UCD-SeRG Lab Manual

Last updated: 2026-01-28

Contents

1 Welcome to UCD-SeRG!

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

1.1 About the lab

Welcome to the Seroepidemiology Research Group (SeRG) at the University of California, Davis, led by Drs. Kristen Aiemjoy and Ezra Morrison. Accurate methods to measure infectious disease burden are essential for guiding public health decisions, yet many infectious diseases remain under-recognized due to limited diagnostics and costly, resource-intensive surveillance systems. Our work addresses this gap by developing seroepidemiologic methods to characterize infection burden in populations. Currently, we focus on enteric fever (Salmonella Typhi and Paratyphi), Scrub Typhus (Orientia tsutsugamushi), Melioidosis (Burkholderia pseudomallei), Shigella (Shigella spp.), and Cholera (Vibrio cholerae). We are supported by the US National Institutes of Health, the Bill and Melinda Gates Foundation, and the Department of Defense, and collaborate with partners around the world. To learn more about the lab, visit ucdserg.ucdavis.edu[2].

1.2 About this lab manual

This lab manual covers our communication strategy, code of conduct, and best practices for reproducibility of computational workflows. It is a living document that is updated regularly.

This manual is a fork of the Benjamin-Chung Lab Manual (Benjamin-Chung et al. 2024), adapted for UCD-SeRG. We are grateful to Dr. Jade Benjamin-Chung and her team for developing and openly sharing their excellent lab manual. You can view the original manual at jadebc.github.io/lab-manual[3]. Original contributors include Jade Benjamin-Chung, Kunal Mishra, Stephanie Djajadi, Nolan Pokpongkiat, Anna Nguyen, Iris Tong, and Gabby Barratt Heitmann.

Feel free to draw from this manual (and please cite it if you do!).

[1] https://jadebc.github.io/lab-manual/index.html
[2] https://ucdserg.ucdavis.edu
[3] https://jadebc.github.io/lab-manual/index.html

2 Culture and conduct

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

2.1 Lab culture

We are committed to a lab culture that is collaborative, supportive, inclusive, open, and free from discrimination and harassment.

We encourage students / staff of all experience levels to respectfully share their honest opinions and ideas on any topic. Our group has thrived upon such respectful honest input from team members over the years, and this document is a product of years of student and staff input (and even debate) that has gradually improved our productivity and overall quality of our work.

2.2 Diversity, equity, and inclusion

UCD-SeRG recognizes the importance of and is committed to cultivating a culture of diversity, equity, and inclusion. This means being a safe, supportive, and anti-racist environment in which students from diverse backgrounds are equally and inclusively supported in their education and training. Diversity takes many forms, and includes, but is not limited to, differences in race, ethnicity, gender, sexuality, socioeconomic status, religion, disability, and political affiliation.

2.3 Protecting human subjects

All lab members must complete CITI Human Subjects Biomedical Group 1[2] training and share their certificate with the lab leadership. Team members will be added to relevant Institutional Review Board protocols prior to their start date to ensure they have permission to work with identifiable datasets.

One of the most relevant aspects of protecting human subjects in our work is maintaining confidentiality. For students supporting our data science efforts, in practice this means:

- Be sure to understand and comply with project-specific policies about where data can be saved, particularly if the data include personal identifiers.
- Do not share data with anyone without permission, including to other members of the group, who might not be on the same IRB protocol as you (check with lab leadership first).

[1] https://jadebc.github.io/lab-manual/culture-and-conduct.html
[2] https://research.ucdavis.edu/policiescompliance/irb-admin/education/

Remember, data that looks like it does not contain identifiers to you might still be classified as data that requires special protection by our IRB or under HIPAA, so always proceed with caution and ask for help if you have any concerns about how to maintain study participant confidentiality.

2.4 Authorship

We adhere to the ICMJE Definition of authorship[3] (International Committee of Medical Journal Editors, n.d.) and are happy for team members who meet the definition of authorship to be included as co-authors on scientific manuscripts. To qualify for authorship, individuals must meet all four criteria:

1. Substantial contributions to conception/design, or acquisition/analysis/interpretation of data
2. Drafting the work or revising it critically for important intellectual content
3. Final approval of the version to be published
4. Agreement to be accountable for all aspects of the work

Authorship practices:

- **First authorship**: Typically goes to the person who led the work
- **Corresponding author**: Usually the PI, unless otherwise agreed
- **Co-authorship**: Determined by substantial intellectual contributions
- **Author order**: Should be discussed and agreed upon by all authors
- **Acknowledgments**: For contributions that don't meet authorship criteria

Authorship should be discussed early in a project and revisited as the work evolves to ensure transparency and fairness. We encourage using the CRediT Taxonomy[4] to document specific author contributions.

[3] http://www.icmje.org/recommendations/browse/roles-and-responsibilities/defining-the-role-of-authors-and-contributors.html
[4] https://credit.niso.org/

3 Communication and coordination

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

One benefit of the academic environment is its schedule flexibility and autonomy. This means that lab members may choose to work in the early morning, afternoon, evening, or weekends. That said, we do not expect lab members to respond outside of normal business hours (unless there are special circumstances).

3.1 Microsoft Teams

- Use Microsoft Teams for scheduling, coding related questions, quick check ins, etc. If your Teams message exceeds 200 words, it might be time to use email.
- Use channels instead of direct messages unless you need to discuss something private.
- Please make an effort to respond to messages that mention you (e.g., `@username`) as quickly as possible and always within 24 hours.
- If you are unusually busy (e.g., taking MCAT/GRE, taking many exams) or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual and also set your status in Teams (e.g., it could say "On vacation") so we know not to expect to see you online.
- Please thread messages in Teams as much as possible.
- Don't wait for meetings to ask questions. As soon as a question comes up, write it out in Teams. This benefits both you (by clarifying your thinking, as discussed in Chapter 22) and the team (by getting the conversation started earlier).

3.2 Email

- Use email for longer messages (>200 words) or messages that merit preservation.
- Generally, strive to respond within 24 hours hours. As noted above, if you are unusually busy or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual.

3.3 Task Management

We use a combination of tools to track and manage project tasks:

- **GitHub Issues and Projects**: For code-related tasks, feature requests, and bug tracking. Lab leadership will assign issues and organize them in GitHub Projects. Issues are prioritized within projects, and you can track your assigned tasks there.

[1] https://jadebc.github.io/lab-manual/communication-and-coordination.html

- **Microsoft To-Do** and other M365 task tracking tools: For general lab tasks and personal task management. Lab leadership may assign tasks through these tools, which integrate with Microsoft Teams.
- Generally, strive to complete assigned tasks by the date listed.
- Use checklists to break down tasks into smaller chunks. Sometimes leadership will create these for you, but you can also add them yourself.
- Update task status as you make progress so the team can stay coordinated.

3.4 Google Drive

- We mostly use Google Drive to create shared documents with longer descriptions of tasks. These documents may be linked to in GitHub Issues or other task tracking tools. Lab leadership often shares these with the whole team since tasks are overlapping, and even if a task is assigned to one person, others may have valuable insights.

3.5 UC Davis Box and SharePoint

- Human subjects data for research studies are generally stored in UC Davis Box or SharePoint. Please check with lab leadership about whether there are special storage and transfer requirements for the datasets you are working with for each study.
- You can access Box via your UC Davis credentials. For more information, visit UC Davis Box Support[2].
- SharePoint is also used for collaborative document storage and team file sharing. Access SharePoint through your UC Davis Microsoft 365 account.

3.6 Meetings

- Our meetings start on the hour.
- If you are going to be late, please send a message in our Teams channel.
- If you are regularly not able to come on the hour, notify the team and we might choose the modify the agenda order or the start time.

3.7 Code Review

When submitting code to or reviewing code from colleagues, use best practices to provide and receive constructive feedback:

- Tidyverse code review principles[3] (Tidyverse Team 2023): Best practices for reviewing R code, including what to look for and how to provide constructive feedback.

[2] https://servicehub.ucdavis.edu/servicehub?id=ucd_kb_article&sysparm_article=KB0000184
[3] https://code-review.tidyverse.org/

4 Reproducibility

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

Our lab adopts the following practices to maximize the reproducibility of our work.

1. Design studies with appropriate methodology and adherence to best practices in epidemiology and biostatistics
2. Register study protocols
3. Write and register pre-analysis plans
4. Create reproducible workflows
5. Process and analyze data with internal replication and masking
6. Use reporting checklists with manuscripts
7. Publish preprints
8. Publish data (when possible) and replication scripts

4.1 What is the reproducibility crisis?

In the past decade, an increasing number of studies have found that published study findings could not be reproduced. Researchers found that it was not possible to reproduce estimates from published studies: 1) with the same data and same or similar code and 2) with newly collected data using the same (or similar) study design. These "failures" of reproducibility were frequent enough and broad enough in scope, occurring across a range of disciplines (epidemiology, psychology, economics, and others) to be deeply troubling. Program and policy decisions based on erroneous research findings could lead to wasted resources, and at worst, could harm intended beneficiaries. This crisis has motivated new practices in reproducibility, transparency, and openness. Our lab is committed to adopting these best practices, and much of the remainder of the lab manual focuses on how to do so.

Recommended readings on the "reproducibility crisis":

- Nuzzo R. How scientists fool themselves – and how they can stop (Nuzzo 2015)
- Stoddart C. Is there a reproducibility crisis in science? (Stoddart 2019)
- Munafò MR, et al. A manifesto for reproducible science (Munafò et al. 2017)

4.2 Study design

Appropriate study design is beyond the scope of this lab manual and is something trainees develop through their coursework and mentoring.

[1] https://jadebc.github.io/lab-manual/reproducibility.html

4.3 Register study protocols

We register all randomized trials on clinicaltrials.gov, and in some cases register observational studies as well.

4.4 Write and register pre-analysis plans

We write pre-analysis plans for most original research projects that are not exploratory in nature, although in some cases, we write pre-analysis plans for exploratory studies as well. The format and content of pre-analysis plans can vary from project to project. Here is an example of one: https://osf.io/tgbxr/. Generally, these include:

1. Brief background on the study (a condensed version of the introduction section of the paper)
2. Hypotheses / objectives
3. Study design
4. Description of data
5. Definition of outcomes
6. Definition of interventions / exposures
7. Definition of covariates
8. Statistical power calculation
9. Statistical analysis:

- Type of model
- Covariate selection / screening
- Standard error estimation method
- Missing data analysis
- Assessment of effect modification / subgroup analyses
- Sensitivity analyses
- Negative control analyses

4.5 Create reproducible workflows

Reproducible workflows allow a user to reproduce study estimates and ideally figures and tables with a "single click". In practice, this typically means running a single bash script that sources all replication scripts in a repository. These replication scripts complete data processing, data analysis, and figure/table generation. The following chapters provide detailed guidance on this topic:

- Chapter 5: Code repositories
- Chapter 6: Coding practices
- Chapter 7: Coding style
- Chapter 8: Code publication
- Chapter 9: Working with big data
- Chapter 10: Github
- Chapter 11: Unix

For additional learning resources on reproducible research practices, see the UC Davis DataLab workshop on reproducible research[2].

4.6 Process and analyze data with internal replication and masking

See my video on this topic: https://www.youtube.com/watch?v=WoYkY9MkbRE

4.7 Use reporting checklists with manuscripts

Using reporting checklists helps ensure that peer-reviewed articles contain the information needed for readers to assess the validity of your work and/or attempt to reproduce it. A collection of reporting checklists is available from the EQUATOR Network ("EQUATOR Network: Enhancing the QUAlity and Transparency of Health Research," n.d.).

4.8 Publish preprints

A preprint is a scientific manuscript that has not been peer reviewed. Preprint servers create digital object identifiers (DOIs) and can be cited in other articles and in grant applications. Because the peer review process can take many months, publishing preprints prior to or during peer review enables other scientists to immediately learn from and build on your work. Importantly, NIH allows applicants to include preprint citations in their biosketches. In most cases, we publish preprints on medRxiv ("medRxiv: The Preprint Server for Health Sciences," n.d.).

4.9 Publish data (when possible) and replication scripts

Publishing data and replication scripts allows other scientists to reproduce your work and to build upon it. We typically publish data on the Open Science Framework ("Open Science Framework," n.d.), share links to Github[3] repositories, and archive code on Zenodo[4].

[2] https://github.com/ucdavisdatalab/workshop_reproducible_research
[3] github.com
[4] zenodo.org

5 Code repositories

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi[1]

Each study has at least one code repository that typically holds R code, shell scripts with Unix code, and research outputs (results .RDS files, tables, figures). Repositories may also include datasets. This chapter outlines how to organize these files. Adhering to a standard format makes it easier for us to efficiently collaborate across projects.

UCD-SeRG projects use R package structure for most R-based work. This provides benefits for reproducibility, collaboration, and code quality even for analysis-only projects.

5.1 Package Structure

All R projects in our lab should be structured as R packages, even if they are primarily analysis projects and not intended for distribution on CRAN or Bioconductor. This standardized structure provides numerous benefits:

5.1.1 Why Use R Package Structure?

1. **Organized code**: Clear separation of functions (`R/`), documentation (`man/`), tests (`tests/`), data (`data/`), and vignettes/analyses
2. **Dependency management**: `DESCRIPTION` file explicitly declares all package dependencies and version restrictions, which simplifies installing those dependencies.
3. **Automatic documentation**: `roxygen2` generates help files from inline comments
4. **Built-in testing**: `testthat` framework integrates seamlessly with package structure
5. **Code quality**: Tools like `devtools::check()` and `lintr` enforce best practices
6. **Reproducibility**: Package structure makes it easy to share and reproduce analyses
7. **Reusable functions**: Decompose complex analyses into well-documented, testable functions
8. **Version control**: Track changes to code, documentation, and data together

5.1.2 Basic Package Structure

```
myproject/
  DESCRIPTION          # Package metadata and dependencies
  NAMESPACE            # Auto-generated, don't edit manually
  R/                   # All R functions (reusable code)
     analysis_functions.R
     data_prep.R
     plotting.R
```

[1] https://jadebc.github.io/lab-manual/code-repositories.html

```
man/                 # Auto-generated documentation
tests/
    testthat/       # Unit tests
data/                # Processed data objects (.rda files)
data-raw/            # Raw data and data processing scripts
    0-prep-data.sh  # Shell scripts for data preparation
    process_survey_data.R
    clean_lab_results.R
vignettes/           # Long-form documentation
    intro.qmd       # Main vignettes (shipped with package)
    tutorial.qmd
    articles/       # Website-only articles (not shipped)
        advanced-topics.qmd
        case-studies.qmd
inst/                # Additional files to include in package
    extdata/        # External data files and .RDS results
        analysis_results.rds
        processed_data.rds
    output/         # Figure and table outputs
        figures/
            fig1.pdf
            fig2.png
        tables/
            table1.csv
            table2.xlsx
    analyses/       # Analyses using restricted data (see below)
.Rproj               # RStudio project file
```

5.1.3 Where to Place Analysis Files

5.1.3.1 Vignettes vs Articles

Vignettes (`vignettes/*.qmd`): - **Shipped with the package** when installed - Accessible via `vignette()` and `browseVignettes()` in R - Displayed on CRAN - Built at package build time - Use for core package documentation and tutorials - Created with `usethis::use_vignette("name")`

Articles (`vignettes/articles/*.qmd`): - **Website-only** (not shipped with the package) - Only appear on the pkgdown website - Not accessible via `vignette()` in R - Not displayed on CRAN - Use for supplementary content, blog posts, extended tutorials, or frequently updated material - Created with `usethis::use_article("name")` - Automatically added to `.Rbuildignore`

When to use each: - **Vignette**: Essential tutorials users need offline, core package workflows - **Article**: Supplementary material, case studies, advanced topics, blog-style content

5.1.3.2 Public Analyses (vignettes/)

Use `vignettes/` for analysis workbooks that:

- Use publicly available data
- Should be accessible to all package users
- Are core to understanding the package

Use `vignettes/articles/` for:

- Extended case studies
- Blog-style posts
- Supplementary analyses
- Material that updates frequently

All vignettes and articles will be rendered by `pkgdown::build_site()` on your package website.

