Practices in R*

Practice 4: Profile and benchmark code

- system.time()

```
system.time({
  out <- df$x^2 + log(df$y)
})
```

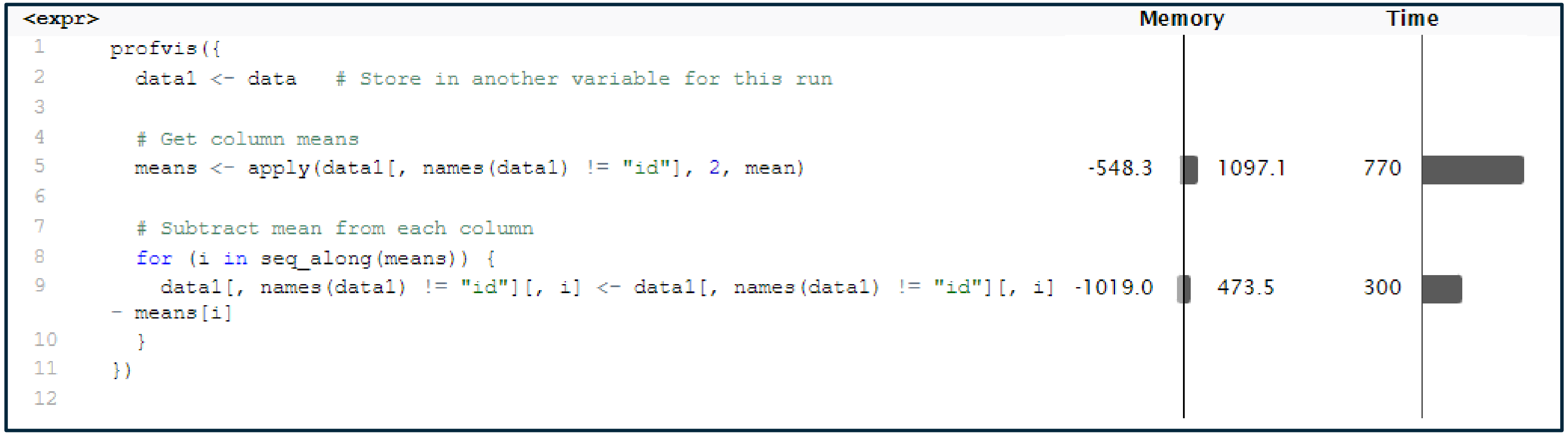
- “tictoc” library

```
library(tictoc)

tic("Step 1: transform")
df$x2 <- df$x^2
toc()

tic("Step 2: summarize")
m <- mean(df$x2)
toc()
```

- “profvis” library provides a visual breakdown of time spent and identifies memory-intensive operations



Computational Efficiency: *Practices in R*

Practice 5: Use efficient input/output (I/O)

- Use high-performance read and write functions for large datasets
  - data.table::fread() / fwrite() or readr::read_csv() are more efficient than base R functions like read.csv()
- Save intermediate data to avoid repeated reads of large raw files
- Formats such as .fst, .rds, or .parquet are typically faster to read and write and more memory-efficient than plain text formats (e.g., .csv, .txt)
- Read only the data you need
  - data.table::fread() / fwrite() and readr::read_csv() can scan the data structure and selectively load columns or rows without reading the entire dataset

Computational Efficiency: *Practices in R*

Practice 6: Manage garbage collection

- R automatically frees memory from unused objects
- Garbage collection requires R to pause and scan memory and can slow performance in memory-intensive workflows

Best Practices

- Reduce unnecessary object creation
- Avoid large intermediate objects and repeated copying
- Use gc() sparingly:
  - After removing very large objects
  - Before a memory-intensive task

Computational Efficiency: *Practices in R*

Practice 7: Parallel computation

- Reduces runtime by distributing independent tasks across multiple CPU cores
- Best when applying the **same** operation to many files, datasets, or units
- Recommended packages: future, furrr, base R's parallel

Example: parallelizing independent tasks with future and furrr

```r
library(future)
library(furrr)

# Set up a parallel plan (adjust workers as appropriate)
plan(multisession, workers = 4)

units <- unique(df$county_id)

# Apply the same function independently to each unit
results <- future_map(units, function(u) {
  sub <- df[df$county_id == u, ]
  mean(sub$value, na.rm = TRUE)
})
```

Computational Efficiency: *Practices in R*

Practice 7: Parallel computation

Pay particular attention to:

- Memory usage. Each parallel worker may require its own copy of data, which can substantially increase memory consumption. Parallelization may be counterproductive when working with very large objects.

- Shared computing environments. When working on shared servers, ensure that parallel execution does not monopolize system resources or interfere with other users. Limit the number of cores used when appropriate.