5.1.3.3 Analyses with Restricted Data (inst/analyses/)

For analyses that rely on **private, sensitive, or restricted data**, place `.qmd` or `.qmd` files in `inst/analyses/`:

```
myproject/
  inst/
     analyses/
        01-confidential-data-analysis.qmd
        02-unpublished-results.qmd
        README.md  # Document data access requirements
     extdata/
  vignettes/
      01-public-analysis.qmd
      02-demo-with-simulated-data.qmd
```

Benefits of this approach:

- Analyses with restricted data are included in version control alongside your code
- They're clearly separated from public documentation
- `inst/analyses/` is **excluded from `pkgdown` builds** and package documentation
- Collaborators with data access can still run these analyses
- You maintain a complete record of all project work

Note on privacy: Files in `inst/analyses/` are not inherently private—they will be visible if your repository is public. Use this folder for analyses that rely on restricted data that is stored separately, not for storing the restricted data itself. If you need to keep the analysis code private, use a private repository.

Best practices for analyses with restricted data:

1. **Document data requirements**: Include a README.md in `inst/analyses/` explaining:
    - What data is required
    - Where to obtain it (if permissible)
    - Data access restrictions
    - How to set up data paths

2. **Use relative paths carefully**: Structure your code so data paths can be configured:

```
# In inst/analyses/01-analysis.qmd
# Users should set this based on their local setup
data_dir <- Sys.getenv("MYPROJECT_DATA",
                       default = "~/restricted_data/myproject")
raw_data <- readr::read_csv(file.path(data_dir, "sensitive.csv"))
```

3. **Create public alternatives**: When possible, create companion vignettes in `vignettes/` using:
   - Simulated data that mimics the structure
   - Publicly available datasets
   - Aggregated/de-identified summaries
4. **Add to .Rbuildignore**: Ensure `inst/analyses/` doesn't cause package checks to fail:

```
# Use usethis to add to .Rbuildignore
usethis::use_build_ignore("inst/analyses")
```

5.1.4 Keep Analysis Workbooks Tidy

Decompose reusable functions from your analysis notebooks into the `R/` directory. Your vignettes should:

- Be clean, readable narratives of your analysis
- Call well-documented functions from your package
- Focus on the "what" and "why" rather than implementation details
- Be reproducible by others with a single click (or with documented data access for private analyses)

Example of what NOT to do (all code in vignette):

```
# Bad: 100 lines of data manipulation in vignette
raw_data <- read_csv("data.csv")
# ... 100 lines of cleaning, transforming, reshaping ...
cleaned_data <- final_result
```

Example of what TO do (functions in R/, simple calls in vignette):

```
# Good: Clean vignette calling documented functions
raw_data <- read_csv("data.csv")
cleaned_data <- prep_study_data(raw_data)  # Function in R/data_prep.R
```

5.1.5 Shell Scripts and Automation

Shell scripts are useful for automating workflows and ensuring reproducibility. Place shell scripts in `data-raw/` alongside the R scripts they coordinate:

```
data-raw/
   0-prep-data.sh          # Shell script to run all data prep
   01-load-survey.R
   02-clean-survey.R
   03-merge-datasets.R
   04-create-analysis-data.R
```

Using shell scripts:

```
# data-raw/0-prep-data.sh
#!/bin/bash
Rscript data-raw/01-load-survey.R
Rscript data-raw/02-clean-survey.R
Rscript data-raw/03-merge-datasets.R
Rscript data-raw/04-create-analysis-data.R
```

This is especially useful when data upstream changes — you can simply run the shell script to reproduce everything. After running shell scripts, `.Rout` log files will be generated for each script. It is important to check these files to ensure everything has run correctly.

5.1.6 Storing Analysis Outputs

Results files (.RDS): Save analysis results in `inst/extdata/`:

```
# Save results
readr::write_rds(analysis_results, here("inst", "extdata", "analysis_results.rds"))

# Load results later
results <- readr::read_rds(here("inst", "extdata", "analysis_results.rds"))
```

Figures and tables: Save publication outputs in `inst/output/`:

```
# Save figure
ggsave(here("inst", "output", "figures", "fig1_incidence_trends.pdf"),
       width = 8, height = 6)

# Save table
readr::write_csv(summary_table,
                 here("inst", "output", "tables", "table1_demographics.csv"))
```

Organization:

```
inst/
   extdata/
       analysis_results.rds
       model_fits.rds
       processed_data.rds
   output/
       figures/
           fig1_incidence_trends.pdf
           fig2_risk_factors.png
           figS1_sensitivity.pdf
       tables/
            table1_demographics.csv
            table2_main_results.xlsx
            tableS1_detailed_results.csv
```

5.2 `.Rproj` files

An "R Project" can be created within RStudio by going to `File >> New Project`. Depending on where you are with your research, choose the most appropriate option. This will save preferences, working directories, and even the results of running code/data (though I'd recommend starting from scratch each time you open your project, in general). Then, ensure that whenever you are working on that specific research project, you open your created project to enable the full utility of `.Rproj` files. This also automatically sets the directory to the top level of the project.

5.3 Organizing the `data-raw` folder

The `data-raw` folder serves as a catch-all for scripts that do not (yet) fit into the package structure described above. The `data-raw` folder should still be organized. We recommend the following subdirectory structure for `data-raw`:

```
0-run-project.sh
0-config.R
1 - Data-Management/
    0-prep-data.sh
    1-prep-cdph-fluseas.R
    2a-prep-absentee.R
    2b-prep-absentee-weighted.R
    3a-prep-absentee-adj.R
    3b-prep-absentee-adj-weighted.R
2 - Analysis/
    0-run-analysis.sh
    1 - Absentee-Mean/
        1-absentee-mean-primary.R
        2-absentee-mean-negative-control.R
        3-absentee-mean-CDC.R
        4-absentee-mean-peakwk.R
        5-absentee-mean-cdph2.R
```

```
        6-absentee-mean-cdph3.R
    2 - Absentee-Positivity-Check/
    3 - Absentee-P1/
    4 - Absentee-P2/
3 - Figures/
    0-run-figures.sh
    ...
4 - Tables/
    0-run-tables.sh
    ...
5 - Results/
    1 - Absentee-Mean/
        1-absentee-mean-primary.RDS
        2-absentee-mean-negative-control.RDS
        3-absentee-mean-CDC.RDS
        4-absentee-mean-peakwk.RDS
        5-absentee-mean-cdph2.RDS
        6-absentee-mean-cdph3.RDS
    ...
.gitignore
```

For brevity, not every directory is "expanded", but we can glean some important takeaways from what we *do* see.

5.3.1 Configuration ('config') File

This is the single most important file for your project. It will be responsible for a variety of common tasks, declare global variables, load functions, declare paths, and more. *Every other file in the project* will begin with `source("0-config")`, and its role is to reduce redundancy and create an abstraction layer that allows you to make changes in one place (`0-config.R`) rather than 5 different files. To this end, paths which will be reference in multiple scripts (i.e. a `merged_data_path`) can be declared in `0-config.R` and simply referred to by its variable name in scripts. If you ever want to change things, rename them, or even switch from a downsample to the full data, all you would then to need to do is modify the path in one place and the change will automatically update throughout your project. See the example config file for more details. The paths defined in the `0-config.R` file assume that users have opened the `.Rproj` file, which sets the directory to the top level of the project.

5.3.2 Order Files and Directories

This makes the jumble of alphabetized filenames much more coherent and places similar code and files next to one another. This also helps us understand how data flows from start to finish and allows us to easily map a script to its output (i.e. `2 - Analysis/1 - Absentee-Mean/1-absentee-mean-primary.R` => `5 - Results/1 - Absentee-Mean/1-absentee-mean-primary.RDS`). If you take nothing else away from this guide, this is the single most helpful suggestion to make your workflow more coherent. Often the particular order of files will be in flux until an analysis is close to completion. At that time it is important to review file order and naming and reproduce everything prior to drafting a manuscript.

5.3.3 Using Bash scripts to ensure reproducibility

Bash scripts are useful components of a reproducible workflow. At many of the directory levels (i.e. in `3 - Analysis`), there is a bash script that runs each of the analysis scripts. This is exceptionally useful when data "upstream" changes – you simply run the bash script. See Chapter 13 for further details.

After running bash scripts, `.Rout` log files will be generated for each script that has been executed. It is important to check these files. Scripts may appear to have run correctly in the terminal, but checking the log files is the only way to ensure that everything has run completely.

6 R Coding Practices

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, Stephanie Djajadi, and Iris Tong[1]

6.1 Lab Protocols for Code and Data

Just as wet labs have strict safety protocols to ensure reproducible results and prevent contamination, our computational lab has protocols for coding and data management. These protocols are not suggestions—they are essential practices that:

- **Ensure reproducibility**: Others (including your future self) can recreate your analysis
- **Prevent errors**: Systematic approaches reduce the risk of mistakes
- **Enable collaboration**: Consistent practices allow team members to work together efficiently
- **Maintain data integrity**: Proper handling prevents data corruption and loss
- **Support publication**: Well-documented, reproducible code is increasingly required for publication

Violating these protocols can have serious consequences, including invalid results, wasted time, inability to publish, and damage to scientific credibility. Treat coding and data management protocols with the same seriousness as you would safety protocols in a wet lab.

6.2 Essential R Package Development Tools

The following tools are essential for R package development in our lab:

6.2.1 usethis: Package Setup and Management

`usethis` automates common package development tasks:

```
# Install usethis
install.packages("usethis")

# Create a new package
usethis::create_package("~/myproject")

# Add common components
usethis::use_mit_license()          # Add a license
```

[1] https://jadebc.github.io/lab-manual/coding-practices.html

```r
usethis::use_git()                  # Initialize git
usethis::use_github()               # Connect to GitHub
usethis::use_testthat()             # Set up testing infrastructure
usethis::use_vignette("intro")      # Create a vignette (shipped with package)
usethis::use_article("case-study")  # Create an article (website-only)
usethis::use_data_raw("dataset")    # Create data processing script
usethis::use_package("dplyr")       # Add a dependency
usethis::use_pipe()                 # Import magrittr pipe operator (no longer recommended)

# Increment version
usethis::use_version()              # Increment package version
```

6.2.2 devtools: Development Workflow

`devtools` provides the core development workflow:

```r
# Install devtools
install.packages("devtools")

# Load your package for interactive development
devtools::load_all()                # Like library(), but for development

# Documentation
devtools::document()                # Generate documentation from roxygen2

# Testing
devtools::test()                    # Run all tests
devtools::test_active_file()        # Run tests in current file

# Checking
devtools::check()                   # R CMD check (comprehensive validation)
devtools::check_man()               # Check documentation only

# Dependencies
devtools::install_dev_deps()        # Install all development dependencies

# Building
devtools::build()                   # Build package bundle
devtools::install()                 # Install package locally
```

6.2.3 pkgdown: Package Websites

`pkgdown` builds beautiful documentation websites from your package:

```r
# Install pkgdown
install.packages("pkgdown")

# Set up pkgdown
```

```r
usethis::use_pkgdown()

# Build website locally
pkgdown::build_site()

# Preview in browser
pkgdown::build_site(preview = TRUE)

# Build components separately
pkgdown::build_reference()          # Function reference
pkgdown::build_articles()           # Vignettes
pkgdown::build_home()               # Home page from README
```

Configure your pkgdown site with `_pkgdown.yml`:

```yaml
url: https://ucd-serg.github.io/YOURPROJECT

template:
  bootstrap: 5

reference:
  - title: "Data Preparation"
    desc: "Functions for preparing and cleaning data"
    contents:
    - prep_study_data
    - validate_data

  - title: "Analysis"
    desc: "Core analysis functions"
    contents:
    - run_primary_analysis
    - sensitivity_analysis

articles:
  - title: "Analysis Workflow"
    navbar: Analysis
    contents:
    - 01-data-preparation
    - 02-primary-analysis
    - 03-sensitivity-analysis
```

6.3 Complete Package Development Workflow

Here's the typical workflow for developing an R package in our lab:

6.3.1 1. Initial Setup

Starting from a template (recommended):

Using our R package template is the fastest way to get started with a new R package, as it provides pre-configured settings, GitHub Actions workflows, and development tools:

- **UCD-SeRG R Package Template** - Our recommended template with pre-configured development tools and CI workflows:
  - Repository: https://github.com/UCD-SERG/rpt
  - Click "Use this template" → "Create a new repository" on GitHub
  - Clone your new repository and start developing

The template includes pre-configured:

- GitHub Actions workflows for R CMD check, test coverage, and pkgdown deployment
- Development tools setup (`{usethis}`[2], `{devtools}`[3], `{roxygen2}`[4])
- Testing infrastructure (`{testthat}`[5])
- Code styling and linting configurations
- Package documentation structure

While the template jumpstarts your project with up-to-date configuration and workflow files, you should still come up to speed on what all the config files do so you can modify and debug them as needed. The template serves as a central location for the most current versions of these files and best practices.

Starting from scratch:

If you prefer to start from scratch or need to understand each setup step, you can create a new package manually:

```
# Create package structure
usethis::create_package("~/myproject")

# Set up infrastructure
usethis::use_git()
usethis::use_github()
usethis::use_testthat()
usethis::use_pkgdown()
usethis::use_mit_license()
usethis::use_readme_rmd()
```

6.3.2 2. Add Dependencies

```
# Add packages your project depends on
usethis::use_package("dplyr")
usethis::use_package("ggplot2")
usethis::use_package("readr")

# Add packages only needed for development/testing
usethis::use_package("testthat", type = "Suggests")
```

[2] https://usethis.r-lib.org/
[3] https://devtools.r-lib.org/
[4] https://roxygen2.r-lib.org/
[5] https://testthat.r-lib.org/

6.3.3 3. Write Functions

Create functions in `R/` directory with roxygen2 documentation:

```
#' Prepare Study Data
#'
#' Clean and prepare raw study data for analysis.
#'
#' @param raw_data A data frame containing raw study data
#' @param validate Logical; whether to run validation checks
#'
#' @returns A cleaned data frame ready for analysis
#'
#' @examples
#' raw_data <- read_csv("data.csv")
#' clean_data <- prep_study_data(raw_data)
#'
#' @export
prep_study_data <- function(raw_data, validate = TRUE) {
  # Function implementation
}
```

6.3.4 4. Document

```
# Generate documentation from roxygen2 comments
devtools::document()
```

6.3.5 5. Test

Create tests in `tests/testthat/`:

```
# tests/testthat/test-data_prep.R
test_that("prep_study_data handles missing values", {
  raw_data <- data.frame(x = c(1, NA, 3))
  result <- prep_study_data(raw_data)
  expect_false(anyNA(result$x))
})
```

Run tests:

```
devtools::test()
```

6.3.6 6. Check

```
# Comprehensive package check
devtools::check()
```

Fix any warnings or errors before proceeding.

6.3.7 7. Build Documentation Site

```
pkgdown::build_site()
```

6.3.8 8. Share and Publish

```
# Push to GitHub
# The pkgdown site can be automatically deployed to GitHub Pages
# using GitHub Actions
```

```
## Organizing scripts


Just as your data "flows" through your project, data should flow naturally through a script

1. describe the work completed in the script in a comment header
2. source your configuration file (`0-config.R`)
3. load all your data
4. do all your analysis/computation
5. save your data.

Each of these sections should be "chunked together" using comments. See [this file](https:/
Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/2a%
%20Statistical-Inputs.R) for a good example of how to cleanly organize a file in a way that



## Testing Requirements {#sec-r-testing}

**ALWAYS establish tests BEFORE modifying functions.** This ensures changes preserve existi

### When to Use Snapshot Tests

Use snapshot tests (`expect_snapshot()`, `expect_snapshot_value()`) when:

- Testing complex data structures (data frames, lists, model outputs)
- Validating statistical results where exact values may vary slightly
- Output format stability is important

```r
test_that("prep_study_data produces expected structure", {
  result <- prep_study_data(raw_data)
  expect_snapshot_value(result, style = "serialize")
})
```
```

6.3.9 When to Use Explicit Value Tests

Use explicit tests (`expect_equal()`, `expect_identical()`) when:

- Testing simple scalar outputs
- Validating specific numeric thresholds
- Testing Boolean returns or categorical outputs

```
test_that("calculate_mean returns correct value", {
  expect_equal(calculate_mean(c(1, 2, 3)), 2)
  expect_equal(calculate_ratio(3, 7), 0.4285714, tolerance = 1e-6)
})
```

6.3.10 Testing Best Practices

- **Seed randomness**: Use `withr::local_seed()` for reproducible tests
- **Use small test cases**: Keep tests fast
- **Test edge cases**: Missing values, empty inputs, boundary conditions
- **Test errors**: Verify functions fail appropriately with invalid input

```
test_that("prep_study_data handles edge cases", {
  # Empty input
  expect_error(prep_study_data(data.frame()))

  # Missing required columns
  expect_error(prep_study_data(data.frame(x = 1)))

  # Valid input with missing values
  result <- prep_study_data(data.frame(id = 1:3, value = c(1, NA, 3)))
  expect_true(all(!is.na(result$value)))
})
```

6.4 Iterative Operations

When applying analyses with different variations (outcomes, exposures, subgroups), use functional programming approaches:

6.4.1 `lapply()` and `sapply()`

```
# Apply function to each element
results <- lapply(outcomes, function(y) {
  run_analysis(data, outcome = y)
})

# Simplify to vector if possible
summary_stats <- sapply(data_list, mean)
```

6.4.2 `purrr::map()` Family

The `purrr` package provides type-stable alternatives:

```
library(purrr)

# Always returns a list
results <- map(outcomes, ~ run_analysis(data, outcome = .x))

# Type-specific variants
means <- map_dbl(data_list, mean)        # Returns numeric vector
models <- map(splits, ~ lm(y ~ x, data = .x))  # Returns list of models
```

6.4.3 `purrr::pmap()` for Multiple Arguments

When iterating over multiple parameter lists:

```
params <- tibble(
  outcome = c("outcome1", "outcome2", "outcome3"),
  exposure = c("exp1", "exp2", "exp3"),
  covariate_set = list(c("age", "sex"), c("age"), c("age", "sex", "bmi"))
)

results <- pmap(params, function(outcome, exposure, covariate_set) {
  run_analysis(
    data = study_data,
    outcome = outcome,
    exposure = exposure,
    covariates = covariate_set
  )
})
```

6.4.4 Parallel Processing

For computationally intensive work, use `future` and `furrr`:

```
library(future)
library(furrr)

# Set up parallel processing
plan(multisession, workers = availableCores() - 1)

# Parallel version of map()
results <- future_map(large_list, time_consuming_function, .progress = TRUE)
```

6.5 Reading and Saving Data

6.5.1 RDS Files (Preferred)

Use RDS format for R objects:

```
# Save single object
readr::write_rds(analysis_results, here("results", "analysis.rds"))

# Read back
results <- readr::read_rds(here("results", "analysis.rds"))
```

Avoid `.RData` files because: - You can't control object names when loading - Can't load individual objects - Creates confusion in older code

6.5.2 CSV Files

For tabular data that may be shared with non-R users:

```
# Write
readr::write_csv(data, here("data-raw", "clean_data.csv"))

# Read
data <- readr::read_csv(here("data-raw", "clean_data.csv"))

# For very large files, use data.table
data.table::fwrite(large_data, "big_file.csv")
data <- data.table::fread("big_file.csv")
```

6.6 Version Control and Collaboration

6.6.1 Version Numbers

Follow semantic versioning (MAJOR.MINOR.PATCH):

- Development versions: `0.0.0.9000`, `0.0.0.9001`, etc.
- First release: `0.1.0`
- Bug fixes: increment PATCH (e.g., `0.1.0` → `0.1.1`)
- New features: increment MINOR (e.g., `0.1.1` → `0.2.0`)
- Breaking changes: increment MAJOR (e.g., `0.2.0` → `1.0.0`)

```
# Increment version
usethis::use_version()
```

6.6.2 NEWS File

Document all user-facing changes in `NEWS.md`:

```
# myproject 0.2.0

## New features

- Added function for data validation
- Improved error messages

## Bug fixes

- Fixed issue with missing values
- Corrected calculation error in summary stats
```

6.7 Quality Assurance Checklist

Before requesting human review on a pull request or finalizing analysis, verify:

- ☐ All functions have complete roxygen2 documentation
- ☐ All functions have corresponding tests
- ☐ `devtools::document()` has been run
- ☐ `devtools::test()` passes with no failures
- ☐ `devtools::check()` passes with no errors, warnings, or notes
- ☐ `lintr::lint_package()` shows no issues (or only acceptable ones)
- ☐ `spelling::spell_check_package()` passes
- ☐ Version number has been incremented
- ☐ `NEWS.md` has been updated with changes
- ☐ `README.Rmd` has been updated (if needed) and `README.md` regenerated
- ☐ `pkgdown::build_site()` builds successfully
- ☐ All changes committed and pushed to GitHub
- ☐ Copilot review completed iteratively until no valuable suggestions remain (typically 1-3 iterations, with all comments addressed or dismissed)

6.8 Automated Code Styling

6.8.1 RStudio Built-in Formatting

Use RStudio's built-in autoformatter (keyboard shortcut: `CMD-Shift-A` or `Ctrl-Shift-A`) to quickly format highlighted code.

6.8.2 styler Package

For automated styling of entire projects:

```
# Install styler
install.packages("styler")

# Style all files in R/ directory
styler::style_dir("R/")

# Style entire package
styler::style_pkg()

# Note: styler modifies files in-place
# Always use with version control so you can review changes
```

6.8.3 lintr Package

For checking code style without modifying files:

```
# Install lintr
install.packages("lintr")

# Lint the entire package
lintr::lint_package()

# Lint a specific file
lintr::lint("R/my_function.R")
```

The linter checks for:

- Unused variables
- Improper whitespace
- Line length issues
- Style guide violations

You can customize linting rules by creating a `.lintr` file in your project root.

See also Section 8.11.

6.9 Documenting your code

6.9.1 Function headers

Every function you write must include documentation to describe its purpose, inputs, and outputs. For any reproducible workflows, this is essential, because R is dynamically typed. This means you can pass a `string` into an argument that is meant to be a `data.table`, or a `list` into an argument meant for a `tibble`. It is the responsibility of a function's author to document what each argument is meant to do and its basic type.

We use `{roxygen2}`[6] (Wickham et al. 2024) for function documentation. Roxygen2 allows you to describe your functions in special comments next to their definitions, and

[6] https://roxygen2.r-lib.org/

automatically generates R documentation files (`.Rd` files) and helps manage your package `NAMESPACE`. The roxygen2 format uses `#'` comments placed immediately before the function definition.

Here is an example of documenting a function using roxygen2:

```
#' Calculate flu season means by site
#'
#' Make a dataframe with rows for flu season and site
#' containing the number of patients with an outcome, the total patients,
#' and the percent of patients with the outcome.
#'
#' @param data A data frame with variables flu_season, site, studyID, and yname
#' @param yname A string for the outcome name
#' @param silent A boolean specifying whether to suppress console output
#'   (default: TRUE)
#'
#' @returns A dataframe as described above
#'
#' @examples
#' calc_fluseas_mean(my_data, "hospitalized", silent = FALSE)
#'
calc_fluseas_mean <- function(data, yname, silent = TRUE) {
  ### function code here

}
```

The roxygen2 header tells you what the function does, its various inputs, and how you might use it. Also notice that all optional arguments (i.e. ones with pre-specified defaults) follow arguments that require user input.

For more information on roxygen2 syntax and features, see https://roxygen2.r-lib.org/.

6.9.2 Using ... (dots) and `@inheritDotParams`

The ... argument (pronounced "dots" or "ellipsis") is a special R construct that allows functions to accept additional arguments that are passed to other functions. This is particularly useful when creating wrapper functions that call other functions internally.

When to use ...:

- You're creating a wrapper function that calls another function
- You want to allow users to pass additional arguments to an internal function
- You want to provide flexibility without explicitly listing all possible arguments

Basic example with ...:

```
#' Plot data with custom ggplot2 styling
#'
#' A wrapper function that creates a scatter plot with custom theme settings.
#' Additional arguments are passed to ggplot2::geom_point().
#'
```

```r
#' @param data A data frame containing the variables to plot
#' @param x A string specifying the x-axis variable name
#' @param y A string specifying the y-axis variable name
#' @param ... Additional arguments passed to ggplot2::geom_point()
#'
#' @returns A ggplot2 object
#'
#' @examples
#' # Pass color and size arguments to geom_point
#' plot_with_style(my_data, "age", "height", color = "blue", size = 3)
#'
plot_with_style <- function(data, x, y, ...) {
  ggplot2::ggplot(data, ggplot2::aes(.data[[x]], .data[[y]])) +
    ggplot2::geom_point(...) +
    ggplot2::theme_minimal()  # Apply a minimal theme
}
```

While the example above documents `...` with a simple description, roxygen2 provides `@inheritDotParams` to automatically inherit parameter documentation from the function you're calling. This is more robust and maintainable because it automatically stays synchronized with the target function's documentation.

Using `@inheritDotParams`:

```r
#' Plot data with custom ggplot2 styling
#'
#' A wrapper function that creates a scatter plot with custom theme settings.
#'
#' @param data A data frame containing the variables to plot
#' @param x A string specifying the x-axis variable name
#' @param y A string specifying the y-axis variable name
#' @inheritDotParams ggplot2::geom_point -mapping -data -stat -position
#'
#' @returns A ggplot2 object
#'
#' @examples
#' # Pass color and size arguments to geom_point
#' plot_with_style(my_data, "age", "height", color = "blue", size = 3)
#'
plot_with_style <- function(data, x, y, ...) {
  ggplot2::ggplot(data, ggplot2::aes(.data[[x]], .data[[y]])) +
    ggplot2::geom_point(...) +
    ggplot2::theme_minimal()  # Apply a minimal theme
}
```

The `@inheritDotParams` tag:

- Automatically imports parameter documentation from `ggplot2::geom_point()`
- Uses `-mapping -data -stat -position` to exclude parameters that don't make sense in this context
- Keeps documentation synchronized if the underlying function changes

- Makes it clear which function receives the `...` arguments

Best practices for `...`:

1. **Always document what receives the dots**: Use `@inheritDotParams` when passing to a specific function, or clearly describe where the arguments go
2. **Exclude irrelevant parameters**: Use the `-param_name` syntax to exclude parameters that don't apply
3. **Validate unexpected arguments**: Consider using the `{ellipsis}`[7] package to catch misspelled argument names:

```
my_function <- function(x, y, ...) {
  ellipsis::check_dots_used()
  # function code
}
```

4. **Consider alternatives**: If you're only passing a few specific arguments, it may be clearer to list them explicitly rather than using `...`

For more details on `@inheritDotParams`, see the roxygen2 documentation on inheriting parameters[8].

> **Note**
>
> As someone trying to call a function, it is possible to access a function's documentation (and internal code) by `CMD-Left-Click`ing the function's name in RStudio

> **Note**
>
> Depending on how important your function is, the complexity of your function code, and the complexity of different types of data in your project, you can also add "type-checking" to your function with the `assertthat::assert_that()` function. You can, for example, `assert_that(is.data.frame(statistical_input))`, which will ensure that collaborators or reviewers of your project attempting to use your function are using it in the way that it is intended by calling it with (at the minimum) the correct type of arguments. You can extend this to ensure that certain assumptions regarding the inputs are fulfilled as well (i.e. that `time_column`, `location_column`, `value_column`, and `population_column` all exist within the `statistical_input` tibble).

6.9.3 Script headers

Every file in a project that doesn't have roxygen function documentation should at least have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. You may optionally wish to include the inputs and outputs of the script as well, though the next section makes this significantly less necessary.

[7] https://ellipsis.r-lib.org/
[8] https://roxygen2.r-lib.org/articles/rd.html#inheriting-documentation

```
################################################################################
# @Organization - Example Organization
# @Project - Example Project
# @Description - This file is responsible for [...]
################################################################################
```

6.9.4 Sections and subsections

Rstudio (v1.4 or more recent[9]) supports the use of Sections and Subsections. You can easily navigate through longer scripts using the navigation pane in RStudio, as shown on the right below.

```
# Section -------

## Subsection -------

### Sub-subsection -------
```

6.9.5 Code folding

Consider using RStudio's code folding[10] feature to collapse and expand different sections of your code. Any comment line with at least four trailing dashes (-), equal signs (=), or pound signs (#) automatically creates a code section. For example:

6.9.6 Comments in the body of your code

Commenting your code is an important part of reproducibility and helps document your code for the future. When things change or break, you'll be thankful for comments. There's no need to comment excessively or unnecessarily, but a comment describing what a large or complex chunk of code does is always helpful. See this file[11] for an example of how to comment your code and notice that comments are always in the form of:

```
# This is a comment -- first letter is capitalized and spaced away from the pound sign
```

See also Section 8.2 for function documentation style guidelines.

[9] https://blog.rstudio.com/2020/12/02/rstudio-v1-4-preview-little-things/
[10] https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections
[11] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/1b%20-%20Map-Management.R

6.10 Object naming

Generally we recommend using nouns for objects and verbs for functions. This is because functions are performing actions, while objects are not.

Try to make your variable names both more expressive and more explicit. Being a bit more verbose is useful and easy in the age of autocompletion! For example, instead of naming a variable `vaxcov_1718`, try naming it `vaccination_coverage_2017_18`. Similarly, `flu_res` could be named `absentee_flu_residuals`, making your code more readable and explicit.

- For more help, check out Be Expressive: How to Give Your Variables Better Names[12]

We recommend you use **snake_case**.

- Base R allows `.` in variable names and functions (such as `read.csv()`), but this goes against best practices for variable naming in many other coding languages. For consistency's sake, `snake_case` has been adopted across languages, and modern packages and functions typically use it (i.e. `readr::read_csv()`). As a very general rule of thumb, if a package you're using doesn't use `snake_case`, there may be an updated version or more modern package that *does*, bringing with it the variety of performance improvements and bug fixes inherent in more mature and modern software.

> **Note**
>
> You may also see `camelCase` throughout the R code you come across. This is *okay* but not ideal – try to stay consistent across all your code with `snake_case`.

> **Note**
>
> Again, it's also worth noting there's nothing inherently wrong with using `.` in variable names, just that it goes against style best practices that are cropping up in data science, so it's worth getting rid of these bad habits now.

See also Section 8.10.

6.11 Function calls

In a function call, use "named arguments" and put each argument on a separate line to make your code more readable.

Here's an example of what not to do when calling the function a function `calc_fluseas_mean` (defined above):

```
mean_Y = calc_fluseas_mean(flu_data, "maari_yn", FALSE)
```

And here it is again using the best practices we've outlined:

[12] https://spin.atomicobject.com/2017/11/01/good-variable-names/

```
mean_Y <- calc_fluseas_mean(
  data = flu_data,
  yname = "maari_yn",
  silent = FALSE
)
```

6.12 The here package

The `here` package is one great R package that helps multiple collaborators deal with the mess that is working directories within an R project structure. Let's say we have an R project at the path `/home/oski/Some-R-Project`. My collaborator might clone the repository and work with it at some other path, such as `/home/bear/R-Code/Some-R-Project`. Dealing with working directories and paths explicitly can be a very large pain, and as you might imagine, setting up a Config with paths requires those paths to flexibly work for all contributors to a project. This is where the `here` package comes in and this a great vignette describing it[13].

See also Section 8.9 for code style guidelines on using the here package.

6.13 Reading/Saving Data

6.13.1 `.RDS` vs `.RData` Files

One of the most common ways to load and save data in Base R is with the `load()` and `save()` functions to serialize multiple objects in a single `.RData` file. The biggest problems with this practice include an inability to control the names of things getting loaded in, the inherent confusion this creates in understanding older code, and the inability to load individual elements of a saved file. For this, we recommend using the RDS format to save R objects.

> **Note**
>
> If you have many related R objects you would have otherwise saved all together using the `save` function, the functional equivalent with `RDS` would be to create a (named) list containing each of these objects, and saving it.

6.13.2 CSVs

Once again, the `readr` package as part of the Tidvyerse is great, with a much faster `read_csv()` than Base R's `read.csv()`. For massive CSVs (> 5 GB), you'll find `data.table::fread()` to be the fastest CSV reader in any data science language out there. For writing CSVs, `readr::write_csv()` and `data.table::fwrite()` outclass Base R's `write.csv()` by a significant margin as well.

[13] https://github.com/jennybc/here_here

6.14 Integrating Box and Dropbox

Box and Dropbox are cloud-based file sharing systems that are useful when dealing with large files. When our scripts generate large output files, the files can slow down the workflow if they are pushed to GitHub. This makes collaboration difficult when not everyone has a copy of the file, unless we decide to duplicate files and share them manually. The files might also take up a lot of local storage. Box and Dropbox help us avoid these issues by automatically storing the files, reading data, and writing data back to the cloud.

Box and Dropbox are separate platforms, but we can use either one to store and share files. To use them, we can install the packages that have been created to integrate Box and Dropbox into R. The set-up instructions are detailed below.

Make sure to authenticate before reading and writing from either Box or Dropbox. The authentication commands should go in the configuration file; it only needs to be done once. This will prompt you to give your login credentials for Box and Dropbox and will allow your application to access your shared folders.

6.14.1 Box

Follow the instructions in this section to use the `boxr` package. Note that there are a few setup steps that need to be done on the box website before you can use the `boxr` package, explained here[14] in the section "Creating an Interactive App." This gets the authentication keys that must be put in box. Once that is done, add the authentication keys to your code in the configuration file, with `box_auth(client_id = "<your_client_id>", client_secret = "<your_client_secret_id>")`. It is also important to set the default working directory so that the code can reference the correct folder in box: `box_setwd(<folder_id>)`. The folder ID is the sequence of digits at the end of the URL.

Further details can be found here[15].

6.14.2 Dropbox

Follow the instructions at this link[16] to use the `rdrop2` package. Similar to the `boxr` package, you must authenticate before reading and writing from Dropbox, which can be done by adding `drop_auth()` to the configuration file.

Saving the authentication token is not required, although it may be useful if you plan on using Dropbox frequently. To do so, save the token with the following commands. Tokens are valid until they are manually revoked.

```
# first time only
# save the output of drop_auth to an RDS file
token <- drop_auth()
# this token only has to be generated once, it is valid until revoked
saveRDS(token, "/path/to/tokenfile.RDS")

# all future usages
```

[14] https://r-box.github.io/boxr/articles/boxr-app-interactive.html#create
[15] https://github.com/r-box/boxr
[16] https://github.com/karthik/rdrop2

```
# to use a stored token, provide the rdstoken argument
drop_auth(rdstoken = "/path/to/tokenfile.RDS")
```

6.15 Tidyverse

Throughout this document there have been references to the Tidyverse, but this section is to explicitly show you how to transform your Base R tendencies to Tidyverse (or Data.Table, Tidyverse's performance-optimized competitor). For most of our work that does not utilize very large datasets, we recommend that you code in Tidyverse rather than Base R. Tidyverse is quickly becoming the gold standard[17] in R data analysis and modern data science packages and code should use Tidyverse style and packages unless there's a significant reason not to (i.e. big data pipelines that would benefit from Data.Table's performance optimizations). Note that `{dtplyr}`[18] provides a data.table backend for dplyr, enabling you to use most of dplyr's tidy syntax with data.table's performance optimizations.

The package author has published R for Data Science (Wickham, Çetinkaya-Rundel, and Grolemund 2023), which leans heavily on many Tidyverse packages and may be worth checking out.

6.16 Core Tidyverse Packages

The tidyverse[19] is a collection of R packages designed for data science that share an underlying design philosophy, grammar, and data structures. As of tidyverse 1.3.0, the following nine packages are included in the core tidyverse and are loaded automatically when you run `library(tidyverse)`:

6.16.1 ggplot2[20]

`{ggplot2}`[21] is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics and what graphical primitives to use, and it takes care of the details.

6.16.2 dplyr[22]

`{dplyr}`[23] provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges. Key functions include `filter()`, `select()`, `mutate()`, `summarize()`, and `arrange()`.

[17] https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/
[18] https://dtplyr.tidyverse.org/
[19] https://www.tidyverse.org/
[20] https://ggplot2.tidyverse.org/
[21] https://ggplot2.tidyverse.org/
[22] https://dplyr.tidyverse.org/
[23] https://dplyr.tidyverse.org/

6.16.3 tidyr[24]

`{tidyr}`[25] provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable. Key functions include `pivot_longer()`, `pivot_wider()`, `separate()`, and `unite()`.

6.16.3.1 When to use dplyr vs tidyr

While both `{dplyr}`[26] and `{tidyr}`[27] work with data frames, they serve different purposes:

- **Use dplyr** for data manipulation within the current structure: filtering rows, selecting columns, creating new variables, summarizing data, or joining datasets. These operations work with your data as-is.
- **Use tidyr** for reshaping your data structure itself: converting between wide and long formats (`pivot_longer()`, `pivot_wider()`), splitting or combining columns (`separate()`, `unite()`), or handling missing values explicitly (`complete()`, `fill()`).

These packages work together seamlessly in data analysis workflows. A typical pattern is to use tidyr to reshape your data into the right structure, then use dplyr to manipulate and analyze it.

6.16.4 readr[28]

`{readr}`[29] provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes.

6.16.5 purrr[30]

`{purrr}`[31] enhances R's functional programming[32] (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. Once you master the basic concepts, purrr allows you to replace many for loops with code that is easier to write and more expressive. See Section 6.4 for more details on using purrr.

[24] https://tidyr.tidyverse.org/
[25] https://tidyr.tidyverse.org/
[26] https://dplyr.tidyverse.org/
[27] https://tidyr.tidyverse.org/
[28] https://readr.tidyverse.org/
[29] https://readr.tidyverse.org/
[30] https://purrr.tidyverse.org/
[31] https://purrr.tidyverse.org/
[32] https://en.wikipedia.org/wiki/Functional_programming

6.16.6 tibble[33]

`{tibble}`[34] is a modern re-imagining of the data frame, keeping what time has proven to be effective, and throwing out what it has not. Tibbles are data.frames that are lazy and surly: they do less and complain more forcing you to confront problems earlier, typically leading to cleaner, more expressive code.

6.16.7 stringr[35]

`{stringr}`[36] provides a cohesive set of functions designed to make working with strings as easy as possible. It is built on top of stringi, which uses the ICU C library to provide fast, correct implementations of common string manipulations.

6.16.8 forcats[37]

`{forcats}`[38] provides a suite of useful tools that solve common problems with factors. R uses factors to handle categorical variables, variables that have a fixed and known set of possible values.

6.16.9 lubridate[39]

`{lubridate}`[40] provides a set of functions for working with date-times, extending and improving on R's existing support for them. Key functions include `ymd()`, `mdy()`, `dmy()` for parsing dates, and `year()`, `month()`, `day()` for extracting components.

6.17 Base R to Tidyverse Translation

The following list is not exhaustive, but is a compact overview to begin to translate Base R into something better:

Base R	Better Style, Performance, and Utility
—	—
`read.csv()`	`readr::read_csv()` or `data.table::fread()`
`write.csv()`	`readr::write_csv()` or `data.table::fwrite()`
`readRDS`	`readr::read_rds()`
`saveRDS()`	`readr::write_rds()`
—	—
`data.frame()`	`tibble::tibble()` or `tibble::tribble()`

[33] https://tibble.tidyverse.org/
[34] https://tibble.tidyverse.org/
[35] https://stringr.tidyverse.org/
[36] https://stringr.tidyverse.org/
[37] https://forcats.tidyverse.org/
[38] https://forcats.tidyverse.org/
[39] https://lubridate.tidyverse.org/
[40] https://lubridate.tidyverse.org/

Base R	Better Style, Performance, and Utility
`rbind()`	`dplyr::bind_rows()`
`cbind()`	`dplyr::bind_cols()`
`df$some_column`	`df |> dplyr::pull(some_column)`
`df$some_column = ...`	`df |> dplyr::mutate(some_column = ...)`
`df[get_rows_condition,]`	`df |> dplyr::filter(get_rows_condition)`
`df[,c(col1, col2)]`	`df |> dplyr::select(col1, col2)`
`merge(df1, df2, by = ..., all.x = ..., all.y = ...)`	`df1 |> dplyr::left_join(df2, by = ...)` or `dplyr::full_join` or `dplyr::inner_join` or `dplyr::right_join`
—	—
`str()`	`dplyr::glimpse()`
`grep(pattern, x)`	`stringr::str_which(string, pattern)`
`gsub(pattern, replacement, x)`	`stringr::str_replace(string, pattern, replacement)`
`ifelse(test_expression, yes, no)`	`if_else(condition, true, false)`
Nested: `ifelse(test_expression1, yes1, ifelse(test_expression2, yes2, ifelse(test_expression3, yes3, no)))`	`case_when(test_expression1 ~ yes1, test_expression2 ~ yes2, test_expression3 ~ yes3, TRUE ~ no)`
`proc.time()`	`tictoc::tic()` and `tictoc::toc()`
`stopifnot()`	`assertthat::assert_that()` or `assertthat::see_if()` or `assertthat::validate_that()`
—	—
`sessionInfo()`	`sessioninfo::session_info()`

For a more extensive set of syntactical translations to Tidyverse, you can check out this document[41].

6.18 Programming with Tidyverse

Working with Tidyverse within functions can be somewhat of a pain due to non-standard evaluation (NSE) semantics. If you're an avid function writer, we'd recommend checking out the following resources:

- Programming with dplyr[42] (package vignette)
- Using dplyr in packages[43] (package vignette)
- Tidy Eval in 5 Minutes[44] (video)
- Tidy Evaluation[45] (e-book)

[41] https://tavareshugo.github.io/data_carpentry_extras/base-r_tidyverse_equivalents/base-r_tidyverse_equivalents.html
[42] https://dplyr.tidyverse.org/articles/programming.html
[43] https://dplyr.tidyverse.org/articles/in-packages.html
[44] https://www.youtube.com/watch?v=nERXS3ssntw
[45] https://dplyr.tidyverse.org/articles/programming.html

- Evaluation[46] (advanced details)
- Data Frame Columns as Arguments to Dplyr Functions[47] (blog)
- Standard Evaluation for *_join[48] (stackoverflow)

See also Section 8.8

6.19 Coding with R and Python

If you're using both R and Python, you may wish to check out the Feather package[49] for exchanging data between the two languages extremely quickly[50].

6.20 Repeating analyses with different variations

In many cases, we will need to apply our modeling on different combinations of interests (outcomes, exposures, etc.). We can certainly use a `for` loop to repeat the execution of a wrapper function, but generally, `for` loops request high memory usage and produce the results in long computation time.

Fortunately, R has some functions which implement looping in a compact form to help repeating your analyses with different variations (subgroups, outcomes, covariate sets, etc.) with better performances.

6.20.1 `lapply()` and `sapply()`

`lapply()` is a function in the base R package that applies a function to each element of a list and returns a list. It's typically faster than `for`. Here is a simple generic example:

```
result <- lapply(X = mylist, FUN = func)
```

There is another very similar function called `sapply()`. It also takes a list as its input, but if the output of the `func` is of the same length for each element in the input list, then `sapply()` will simplify the output to the simplest data structure possible, which will usually be a vector.

6.20.2 `mapply()` and `pmap()`

Sometimes, we'd like to employ a wrapper function that takes arguments from multiple different lists/vectors. Then, we can consider using `mapply()` from the base R package or `pmap()` from the `purrr` package.

Please see the simple specific example below where the two input lists are of the same length and we are doing a pairwise calculation:

[46] https://adv-r.hadley.nz/evaluation.html
[47] https://www.brodrigues.co/blog/2016-07-18-data-frame-columns-as-arguments-to-dplyr-functions/
[48] https://stackoverflow.com/questions/28125816/r-standard-evaluation-for-join-dplyr
[49] https://www.rdocumentation.org/packages/feather/versions/0.3.3
[50] https://blog.rstudio.com/2016/03/29/feather/

```
mylist1 = list(0:3)
mylist2 = list(6:9)
mylists = list(mylist1, mylist2)

square_sum <- function(x, y) {
  x^2 + y^2
}

#Use `mapply()`
result1 <- mapply(FUN = square_sum, mylist1, mylist2)

#Use `pmap()`
library(purrr)
result2 <- pmap(.l = mylists, .f = square_sum)

#unlist(as.list(result1)) = result2 = [36 50 68 90]
```

There are two major differences between `mapply()` and `pmap()`. The first difference is that `mapply()` takes seperate lists as its input arguments, while `pmap()` takes a list of list. Secondly, the output of `mapply()` will be in the form of a matrix or an array, but `pmap()` produces a list directly.

However, when **the input lists are of different lengths AND/OR the wrapper function doesn't take arguments in pairs, `mapply()` and `pmap()` may not give the preferable results.**

Both `mapply()` and `pmap()` will recycle shorter input lists to match the length of the longest input list. Assume that now `mylist2 = list(6:12)`. Then, `pmap(mylists, square_sum)` will generate `[36  50  68  90 100 122 148]` where elements 0, 1, and 2 are recycled to match 10, 11, and 12. And it will return an error message that "longer object length is not a multiple of shorter object length."

Thus, unless the recycling pattern described above is desirable feature for a certain experiment design, **when the input lists are of different lengths, the best practice is probably to use `lapply()` and then combine the results.**

Here is an example where we'd like to find the `square_sum` for every element combination of `mylist1` and `mylist2`.

```
mylist1 <- list(0:3)
mylist2 <- list(6:12)

square_sum <- function(x, y) {
  x^2 + y^2
}

results <- list()

for (i in seq_along(mylist1[[1]])) {
  result <- lapply(X = mylist2, FUN = function(y) square_sum(mylist1[[1]][i], y))
  results[[i]] <- result
}
```

This example doesn't work in the way that 0 is paired to 6, 1 is paired to 7, and so on. Instead, every element in `mylist1` will be paired with every element in `mylist2`. Thus, the "unlisted" results from the example will have $4 * 7 = 28$ elements.

We can use `flatten()` or `unlist()` functions to decrease the depths of our results. If the results are data frames, then we will need to use `bind_rows()` to combine them.

6.20.3 Parallel processing with `parallel` and `future` packages

One big drawback of `lapply()` is its long computation time, especially when the list length is long. Fortunately, computers nowadays must have multiple cores which makes parallel processing possible to help make computation much faster.

Assume you have a list called `mylist` of length 1000, and `lapply(X = mylist, FUN = func)` applies the function to each of the 1000 elements one by one in T seconds. If we could execute the `func` in n processors simultaneously, then ideally, we would shrink the computation time to T/n seconds.

In practice, using functions under the `parallel` and the `future` packages, we can split `mylist` into smaller chunks and apply the function to each element of the several chunks in parallel in different cores to significantly reduce the run time.

6.20.3.1 `parLapply()`

Below is a generic example of `parLapply()`:

```
library(parallel)

# Set how many processors will be used to process the list and make cluster
n_cores <- 4
cl <- makeCluster(n_cores)

#Use parLapply() to apply func to each element in mylist
result <- parLapply(cl = cl, x = mylist, FUN = func)

#Stop the parallel processing
stopCluster(cl)
```

Let's still assume `mylist` is of length 1000. The `parLapply` above splits `mylist` into 4 sub-lists each of length 250 and applies the function to the elements of each sub-list in parallel. To be more specific, first apply the function to element 1, 251, 501, 751; second apply to element 2, 252, 502, 752; so on and so forth. As such, the computation time will be greatly reduced.

You can use `parallel::detectCores()` to test how many cores your machine has and to help decide what to put for `n_cores`. It would be a good idea to leave at least one core free for the operating system to use.

We will always start `parLapply()` with `makeCluster()`. **`stopCluster()` is not fully necessary but follows the best practices.** If not stopped, the processing will continue in the back end and consuming the computation capacity for other software in your machine.

But keep in mind that stopping the cluster is similar quitting R, meaning that you will need to re-load the packages needed when you need to do parallel processing use `parLapply()` again.

6.20.3.2 `future.lapply()`

Below is a generic example of `future.lapply()`:

```
library(future)
library(future.apply)

# First, plan how the future_lapply() will be resolved
future::plan(
  multisession, workers = future::availableCores() - 1
)

# Use future_lapply() to apply func to each element in mylist
future_lapply(x = mylist, FUN = func)
```

Here, `future::availableCores()` checks how many cores your machine has. Similar to `parLapply()` showed above, `future_lapply()` parallelizes the computation of `lapply()` by executing the function `func` simultaneously on different sub-lists of `mylist`.

6.21 Reviewing Code

Before publishing new changes, it is important to ensure that the code has been tested and well-documented. GitHub makes it possible to document all of these changes in a pull request. Pull requests can be used to describe changes in a branch that are ready to be merged with the base branch (more information in the GitHub section).

This section provides guidance on both constructing effective pull requests and reviewing code submitted by others. Much of the content in this section is adapted from the Tidyverse code review principles (Tidyverse Team 2023), which provides excellent principles for code review in R package development.

6.22 Constructing Pull Requests

6.22.1 Write Focused PRs

A focused pull request is **one self-contained change** that addresses just one thing. Writing focused PRs has several benefits:

- **Faster reviews**: It's easier for a reviewer to find 5-10 minutes to review a single bug fix than to set aside an hour for one large PR implementing many features.
- **More thorough reviews**: Large PRs with many changes can overwhelm reviewers, leading to important points being missed.
- **Fewer bugs**: Smaller changes make it easier to reason about impacts and identify potential issues.

- **Easier to merge**: Large PRs take longer and are more likely to have merge conflicts.
- **Less wasted work**: If the overall direction is wrong, you've wasted less time on a small PR.

As a guideline, 100 lines is usually a reasonable size for a PR, and 1000 lines is usually too large. However, the number of files affected also matters—a 200-line change in one file might be fine, but the same change spread across 50 files is usually too large.

6.22.2 Writing PR Descriptions

When you submit a pull request, include a detailed PR title and description. A comprehensive description helps your reviewer and provides valuable historical context.

PR Title: The title should be a short summary (ideally under 72 characters) of what is being done. It should be informative enough that future developers can understand what the PR did without reading the full description.

Poor titles that lack context:

- "Fix bug"
- "Add patch"
- "Moving code from A to B"

Better titles that summarize the actual change:

- "Fix missing value handling in data processing function"
- "Add support for custom date formats in import functions"

PR Description Body: The description should provide context that helps the reviewer understand your PR. Consider including:

- A brief description of the problem being solved
- Links to related issues (e.g., "Closes #123" or "Related to #456")
- A before/after example showing changed behavior
- Possible shortcomings of the approach being used
- For complex PRs, a suggested reading order for the reviewer
- The `Files` tab of a Pull Request page on GitHub allows you to annotate your pull request with inline comments. These comments are not part of the source files; they only exist in GitHub's metadata. Use these comments to explain *changes* whose reasoning might not be self-apparent to a reviewer.

6.22.3 Add Tests

Focused PRs should include related test code. A PR that adds or changes logic should be accompanied by new or updated tests for the new behavior. Pure refactoring PRs should also be covered by tests—if tests don't exist for code you're refactoring, add them in a separate PR first to validate that behavior is unchanged.

6.22.4 Separate Out Refactorings

It's usually best to do refactorings in a separate PR from feature changes or bug fixes. For example, moving and renaming a function should be in a different PR from fixing a bug in that function. This makes it much easier for reviewers to understand the changes introduced by each PR.

Small cleanups (like fixing a local variable name) can be included in a feature change or bug fix PR, but large refactorings should be separate.

6.23 Reviewing Pull Requests

6.23.1 Purpose of Code Review

The primary purpose of code review is to ensure that the overall code health of our projects improves over time. Reviewers should balance the need to make forward progress with the importance of maintaining code quality.

Key principle: Reviewers should favor approving a PR once it is in a state where it definitely improves the overall code health of the system, even if the PR isn't perfect. There is no such thing as "perfect" code—there is only better code. Rather than seeking perfection, seek continuous improvement.

6.23.2 Monitoring PRs Awaiting Your Review

To ensure timely code reviews, bookmark GitHub's review-requested page and check it regularly (at least daily):

- **General bookmark**: https://github.com/pulls/review-requested shows all PRs across GitHub where you've been requested as a reviewer
- **Project-specific bookmark**: For frequently-reviewed repositories, you can bookmark project-specific versions using GitHub's search syntax. For example, to see PRs awaiting your review in this repository: https://github.com/UCD-SERG/lab-manual/pulls/review-requested/YOUR-USERNAME (replace `YOUR-USERNAME` with your GitHub username)

Checking these pages regularly helps ensure that PRs don't languish waiting for review, which is important for maintaining team productivity and code quality.

6.23.3 Writing Review Comments

When reviewing code, maintain courtesy and respect while being clear and helpful:

- Comment on the **code**, not the author
- Explain **why** you're making suggestions (reference best practices, design patterns, or how the suggestion improves code health)
- Balance pointing out problems with providing guidance (help authors learn while being constructive)
- Highlight positive aspects too—if you see good practices, comment on those to reinforce them

Poor comment: "Why did you use this approach when there's obviously a better way?"

Better comment: "This approach adds complexity without clear benefits. Consider using [alternative approach] instead, which would simplify the logic and improve readability."

6.23.4 Mentoring Through Review

Code review is an excellent opportunity for mentoring. As a reviewer:

- Leave comments that help authors learn something new
- Link to relevant sections of style guides or best practices documentation
- Consider pair programming for complex reviews—live review sessions can be very effective for teaching

6.23.5 Giving Constructive Feedback

In general, it is the author's responsibility to fix a PR, not the reviewer's. Strike a balance between pointing out problems and providing direct guidance. Sometimes pointing out issues and letting the author decide on a solution helps them learn and may result in a better solution since they are closer to the code.

For very small tweaks (typos, comment additions), use GitHub's suggestion feature to allow authors to quickly accept changes directly in the UI.

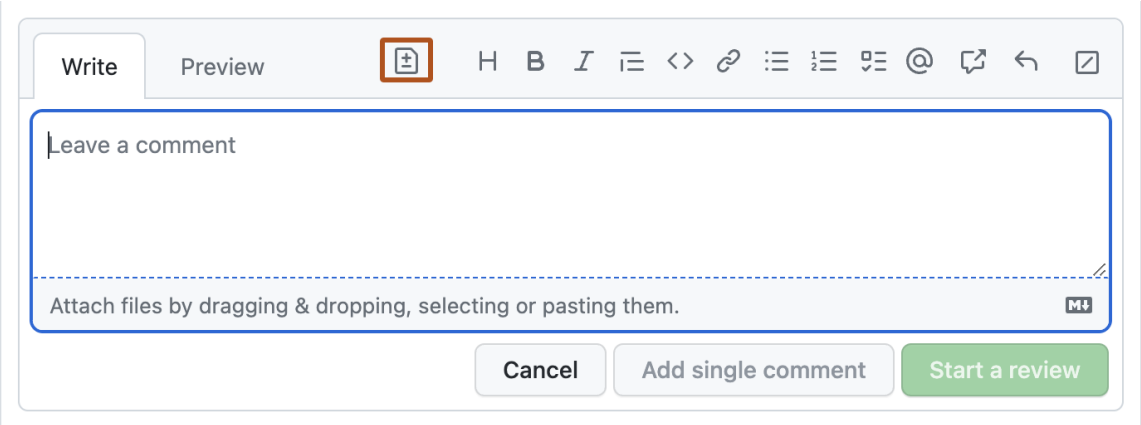

Figure 6.1: GitHub's suggestion feature in a PR review comment

6.23.6 Ignoring Auto-Generated Files

When reviewing pull requests in R package repositories, you can typically ignore changes to `.Rd` files in the `man/` directory. These are R documentation files automatically generated by `{roxygen2}`[51] from special comments in the R source code (see Section 6.9).

Why ignore `.Rd` files?

- They are auto-generated and should never be manually edited
- Changes to `.Rd` files simply reflect changes already visible in the roxygen2 comments
- Reviewing the source roxygen2 comments is more informative and efficient

[51] https://roxygen2.r-lib.org/

- The `.Rd` files will be regenerated during the package build process

What to review instead:

Focus your review on the roxygen2 documentation comments in the actual R source files (`.R` files in the `R/` directory). These special comments start with `#'` and appear immediately before function definitions. Any changes to function documentation will be visible there.

If the repository has a preview workflow (such as pkgdown for R packages or Quarto for documentation sites), you can also review the rendered documentation in the preview build. The preview workflow should automatically post a comment on the PR containing a link to a preview version of the revised documentation.

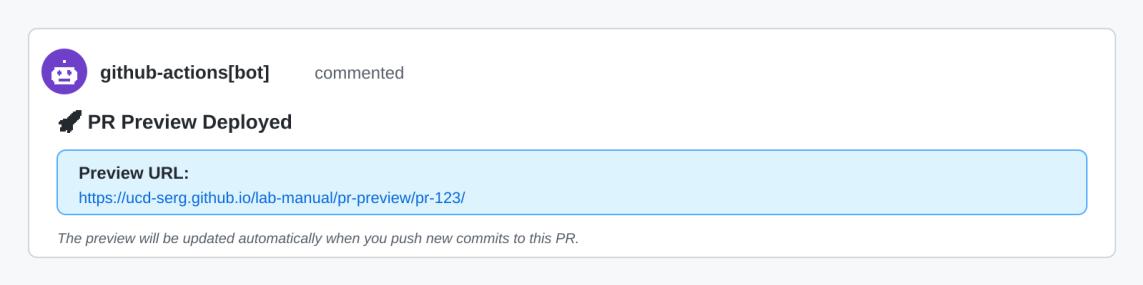

Figure 6.2: Example of an automated PR preview comment posted by GitHub Actions

GitHub review tip:

In GitHub's pull request "Files changed" view, you can click the three dots (...) next to a file and select "View file" to hide it from the diff view. This helps you focus on the meaningful changes.

6.23.7 Reviewing Copilot-Generated Pull Requests

When reviewing pull requests created by GitHub Copilot coding agents, apply the same standards and principles as any other PR, but be aware of some unique considerations:

Workflow approval requirements:

- **You must manually approve GitHub Actions workflows** for Copilot PRs
- This is a security measure because Copilot can modify any file, including workflow files themselves
- Click the approval button in the Actions tab or on the PR to trigger workflows
- There is currently no way to bypass this manual approval, even if you are the repository owner

Review focus areas:

- **Verify the solution addresses the issue**: Ensure Copilot understood the requirements correctly
- **Check for over-engineering**: Copilot may sometimes add unnecessary complexity or features beyond what was requested
- **Review test coverage**: Verify that tests are appropriate and comprehensive
- **Check documentation**: Ensure documentation is clear and follows project conventions

- **Look for edge cases**: AI-generated code may miss edge cases or error handling

Iterating on Copilot PRs:

When you find issues in a Copilot PR, you have two options:

1. **Request changes from Copilot**: Leave review comments and ask Copilot to address them. This works well for complex changes or when you want to see how Copilot interprets your feedback.
2. **Make direct changes yourself**: Push commits directly to the Copilot PR branch. This is often faster for simple fixes like typos, formatting, or small adjustments.

For quick fixes, you can often make changes faster than writing review comments and waiting for Copilot to respond.

Best practices:

- **Don't push while Copilot is working**: Wait for Copilot to complete its current iteration before pushing your own changes to avoid merge conflicts
- **Review incrementally**: If a Copilot PR is large, review it in stages as the agent updates it rather than waiting until the end
- **Trust but verify**: Copilot is a powerful tool, but human review is essential for catching issues and ensuring quality

6.24 Creating a Pull Request Template

GitHub allows you to create a pull request template in a repository to standardize the information in pull requests. When you add a template, everyone will automatically see its contents in the pull request body.

Follow these steps to add a pull request template:

1. On GitHub, navigate to the main page of the repository.
2. Above the file list, click `Create new file`.
3. Name the file `pull_request_template.md`. GitHub will not recognize this as the template if it is named anything else. The file must be on the default branch.
    - To store the file in a hidden directory, name it `.github/pull_request_template.md`.
4. In the body of the new file, add your pull request template.

Here is an example pull request template:

```
# Description

## Summary of change

Please include a summary of the change, including any new functions added and example usage

## Related Issues

Closes #(issue number)
Related to #(issue number)
```

```
## Testing

Describe how this change has been tested.

## Checklist

- [ ] Tests added/updated
- [ ] Documentation updated
- [ ] Code follows project style guidelines

## Who should review the pull request?

@username
```

6.25 Getting Help with Code

When you encounter a coding problem, creating a **reprex** (minimal **repr**oducible **ex**ample) is one of the most effective ways to get help—and often helps you solve the problem yourself.

A good reprex (Bryan et al. 2024):

- Is **reproducible**: Contains all necessary code, including `library()` calls and data
- Is **minimal**: Strips away everything not directly related to your problem
- Uses small, simple example data (often built-in datasets)

Why create a reprex:

- 80% of the time, creating a reprex helps you discover the solution yourself
- 20% of the time, you'll have a clear example that makes it easy for others to help you
- It respects others' time by making your problem easy to understand and reproduce

Resources:

- `{reprex}`[52] package: Automates creation of reproducible examples
- R for Data Science: Making a reprex[53] (Wickham, Çetinkaya-Rundel, and Grolemund 2023): Step-by-step guide to creating effective reproducible examples

6.26 Additional Resources

6.26.1 R Package Development

- R Packages (Wickham and Bryan 2023) - comprehensive guide to R package development
- Tidyverse design guide (Wickham 2023b) - principles for designing R packages and APIs that are intuitive, composable, and consistent with tidyverse philosophy

[52] https://reprex.tidyverse.org/
[53] https://r4ds.hadley.nz/workflow-help.html#making-a-reprex

- usethis documentation[54] - workflow automation for R projects
- devtools documentation[55] - essential development tools
- pkgdown documentation[56] - create package websites
- testthat documentation[57] - unit testing framework

6.26.2 General R Programming

- R for Data Science (Wickham, Çetinkaya-Rundel, and Grolemund 2023) - learn data science with the tidyverse
- Advanced R (Wickham 2019) - deep dive into R programming and internals

6.26.3 Shiny Development

- Mastering Shiny (Wickham 2021) - comprehensive guide to building web applications with Shiny
- Engineering Production-Grade Shiny Apps (Fay et al. 2021) - best practices for production Shiny applications

6.26.4 Git and Version Control

- Happy Git and GitHub for the useR (Bryan 2023) - essential guide to using Git and GitHub with R

[54] https://usethis.r-lib.org/
[55] https://devtools.r-lib.org/
[56] https://pkgdown.r-lib.org/
[57] https://testthat.r-lib.org/

7 Continuous Integration

7.1 Understanding GitHub Actions

GitHub Actions[1] is GitHub's built-in automation platform that makes it easy to automate software workflows, including continuous integration and deployment (CI/CD). For R packages, this means you can automatically test your code, check for errors, and deploy documentation every time you push changes to GitHub.

Key benefits of GitHub Actions:

- **Automated testing**: Run `R CMD check` across multiple operating systems (Linux, macOS, Windows) and R versions
- **Immediate feedback**: Get notified of problems quickly, when they're easier to fix
- **Better collaboration**: External contributors can see if their changes pass all checks before you review
- **Quality assurance**: Catch platform-specific issues before they reach users
- **Documentation deployment**: Automatically build and deploy your pkgdown website

Even for solo developers, having automated checks run on different platforms helps avoid the "works on my machine" problem.

7.2 Setting Up GitHub Actions

The easiest way to add GitHub Actions to your R package is using `{usethis}`. The tidyverse team maintains a collection of ready-to-use workflows at r-lib/actions[2] that handle common R package tasks.

7.2.1 Essential Workflows

1. R CMD check (most important):

```
usethis::use_github_action("check-standard")
```

This runs `R CMD check` on Linux, macOS, and Windows to ensure your package works across platforms. If you only set up one workflow, make it this one.

2. Test coverage:

[1] https://github.com/features/actions
[2] https://github.com/r-lib/actions

```
usethis::use_github_action("test-coverage")
```

Calculates what percentage of your code is covered by tests and reports to codecov.io[3].

3. Package website:

```
usethis::use_github_action("pkgdown")
```

Automatically builds and deploys your pkgdown documentation site to GitHub Pages.

7.2.2 Interactive Setup

Running `usethis::use_github_action()` without arguments shows a menu of recommended workflows:

```
usethis::use_github_action()
#> Which action do you want to add? (0 to exit)
#> (See <https://github.com/r-lib/actions/tree/v2/examples> for other options)
#>
#> 1: check-standard: Run `R CMD check` on Linux, macOS, and Windows
#> 2: test-coverage: Compute test coverage and report to https://about.codecov.io
#> 3: pr-commands: Add /document and /style commands for pull requests
```

7.3 How GitHub Actions Workflows Work

When you set up a workflow, `usethis` creates a YAML configuration file in `.github/workflows/`. For example, `check-standard` creates `.github/workflows/R-CMD-check.yaml`.

This workflow automatically runs when you:

- Push commits to `main` or `master`
- Open or update a pull request

You can view workflow results in the "Actions" tab of your GitHub repository. A status badge is added to your README showing whether checks are passing.

7.4 Workflow Files and Security

> **Warning**
>
> **Important Security Consideration**
> Workflow files (`.github/workflows/*.yaml`) have access to repository secrets and can execute code. Always review workflow files carefully before committing them, especially if copied from external sources.
> See Section 18.5.8 for guidance on working with workflow files using AI tools.

[3] https://about.codecov.io

The workflow YAML files in `.github/workflows/` are configuration files that tell GitHub Actions:

- When to run (on push, pull request, schedule, etc.)
- What operating systems and R versions to use
- What steps to execute (install dependencies, run checks, etc.)

7.5 Troubleshooting Failed Workflows

When workflows fail, check the "Actions" tab in your GitHub repository for detailed logs. Common issues include:

- **Test failures**: Your tests found a bug (this is good! fix the bug)
- **Platform-specific issues**: Code works on your machine but not on other platforms
- **Missing dependencies**: System libraries needed for packages aren't installed
- **Linting errors**: Code style issues detected by automated checks

For help addressing workflow failures, see Section 18.5.6.

7.6 Pull Request Comment Automation

GitHub Actions can automatically comment on pull requests to provide feedback, status updates, or deployment previews. This section compares commonly used actions for managing PR comments, helping you choose the right tool for your workflow.

7.6.0.1 Common Use Cases

PR comment automation is particularly useful for:

- **CI/CD status updates**: Report test results, build status, or deployment progress
- **Code quality reports**: Post coverage reports, linting results, or security scan findings
- **Deployment previews**: Share links to preview deployments (e.g., documentation sites, app previews)
- **Bot feedback**: Provide automated feedback without cluttering the PR conversation

7.6.0.2 Comparison of Popular Actions

marocchino/sticky-pull-request-comment[4]

A widely-used action for creating or updating a single comment per workflow (as of early 2026, ~580 GitHub stars). Prevents comment spam by updating the same comment each time the workflow runs.

Key features:

- **Sticky comments**: Creates or updates a comment identified by a unique header

[4] https://github.com/marocchino/sticky-pull-request-comment

- **Multiple independent comments**: Different workflows can maintain separate sticky comments using different headers
- **Flexible update modes**: Replace, append, recreate, delete, or hide comments
- **File-based messages**: Load comment content from files for complex templates
- **Works with push events**: Can find and comment on PRs from push triggers (useful for monorepos)

Typical usage:

```yaml
- uses: marocchino/sticky-pull-request-comment@v2
  with:
    header: test-results
    message: |
      ## Test Results
      ```
      ${{ steps.test.outputs.summary }}
      ```
```

Best for: Projects needing clean, updatable status comments without duplicates. Ideal when you want the same type of information always visible in one place.

hasura/comment-progress[5]

Designed for tracking workflow progress with multiple updates as jobs complete. Similar to how Netlify or SonarCloud bots provide progressive feedback.

Key features:

- **Progress tracking**: Update comments as workflow steps complete or fail
- **Identifier-based updates**: Uses a hidden identifier to find and update the correct comment
- **Multiple update modes**: Append to existing comments, recreate, or delete
- **Flexible targets**: Comment on PRs, issues, or specific commits
- **Failure handling**: Optionally fail the workflow and append failure messages

Typical usage:

```yaml
- uses: hasura/comment-progress@v2.3.0
  with:
    github-token: ${{ secrets.GITHUB_TOKEN }}
    repository: ${{ github.repository }}
    number: ${{ github.event.number }}
    id: deploy-progress
    message: "Deploy in progress..."
    append: true
```

Best for: Long-running workflows where you want to provide incremental status updates as different stages complete. Good for deployment pipelines or multi-stage builds.

thollander/actions-comment-pull-request[6]

[5] https://github.com/hasura/comment-progress
[6] https://github.com/thollander/actions-comment-pull-request

A simpler, more straightforward action for posting or updating PR comments. Good balance of features and ease of use.

Key features:

- **Simple comment creation**: Easy to post one-time or updated comments
- **Comment updates**: Find and update existing comments by ID or content
- **Reactions**: Add emoji reactions to comments
- **Comment deletion**: Remove comments when no longer needed
- **Dynamic content**: Supports multi-line messages and environment variables

Typical usage:

```
- uses: thollander/actions-comment-pull-request@v2
  with:
    message: |
      ## Deployment Status
        Successfully deployed to preview environment
      Preview URL: https://preview-${{ github.event.number }}.example.com
```

Best for: Straightforward commenting needs without complex update logic. Good for simple status messages or one-time notifications.

7.6.0.3 Feature Comparison

Table 7.1: Comparison of PR comment action features

Feature	marocchino/sticky	hasura/comment-progress	thollander/actions-comment
Update existing comment	(by header)	(by identifier)	(by ID or content)
Multiple independent comments	(via headers)	(via identifiers)	(limited)
Append mode			
Delete comments			
Hide comments			
File-based messages			
Emoji reactions			

Feature	marocchino/sticky	hasura/comment-progress	thollander/actions-comment
Works with push events		(requires PR number)	(requires PR number)
Progress tracking focus	(flexible)		

7.6.0.4 Choosing the Right Action

Use marocchino/sticky-pull-request-comment when:

- You need to maintain multiple independent status comments (test results, coverage, deployment, etc.)
- You want to prevent comment spam by updating the same comment
- You need advanced features like hiding outdated comments or file-based templates
- Your workflow triggers on push events and needs to find the associated PR

Use hasura/comment-progress when:

- You have long-running workflows with multiple stages
- You want to provide progressive feedback as each stage completes
- You need the workflow to fail and report the failure in the comment
- You want a pattern similar to third-party CI/CD service bots

Use thollander/actions-comment-pull-request when:

- You need simple comment posting without complex update logic
- You want to add emoji reactions to comments
- You're comfortable with the action's simpler update mechanism
- Your use case doesn't require the advanced features of the other options

7.6.0.5 Security Considerations

> ⚠ Warning
>
> **Important: PR Comment Permissions**
> PR comment actions require write access to pull requests, which means they need the `pull-requests: write` permission.
> For workflows triggered by pull requests from forks (common in open-source projects), be careful about what information you expose in comments, as fork contributors can trigger these workflows. Never expose secrets or sensitive information in PR comments. See Section 18.5.8 for more guidance on workflow security.

7.7 Additional Resources

- GitHub Actions features overview[7]
- r-lib/actions repository[8] - R-specific actions and example workflows
- R Packages book: Continuous Integration[9]
- GitHub Actions documentation[10]
- Where to find help with r-lib/actions[11]

[7] https://github.com/features/actions
[8] https://github.com/r-lib/actions
[9] https://r-pkgs.org/software-development-practices.html#sec-sw-dev-practices-ci
[10] https://docs.github.com/en/actions
[11] https://github.com/r-lib/actions#where-to-find-help

8 R Code Style

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi[1]

Follow these code style guidelines for all R code:

8.1 General Principles

- **Follow tidyverse style guide**: https://style.tidyverse.org
- **Use native pipe**: `|>` not `%>%` (available in R >= 4.1.0)
- **Naming**: Use `snake_case` for functions and variables; acronyms may be uppercase (e.g., `prep_IDs_data`)
- **Write tidy code**: Keep code clean, readable, and well-organized
- **Avoid redundant logical comparisons**: Use logical variables directly in conditional statements (e.g., `if (x)` instead of `if (x == TRUE)` or `if (x == 1)`)
- **Use pipes to emphasize primary inputs**: When writing functions and code, use the pipe operator to clearly show transformations on a primary object. The primary input should flow as the first argument to each function in the chain. Design functions so the most important argument (usually data) comes first, enabling natural pipeline composition. See the tidyverse design principles[2] for more details.

8.2 Function Structure and Documentation

Every function should follow this pattern:

```
#' Short Title (One Line)
#'
#' Longer description providing details about what the function does,
#' when to use it, and important considerations.
#'
#' @param param1 Description of first parameter, including type and constraints
#' @param param2 Description of second parameter
#'
#' @returns Description of return value, including type and structure
#'
#' @examples
#' # Example usage
#' result <- my_function(param1 = "value", param2 = 10)
#'
```

[1] https://jadebc.github.io/lab-manual/coding-style.html
[2] https://design.tidyverse.org/important-args-first.html

```r
#' @export
my_function <- function(param1, param2) {
  # Implementation
}
```

See also Section 6.9 for general code documentation practices.

8.3 Comments

Use comments to explain *why*, not *what*:

```r
# Good: Explains reasoning
# Use log scale because distribution is highly skewed
ggplot(data, aes(x = log10(income))) + geom_histogram()

# Bad: States the obvious
# Create a histogram
ggplot(data, aes(x = income)) + geom_histogram()
```

File headers (for scripts in `data-raw/` or `inst/analyses/`):

```
################################################################################
# @Organization - Example Organization
# @Project - Example Project
# @Description - This file is responsible for [...]
################################################################################
```

File Structure - Just as your data "flows" through your project, data should flow naturally through a script. Very generally, you want to

1) source your config =>
2) load all your data =>
3) do all your analysis/computation => save your data.

Each of these sections should be "chunked together" using comments. See this file[3] for a good example of how to cleanly organize a file in a way that follows this "flow" and functionally separate pieces of code that are doing different things.

> **Note**
>
> If your computer isn't able to handle this workflow due to RAM or requirements, modifying the ordering of your code to accommodate it won't be ultimately helpful and your code will be fragile, not to mention less readable and messy. You need to look into high-performance computing (HPC) resources in this case.

Single-Line Comments - Commenting your code is an important part of reproducibility and helps document your code for the future. When things change or break, you'll be

[3] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/2a%20-%20Statistical-Inputs.R

thankful for comments. There's no need to comment excessively or unnecessarily, but a comment describing what a large or complex chunk of code does is always helpful. See this file[4] for an example of how to comment your code and notice that comments are always in the form of:

```
# This is a comment -- first letter is capitalized and spaced away from the pound sign
```

Multi-Line Comments - Occasionally, multi-line comments are necessary. You should manually insert line breaks to "hard-wrap" code and comments, whenever lines become longer than 80 characters. `lintr` should object otherwise, even for comments. Try to break lines at semantic boundaries: ends of sentences or phrases. Long lines in source code files make it more difficult to see and comment on diffs in pull requests.

In prose text chunks, Quarto ignores single line breaks, so you should also line-break your prose text in .qmd files to keep them under 80 characters.

You can configure RStudio's settings to display the 80-character margin.

8.4 Line Breaks and Formatting

Blank Lines Before Lists

Always include a blank line before starting a bullet list or numbered list in markdown/Quarto documents. This ensures proper rendering and readability.

Correct:

```
Here are the requirements:

- First item
- Second item
```

Incorrect:

```
Here are the requirements:
- First item
- Second item
```

Here's what happens if you don't add the blank line:

Here are the requirements: - First item - Second item

Line Breaks in Code

- For `ggplot` calls and `dplyr` pipelines, do not crowd single lines. Here are some nontrivial examples of "beautiful" pipelines, where beauty is defined by coherence:

[4] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/1b%20-%20Map-Management.R

```
# Example 1
school_names = list(
  OUSD_school_names = absentee_all |>
    filter(dist.n == 1) |>
    pull(school) |>
    unique |>
    sort,

  WCCSD_school_names = absentee_all |>
    filter(dist.n == 0) |>
    pull(school) |>
    unique |>
    sort
)
```

```
# Example 2
absentee_all = fread(file = raw_data_path) |>
  mutate(program = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                             schoolyr %in% program_schoolyrs ~ 1)) |>
  mutate(period = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                            schoolyr %in% LAIV_schoolyrs ~ 1,
                            schoolyr %in% IIV_schoolyrs ~ 2)) |>
  filter(schoolyr != "2017-18")
```

And of a complex `ggplot` call:

```
# Example 3
ggplot(data=data) +

  aes(x=.data[["year"]], y=.data[["rd"]], group=.data[[group]]) +

  geom_point(mapping = aes(col = .data[[group]], shape = .data[[group]]),
             position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping = aes(ymin=.data[["lb"]], ymax= .data[["ub"]], col= .data[[group]]),
                position=position_dodge(width=0.2),
                width=0.2) +

  geom_point(position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping=aes(ymin=lb, ymax=ub),
                position=position_dodge(width=0.2),
                width=0.1) +

  scale_y_continuous(limits=limits,
                     breaks=breaks,
                     labels=breaks) +
```

```
  scale_color_manual(std_legend_title,values=cols,labels=legend_label) +
  scale_shape_manual(std_legend_title,values=shapes, labels=legend_label) +
  geom_hline(yintercept=0, linetype="dashed") +
  xlab("Program year") +
  ylab(yaxis_lab) +
  theme_complete_bw() +
  theme(strip.text.x = element_text(size = 14),
        axis.text.x = element_text(size = 12)) +
  ggtitle(title)
```

Imagine (or perhaps mournfully recall) the mess that can occur when you don't strictly style a complicated `ggplot` call. Trying to fix bugs and ensure your code is working can be a nightmare. Now imagine trying to do it with the same code 6 months after you've written it. Invest the time now and reap the rewards as the code practically explains itself, line by line.

8.5 Markdown and Quarto Formatting

8.5.1 Writing about code in Quarto documents

When writing about code in prose sections of quarto documents, use backticks to apply a code style: for example, `dplyr::mutate()`. When talking about packages, use backticks and curly-braces with a hyperlink to the package website. For example: `{dplyr}`[5].

Important: Do not use raw HTML (`<a href="...">`) in `.qmd` files. Always use Quarto/markdown link syntax instead.

8.6 Messaging and User Communication

Use `cli` package functions for all user-facing messages in package functions:

```
# Good
cli::cli_inform("Analysis complete")
cli::cli_warn("Missing data detected")
cli::cli_abort("Invalid input: {x}")

# Bad - don't use these in package code
message("Analysis complete")
warning("Missing data detected")
stop("Invalid input")
```

[5] https://dplyr.tidyverse.org/

8.7 Package Code Practices

- **No `library()` in package code**: Use `::` notation or declare in DESCRIPTION Imports
- **Document all exports**: Use roxygen2 (`@title`, `@description`, `@param`, `@returns`, `@examples`)
- **Avoid code duplication**: Extract repeated logic into helper functions

8.8 Tidyverse Replacements

Use modern tidyverse/alternatives for base R functions:

```
# Data structures
tibble::tibble()           # instead of data.frame()
tibble::tribble()          # instead of manual data.frame creation

# I/O
readr::read_csv()          # instead of read.csv()
readr::write_csv()         # instead of write.csv()
readr::read_rds()          # instead of readRDS()
readr::write_rds()         # instead of saveRDS()

# Data manipulation
dplyr::bind_rows()         # instead of rbind()
dplyr::bind_cols()         # instead of cbind()

# String operations
stringr::str_which()       # instead of grep()
stringr::str_replace()     # instead of gsub()

# Date/time operations
lubridate::NA_Date_        # instead of as.Date(NA)

# Session info
sessioninfo::session_info() # instead of sessionInfo()
```

See also Section 6.15.

8.9 The here Package

The `here` package helps manage file paths in projects by automatically finding the project root and building paths relative to it:

```
library(here)

# Automatically finds project root and builds paths
data <- readr::read_csv(here("data-raw", "survey.csv"))
saveRDS(results, here("inst", "analyses", "results.rds"))
```

This solves the problem of different working directory paths across collaborators. For example, one person might have the project at `/home/oski/Some-R-Project` while another has it at `/home/bear/R-Code/Some-R-Project`. The `here` package handles this automatically.

This works regardless of where collaborators clone the repository. For more details, see the here package vignette[6].

See also Section 6.12 for detailed explanation of the here package.

8.10 Object Naming

Use descriptive names that are both expressive and explicit. Being verbose is useful and easy in the age of autocompletion:

```
# Good
vaccination_coverage_2017_18
absentee_flu_residuals

# Less good
vaxcov_1718
flu_res
```

Prefer nouns for objects and verbs for functions:

```
# Good
clean_data <- prep_study_data(raw_data)  # verb for function, noun for object

# Less clear
data <- process(input)
```

Generally we recommend using nouns for objects and verbs for functions. This is because functions are performing actions, while objects are not.

Use `snake_case` for all variable and function names. Avoid using `.` in names (as in base R's `read.csv()`), as this goes against best practices in modern R and other languages. Modern packages like `readr::read_csv()` follow this convention.

Try to make your variable names both more expressive and more explicit. Being a bit more verbose is useful and easy in the age of autocompletion! For example, instead of naming a variable `vaxcov_1718`, try naming it `vaccination_coverage_2017_18`. Similarly, `flu_res` could be named `absentee_flu_residuals`, making your code more readable and explicit.

Base R allows `.` in variable names and functions (such as `read.csv()`), but this goes against best practices for variable naming in many other coding languages. For consistency's

[6] https://github.com/jennybc/here_here

sake, `snake_case` has been adopted across languages, and modern packages and functions typically use it (i.e. `readr::read_csv()`). As a very general rule of thumb, if a package you're using doesn't use `snake_case`, there may be an updated version or more modern package that *does*, bringing with it the variety of performance improvements and bug fixes inherent in more mature and modern software.

Note

You may also see `camelCase` throughout the R code you come across. This is *okay* but not ideal – try to stay consistent across all your code with `snake_case`.

Note

Again, it's also worth noting there's nothing inherently wrong with using `.` in variable names, just that it goes against style best practices that are cropping up in data science, so it's worth getting rid of these bad habits now.

For more help, check out Be Expressive: How to Give Your Variables Better Names[7]

8.11 Automated Tools for Style and Project Workflow

8.11.1 Styling

8.11.1.1 RStudio shortcuts

1. **Code Autoformatting** - RStudio includes a fantastic built-in utility (keyboard shortcut: `CMD-Shift-A` (Mac) or `Ctrl-Shift-A` (Windows/Linux)) for autoformatting highlighted chunks of code to fit many of the best practices listed here. It generally makes code more readable and fixes a lot of the small things you may not feel like fixing yourself. Try it out as a "first pass" on some code of yours that *doesn't* follow many of these best practices!
2. **Assignment Aligner** - A cool R package[8] allows you to very powerfully format large chunks of assignment code to be much cleaner and much more readable. Follow the linked instructions and create a keyboard shortcut of your choosing (recommendation: `CMD-Shift-Z`). Here is an example of how assignment aligning can dramatically improve code readability:

[7] https://spin.atomicobject.com/2017/11/01/good-variable-names/
[8] https://www.r-bloggers.com/align-assign-rstudio-addin-to-align-assignment-operators/

```
# Before
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern
  "Carl Munck Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Munck
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pat
  "East Oakland PRIDE Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern
  "EnCompass Academy" = str_subset(string = OUSD_school_shapes$schnam, pattern = "EnCompass
  "Global Family School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Global
  "International Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern
  "Madison Park Lower Campus" = "Madison Park Academy TK-5",
  "Manzanita Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "
  "Martin Luther King Jr Elementary" = str_subset(string = OUSD_school_shapes$schnam, patte
  "PLACE @ Prescott" = "Preparatory Literary Academy of Cultural Excellence",
  "RISE Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Rise
)
```

```
# After
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pat
  "Carl Munck Elementary"              = str_subset(string = OUSD_school_shapes$schnam, pat
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pat
  "East Oakland PRIDE Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pat
  "EnCompass Academy"                  = str_subset(string = OUSD_school_shapes$schnam, pat
  "Global Family School"               = str_subset(string = OUSD_school_shapes$schnam, pat
  "International Community School"     = str_subset(string = OUSD_school_shapes$schnam, pat
  "Madison Park Lower Campus"          = "Madison Park Academy TK-5",
  "Manzanita Community School"         = str_subset(string = OUSD_school_shapes$schnam, pat
  "Martin Luther King Jr Elementary"   = str_subset(string = OUSD_school_shapes$schnam, pat
  "PLACE @ Prescott"                   = "Preparatory Literary Academy of Cultural Excellen
  "RISE Community School"              = str_subset(string = OUSD_school_shapes$schnam, pat
)
```

8.11.1.2 {styler}[9]

{styler}[10] is another cool R package from the Tidyverse[11] that can be powerful and used as a first pass on entire projects that need refactoring. The most useful function of the package is the `style_dir` function, which will style all files within a given directory. See the function's documentation[12] and the vignette linked above for more details.

> **Note**
>
> The default Tidyverse styler is subtly different from some of the things we've advocated for in this document. Most notably we differ with regards to the assignment operator (<- vs =) and number of spaces before/after "tokens" (i.e. Assignment Aligner add spaces before = signs to align them properly). For this reason, we'd recom-

[9] https://styler.r-lib.org/
[10] https://styler.r-lib.org/
[11] https://tidyverse.org/blog/2017/12/styler-1.0.0/
[12] https://www.rdocumentation.org/packages/styler/versions/1.1.0/topics/style_dir

mend the following: `style_dir(path = ..., scope = "line_breaks", strict = FALSE)`. You can also customize `{styler}`[a] even more[b] if you're really hardcore.

[a] https://styler.r-lib.org/
[b] http://styler.r-lib.org/articles/customizing_styler.html

Note

As is mentioned in the package vignette linked above, `{styler}`[a] modifies things *in-place*, meaning it overwrites your existing code and replaces it with the updated, properly styled code. This makes it a good fit on projects *with version control*, but if you don't have backups or a good way to revert back to the initial code, I wouldn't recommend going this route.

[a] https://styler.r-lib.org/

styler Package

For automated styling of entire projects:

```
# Install styler
install.packages("styler")

# Style all files in R/ directory
styler::style_dir("R/")

# Style entire package
styler::style_pkg()

# Note: styler modifies files in-place
# Always use with version control so you can review changes
```

8.11.1.3 `{lintr}`[13]

Linters are programming tools that check adherence to a given style, syntax errors, and possible semantic issues. The R linter, called `lintr`, can be found in this package[14]. It helps keep files consistent across different authors and even different organizations. For example, it notifies you if you have unused variables, global variables with no visible binding, not enough or superfluous whitespace, and improper use of parentheses or brackets. A list of its other purposes can be found in this link[15], and most guidelines are based on the Tidyverse R Style Guide[16].

[13] https://lintr.r-lib.org/
[14] https://www.rdocumentation.org/packages/lintr/versions/1.0.3
[15] https://cran.r-project.org/web/packages/lintr/readme/README.html#available-linters
[16] https://style.tidyverse.org/

Note

You can customize your settings to set defaults or to exclude files. More details can be found here[a].

[a] https://cran.r-project.org/web/packages/lintr/readme/README.html#project-configuration

Note

The `lintr` package goes hand in hand with the `styler` package. The styler can be used to automatically fix the problems that the lintr catches.

`lintr` package

For checking code style without modifying files:

```
# Install lintr
install.packages("lintr")

# Lint the entire package
lintr::lint_package()

# Lint a specific file
lintr::lint("R/my_function.R")
```

The linter checks for:

- Unused variables
- Improper whitespace
- Line length issues
- Style guide violations

You can customize linting rules by creating a `.lintr` or `lintr.R` file in your project root.

8.12 Additional Resources

- Tidyverse style guide (Wickham 2023a): Detailed coding style conventions for writing clear, consistent R code. Covers naming, syntax, pipes, functions, and more.

9 Big data

Adapted by UCD-SeRG team from original by Kunal Mishra and Jade Benjamin-Chung[1]

9.1 The data.table package

It may also be the case that you're working with very large datasets. Generally I would define this as 10+ million rows. As is outlined in this document, the 3 main players in the data analysis space are Base R, `Tidvyerse` (more specificially, `dplyr`), and `data.table`. For a majority of things, Base R is inferior to both `dplyr` and `data.table`, with concise but less clear syntax and less speed. `Dplyr` is architected for medium and smaller data, and while its very fast for everyday usage, it trades off maximum performance for ease of use and syntax compared to `data.table`. An overview of the `dplyr` vs `data.table` debate can be found in this stackoverflow post[2] and all 3 answers are worth a read.

You can also achieve a performance boost by running `dplyr` commands on `data.table`s, which I find to be the best of both worlds, given that a `data.table` is a special type of `data.frame` and fairly easy to convert with the `as.data.table()` function. The speedup is due to `dplyr`'s use of the `data.table` backend and in the future this coupling should become even more natural.

If you want to test whether using a certain coding approach increases speed, consider the `tictoc` package. Run `tic()` before a code chunk and `toc()` after to measure the amount of system time it takes to run the chunk. For example, you might use this to decide if you *really* need to switch a code chunk from `dplyr` to `data.table`.

9.2 Using downsampled data

In our studies with very large datasets, we save "downsampled" data that usually includes a 1% random sample stratified by any important variables, such as year or household id. This allows us to efficiently write and test our code without having to load in large, slow datasets that can cause RStudio to freeze. Be very careful to be sure which dataset you are working with and to label results output accordingly.

9.3 Optimal RStudio set up

Using the following settings will help ensure a smooth experience when working with big data. In RStudio, go to the "Tools" menu, then select "Global Options". Under "General":

Workspace

[1] https://jadebc.github.io/lab-manual/working-with-big-data.html

[2] https://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly/27840349#27840349

- **Uncheck** Restore RData into workspace at startup
- Save workspace to RData on exit – choose **never**

History

- **Uncheck** Always save history

Unfortunately RStudio often gets slow and/or freezes after hours working with big datasets. Sometimes it is much more efficient to just use Terminal / gitbash to run code and make updates in git.

10 Data masking

Adapted by UCD-SeRG team from original by Anna Nguyen, Jade Benjamin-Chung, and Gabby Barratt Heitmann[1]

For information about UC Davis computing resources for data-intensive work, see Chapter 17.

10.1 General Overview

This chapter covers data masking, a unique process in R in which columns are treated as distinct objects within their dataframe's environment. In our lab, data masking most frequently comes up when writing wrapper functions where arguments to indicate column names are supplied as strings. We often do this when we repeat the same code on multiple columns, and want to apply a function to a vector of strings that correspond to column names in a dataframe. For example, we might want to clean multiple columns using the same function or estimate the same model under different feature sets. Here, we try to break down what data masking is, why this error comes up, and common approaches to solve this problem.

10.1.1 What is Data Masking?

Within certain tidyverse operations, columns are called as if they were variables. For example, while running `df |> mutate(X = …)` R recognizes that `X` specifically references a column in `df` without explicitly stating its membership `df |> mutate(df$X = …)` or calling the column name as a string `df |> mutate("X" = …)`.

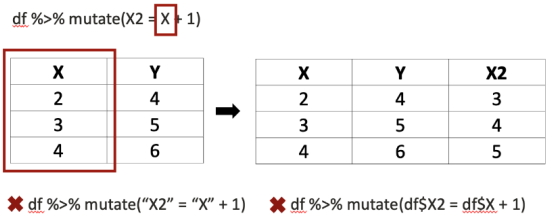

Figure 10.1: Data masking in tidyverse operations

However, this behavior may introduce errors when we attempt to incorporate variables from the global environment within these tidyverse pipelines. In the example shown in Figure 10.1, `column_name = "X"` followed by `df |> mutate(X2 = column_name + 1)` would yield an error, since `column_name` is not a column in `df` and the variable `column_name` is not defined within the environment of `df`

[1] https://jadebc.github.io/lab-manual/data-masking.html

10.1.2 Using tidy evaluation for data masking

In dplyr-based R programming, we make use of tidy evaluation. This allows us to avoid using base R syntax to reference specific columns in a data frame. By leveraging Tidy evaluation-based data masking, we can employ long pipes with several dplyr verbs to manipulate our data using stand-alone variables that store column names as strings.

For example, consider a data frame "df" that contains a column called "heavyrain" that we want to manipulate. Suppose we wanted to convert the values of "heavyrain" into a factor.

Using base R, which does not mask data, heavyrain must have quotes to be treated as a data-variable:

```
df[["outcome"]] = as.factor(df[["heavyrain"]])
```

In a dplyr pipe, heavyrain is being masked using tidy evaluation and will be correctly interpreted as a column because it is recognized as a data-variable: df |> mutate(outcome = as.factor(heavyrain))

With modified data masking, heavyrain is a string that is coerced into being recognized as a data-variable:

```
var_name = "heavyrain"
df |> mutate(outcome = as.factor(!!sym(var_name))
```

While cleaner and often more convenient, the data frame that var_name is in is now "masked" and we refer to the vectors in the dataframe (data-variables) as though it is an object of its own (an environmental-variable). This is why we can just say the variable's name in the context of a pipe – we treat it as though it's an object defined in our environment. Within normal scripts, this is usually fine, because the data frame is "held on to" in the pipe. However, it can cause some programming hurdles when writing functions that take strings of variable/column names as arguments. In the next section, we briefly describe how to troubleshoot common errors in data masking, as relevant to our lab's work.

10.2 Technical Overview

This section covers the R functions and tools that we often use in the context of data masking, focusing on the bang bang operator (`!!`) with symbol coercion (`sym()`) and the Walrus operator (`:=`).

The combined use of `!!` and `sym()` allows us to use strings, rather than data-variables, to reference column names within dplyr. Together, `!!sym("column_name")` forces dplyr to recognize "column_name" as a data-variable prior to evaluating the rest of the expression, enabling the ability to perform calculations on the column while referring to it as a string. `sym()` is a function that turns strings into symbols. In the context of a dplyr pipe, these symbols are interpreted as data-variables. The `!!` (bang bang) operator tells dplyr to evaluate the sym() expression first, e.g. to unquote its expression (e.g. "column_name") and evaluate it as a pre-existing object, first. This is helpful because often we use `sym("column_name")` within a larger expression, and dplyr might evaluate other elements of the expression first without !!, causing errors.

When we want to create a new column (via mutate or summarize), the Walrus operator (`:=`) allows us to specify the new column's name using a string. For example, while `df |> mutate("new_column" = values)` would yield an error, `df |> mutate("new_column" := values)` will correctly create a new column called "new_column". If we want to use a variable representing a string, we can use !! to force the variable to be evaluated before using `:=` to assign the value of the new column.

```
col_name = "new_column"
df |> mutate(!!col_name := values)
```

10.2.1 Example

Suppose we want to write a function "generate_descriptive_table" to summarize how the prevalence of "outcome" varies under different levels of a "risk_factor" in a data frame "df"

We can start by writing the function shell:

```
generate_descriptive_table <- function (df, outcome, rf) {
outcome_dist_by_rf <- ….
return(outcome_dist_by_rf)
}
```

Next, we can filter the data frame for only rows in which "rf" and "outcome" are not missing. We can use !! and sym() within filter to evaluate the strings stored in "rf" and "outcome". Note that defining `!!sym(outcome)` or `!!sym(outcome)` in variables *outside of the dplyr pipeline* will *not* work.

```
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  ….
  return(outcome_dist_by_rf)
}
```

Similarly, we use !! and sym() in group_by to evaluate column name, stored as a string in the argument "rf"

```
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  ….
  return(outcome_dist_by_rf)
}
```

Finally, we can use the walrus operator, !! and sym() with "summarize" to create a new column that takes the mean of the column referenced in "rf". We also use "glue" or "paste" to give the new column an informative name that includes the "outcome" it describes.

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(glue::glue("{outcome}_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(paste0(outcome, "_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  new_column_name = paste0(outcome, "_prev")
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(new_column_name) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

11 Quarto

11.1 Introduction

Quarto[1] is an open-source scientific and technical publishing system that allows you to create documents, books, websites, presentations, and more. Quarto provides a unified authoring framework for data science, combining your code, its results, and your prose. Quarto documents are fully reproducible and support dozens of output formats, like PDFs, Word files, presentations, and more.

Quarto files are designed to be used in three ways:

1. **For communicating to decision-makers,** who want to focus on the conclusions, not the code behind the analysis.
2. **For collaborating with other data scientists** (including future you!), who are interested in both your conclusions, and how you reached them (i.e., the code).
3. **As an environment in which to do data science,** as a modern-day lab notebook where you can capture not only what you did, but also what you were thinking.

11.1.1 Key Features

Multi-format Output: Quarto documents can be rendered into HTML, PDF, MS Word, ePub, PowerPoint, Revealjs presentations, dashboards, websites, and books from a single source file. This allows authors to maintain one document but publish it in multiple formats without rewriting content.

Rich Markdown Authoring: Content is created in markdown, with support for figures, tables, equations (LaTeX), citations, cross-references, and advanced layout features like tabs, callouts, and panels.

Embedded Executable Code: Integrate code chunks (R, Python, Julia, Observable JS) that can be executed and the results rendered directly in the document. This allows for dynamic results, data analysis, plots, and reproducible research workflows.

Interactivity: Add interactive components such as widgets, tab sets, and collapsible sections for richer communication with readers.

Customization: Extensive theming and styling options, including custom CSS and advanced layout controls for polished, publication-quality output.

Project Management: Organize large projects and integrate with version control tools like Git. Use Quarto projects to group related documents, manage dependencies, and orchestrate rendering.

[1] https://quarto.org/

11.1.2 Why Quarto?

If you're an R Markdown user, you might be thinking "Quarto sounds a lot like R Markdown." You're not wrong! Quarto unifies the functionality of many packages from the R Markdown ecosystem (rmarkdown, bookdown, distill, xaringan, etc.) into a single consistent system as well as extends it with native support for multiple programming languages like Python and Julia in addition to R. In a way, Quarto reflects everything that was learned from expanding and supporting the R Markdown ecosystem over a decade.

11.1.3 Getting Started

To get started with Quarto:

- **Installation:** Quarto CLI is included with RStudio, so if you have a recent version of RStudio, you already have Quarto. Otherwise, visit https://quarto.org/docs/get-started/
- **Documentation:** The official Quarto documentation is available at https://quarto.org/docs/guide/
- **R4DS Chapter:** For an excellent introduction to using Quarto with R, see the Quarto chapter in R for Data Science[2] (Wickham, Çetinkaya-Rundel, and Grolemund 2023)

11.2 Quarto Basics

A Quarto document is a plain text file with the extension `.qmd`. It contains three important types of content:

1. An (optional) **YAML header** surrounded by `---`s
2. **Chunks** of code surrounded by ```` ``` ````
3. Text mixed with simple text formatting like `# heading` and `_italics_`

11.2.1 Creating a New Quarto Document

In RStudio, create a new Quarto document using **File > New File > Quarto Document...** in the menu bar. RStudio will launch a wizard that you can use to pre-populate your file with useful content that reminds you how the key features of Quarto work.

11.2.2 Visual vs. Source Editor

RStudio provides two ways to edit Quarto documents:

Visual Editor: The Visual editor provides a WYSIWYM[3] interface for authoring Quarto documents. If you're new to computational documents but have experience using tools like Google Docs or MS Word, the visual editor is the easiest way to get started. In the visual

[2] https://r4ds.hadley.nz/quarto.html
[3] https://en.wikipedia.org/wiki/WYSIWYM

editor you can use the buttons on the menu bar to insert images, tables, cross-references, etc., or you can use the catch-all + / or Ctrl + / shortcut to insert just about anything.

Source Editor: The source editor allows you to edit the raw markdown and code. While the visual editor will feel familiar to those with experience in word processors, the source editor will feel familiar to those with experience writing R scripts or R Markdown documents. The source editor can also be useful for debugging any Quarto syntax errors since it's often easier to catch these in plain text.

You can switch between the visual and source editors at any time using the toggle in the top-left of the editor pane.

11.2.3 Rendering Documents

To produce a complete report containing all text, code, and results:

- Click **"Render"** in RStudio, or
- Press **Cmd/Ctrl + Shift + K**, or
- Use `quarto::quarto_render("document.qmd")` in R, or
- Use `quarto render document.qmd` in the terminal

When you render the document, Quarto sends the `.qmd` file to **knitr**, which executes all of the code chunks and creates a new markdown (`.md`) document which includes the code and its output. The markdown file generated by knitr is then processed by **pandoc**, which is responsible for creating the finished file in your chosen format (HTML, PDF, Word, etc.).

11.2.4 Code Chunks

To run code inside a Quarto document, you need to insert a chunk. There are three ways to do so:

1. The keyboard shortcut **Cmd + Option + I** / **Ctrl + Alt + I**
2. The **"Insert"** button icon in the editor toolbar
3. By manually typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````

Chunks can be given an optional label and various chunk options:

````
```{r}
#| label: simple-addition
#| echo: false
1 + 1
```
````

Common chunk options include:

- `#| label:` - give the chunk a name
- `#| echo: false` - hide the code but show the output
- `#| code-fold: true` - allow readers to toggle code visibility (useful when output is more important to the narrative than the code)
- `#| eval: false` - show the code but don't run it
- `#| include: false` - run the code but hide both code and output
- `#| warning: false` - hide warnings

- `#| message: false` - hide messages

Use `code-fold: true` for chunks where the output is important to the narrative and not the code used to produce it. This allows interested readers to expand and view the code while keeping the document focused on results.

11.2.5 Format-Specific Settings

When rendering to multiple output formats (HTML, PDF, DOCX, EPUB), you may want different chunk options or behavior for different formats. Use `knitr::pandoc_to()` with `if ()` statements to detect the output format and set format-specific settings.

Example: Different figure sizes for different formats

````
```{r}
#| label: example-plot
#| fig-width: !expr if (knitr::pandoc_to("html")) 8 else 6
#| fig-height: !expr if (knitr::pandoc_to("html")) 6 else 4

plot(1:10)
```
````

Example: Conditional code execution based on format

````
```{r}
if (knitr::pandoc_to("docx")) {
  # DOCX-specific code
  knitr::kable(data, format = "simple")
} else if (knitr::pandoc_to("html")) {
  # HTML-specific code
  knitr::kable(data, format = "html")
} else {
  # PDF or other formats
  knitr::kable(data, format = "latex")
}
```
````

Common format detection patterns:

- `knitr::pandoc_to("html")` - returns `TRUE` for HTML output
- `knitr::pandoc_to("latex")` - returns `TRUE` for PDF output
- `knitr::pandoc_to("docx")` - returns `TRUE` for Word output
- `knitr::pandoc_to("epub")` - returns `TRUE` for EPUB output

This technique is particularly useful when you need to:

- Adjust figure dimensions for different page sizes
- Use different table formatting for different outputs
- Include or exclude content based on output format
- Set format-specific styling or options

11.2.6 Text Formatting

Quarto uses Pandoc's markdown for text formatting:

- `*italic*` or `_italic_` produces *italic* text
- `**bold**` or `__bold__` produces **bold** text
- `` `code` `` produces `code` formatting
- `# Heading 1`, `## Heading 2`, `### Heading 3` for headings
- Bullet lists start with `-` or `*`
- Numbered lists start with `1.`, `2.`, etc.
- `[link text](url)` creates hyperlinks
- `![alt text](image.png)` inserts images

Important: Always include a blank line before bullet lists and numbered lists in markdown and Quarto documents.

For more details on using Quarto for writing and analysis, see the Quarto chapter in R for Data Science[4] (Wickham, Çetinkaya-Rundel, and Grolemund 2023).

11.3 Building Quarto Books

Quarto books let you author entire books (or course notes, manuals, dissertations, etc.) in markdown. Quarto books are ideal for documentation, tutorials, lab manuals, and other long-form content.

11.3.1 Creating a Quarto Book

Starting from a template (recommended):

Using a template is the fastest way to get started with a Quarto book, as it provides pre-configured settings, example content, and GitHub Actions workflows for automated deployment:

1. **UCD-SeRG Quarto Book Template** - Our recommended template with pre-configured settings for lab publications:
   - Repository: https://github.com/UCD-SERG/qbt
   - Click "Use this template" → "Create a new repository" on GitHub
   - Clone your new repository and start editing
   - Includes GitHub Actions for automatic deployment to GitHub Pages
2. **Coatless Tutorials Quarto Book Template** - Another template with helpful examples:
   - Repository: https://github.com/coatless-tutorials/quarto-book-template
   - Includes examples of common Quarto book features
3. **DataLab Quarto Template** - Template from the UC Davis DataLab and Davis R Users Group:
   - Repository: https://github.com/d-rug/datalab_template_quarto
   - Provides a starting point for DataLab workshop materials and tutorials

[4] https://r4ds.hadley.nz/quarto.html

While these templates jumpstart your project with up-to-date configuration and workflow files, you should still come up to speed on what all the config files do (particularly `_quarto.yml` and any GitHub Actions workflows) so you can modify and debug them as needed. The templates serve as central locations for the most current versions of these files and best practices.

Starting from scratch:

If you prefer to start from scratch, you can create a new Quarto book project using the Quarto CLI:

```
# Create a new Quarto book project
quarto create project book mybook
cd mybook
```

This will create a basic book structure with:

- `_quarto.yml` - configuration file for your book
- `index.qmd` - the home page / preface
- Sample chapter files
- `references.qmd` - bibliography/references page

11.3.2 Building and Previewing

Once you have a Quarto book project, you can build and preview it:

```
# Render the entire book
quarto render

# Preview with live reload (recommended during development)
quarto preview
```

The `quarto preview` command starts a local web server and automatically refreshes the preview whenever you save changes to your files.

11.3.3 Book Structure

A typical Quarto book is organized as follows:

`_quarto.yml`: The main configuration file that defines:

- Book metadata (title, author, date)
- Chapter order
- Output formats (HTML, PDF, ePub, etc.)
- Styling and theme
- Navigation options

Chapter files: Individual `.qmd` files for each chapter. These are listed in the `chapters` section of `_quarto.yml`.

Parts: You can organize chapters into parts for better structure:

```yaml
book:
  chapters:
    - index.qmd
    - part: "Getting Started"
      chapters:
        - intro.qmd
        - basics.qmd
    - part: "Advanced Topics"
      chapters:
        - advanced.qmd
```

11.3.4 Book Features

Quarto books support many advanced features:

Cross-references: Reference figures, tables, equations, and sections throughout your book:

```
See @fig-plot for details.
As shown in @tbl-results.
Refer to @sec-introduction.
```

Citations: Include a bibliography and cite sources:

```
According to @smith2020, the method works well.
```

Search: Automatic full-text search in HTML output.

Downloads: Offer PDF, ePub, and Word versions alongside HTML.

Navigation: Automatic table of contents, previous/next chapter buttons, and breadcrumbs.

Customization: Custom themes, CSS, and templates for professional appearance.

11.3.5 Example: This Lab Manual

This lab manual itself is a Quarto book! You can view its source code at https://github.com/UCD-SERG/lab-manual to see how we've structured chapters, used includes for modular content, and configured various output formats.

11.3.6 Resources

- Quarto Books Guide[5] - comprehensive documentation for Quarto books
- Quarto Publishing Guide[6] - how to publish your book online
- R4DS Quarto chapter[7] (Wickham, Çetinkaya-Rundel, and Grolemund 2023) - excellent introduction to Quarto

[5] https://quarto.org/docs/books/
[6] https://quarto.org/docs/publishing/
[7] https://r4ds.hadley.nz/quarto.html

11.4 Quarto Profiles

Quarto profiles allow you to customize rendering behavior for different purposes. A profile is a named set of configuration options that can be activated when rendering. This is particularly useful when you want to render the same source files in different formats or for different audiences.

11.4.1 What are Profiles?

Profiles let you maintain multiple `_quarto.yml` configuration files in the same project. For example, you might have:

- `_quarto.yml` - default book configuration
- `_quarto-revealjs.yml` - configuration for rendering chapters as slide decks
- `_quarto-print.yml` - configuration optimized for PDF printing

11.4.2 Example: Rendering Chapters as Slides

A excellent example of using Quarto profiles comes from the Regression Models for Epidemiology[8] course materials by D. Morrison.

The project includes a `_quarto-revealjs.yml` profile that allows each chapter to be compiled as a RevealJS slide deck, in addition to being part of the book.

To render a single chapter as slides:

```
quarto render chapter-name.qmd --profile=revealjs
```

To render all chapters listed in the profile as slides:

```
quarto render --profile=revealjs
```

11.4.3 Creating a Profile

To create a profile:

1. Create a new YAML file named `_quarto-{profile-name}.yml`
2. Include only the configuration options that differ from your default `_quarto.yml`
3. Activate the profile when rendering using `--profile={profile-name}`

Example profile structure:

`_quarto-slides.yml`:

[8] https://d-morrison.github.io/rme/

```yaml
project:
  type: default
  output-dir: slides

format:
  revealjs:
    theme: serif
    slide-number: true
    preview-links: auto
```

11.4.4 Common Use Cases

Multiple output formats: Maintain separate configurations for web, print, and presentation versions of your content.

Different audiences: Create versions with or without solutions, technical details, or instructor notes.

Development vs. production: Use a development profile with faster rendering options during writing, and a production profile with full features for final output.

Course materials: Render the same content as both a reference book and lecture slides, as demonstrated in the RME course[9].

11.4.5 Resources

- Quarto Profiles Documentation[10]
- RME Example[11] - see the source at https://github.com/d-morrison/rme for a working example

11.5 Advanced Features

11.5.1 Cross-References

Quarto provides a powerful cross-reference system for figures, tables, equations, and sections. Cross-references automatically number your content and create clickable links in HTML and PDF output.

Required label prefixes:

- Figures: `#fig-` (e.g., `#fig-workflow-diagram`)
- Tables: `#tbl-` (e.g., `#tbl-summary-stats`)
- Equations: `#eq-` (e.g., `#eq-regression-model`)
- Sections: `#sec-` (e.g., `#sec-introduction`)
- Theorems: `#thm-`, Lemmas: `#lem-`, Corollaries: `#cor-`
- Propositions: `#prp-`, Examples: `#exm-`, Exercises: `#exr-`

[9] https://d-morrison.github.io/rme/
[10] https://quarto.org/docs/projects/profiles.html
[11] https://d-morrison.github.io/rme/

For figures (static images):

```
![Caption text](path/to/image.png){#fig-label}
```

For code-generated figures:

````
```{r}
#| label: fig-plot-name
#| fig-cap: "Caption text describing the plot"

# R code to generate plot
ggplot(data, aes(x, y)) + geom_point()
```
````

For tables (markdown tables):

```
| Column 1 | Column 2 |
|----------|----------|
| Data     | Data     |

: Caption text {#tbl-label}
```

For code-generated tables:

````
```{r}
#| label: tbl-table-name
#| tbl-cap: "Caption text"

# R code to generate table
knitr::kable(data)
```
````

Referencing in text:

- Figures: `@fig-label` produces "Figure X"
- Tables: `@tbl-label` produces "Table X"
- Equations: `@eq-label` produces "Equation X"
- Sections: `@sec-label` produces "Section X"

Benefits:

- Automatic numbering of figures, tables, and equations
- Automatic updates when content is reordered
- Clickable cross-references in HTML and PDF output
- Consistent formatting across all output formats
- Better accessibility for screen readers

For complete details, see the Quarto Cross-References documentation[12].

[12] https://quarto.org/docs/authoring/cross-references.html

11.5.2 Using Includes for Modular Content

Quarto's include feature allows you to decompose large documents into smaller, more manageable files. This is particularly useful for books and long documents.

Basic syntax:

```
{{< include path/to/file.qmd >}}
```

Benefits:

1. **Better Git History:** When sections are reordered, only the main chapter file changes (moving include statements), making it immediately clear that content was reorganized rather than edited.
2. **Easier Code Review:** Reviewers can see exactly what changed—either the organization (main file) or the content (include file).
3. **Modular Maintenance:** Each section lives in its own file, making it easier to find and edit specific content, reuse sections across chapters, and work on different sections simultaneously without merge conflicts.
4. **Clear Structure:** The main chapter file becomes a table of contents showing the organization at a glance.

Recommended pattern:

Main chapter file (e.g., `05-coding-practices.qmd`):

```
# Coding Practices

## Section Heading

{{< include coding-practices/section-name.qmd >}}

## Another Section

{{< include coding-practices/another-section.qmd >}}
```

Include files (e.g., `coding-practices/section-name.qmd`):

- Stored in a subdirectory matching the chapter name
- Contains only the content for that section (no heading)
- The heading stays in the main chapter file
- Named descriptively using kebab-case

Note: The heading must be in the main file, followed by a blank line, then the include statement. This keeps the document structure clear in the main file.

For more details, see the Quarto Includes documentation[13].

[13] https://quarto.org/docs/authoring/includes.html

11.6 Additional Resources

11.6.1 Official Documentation

- Quarto Official Guide[14] - comprehensive official documentation
- Quarto Books Guide[15] - documentation specific to creating books
- Quarto Publishing Guide[16] - how to publish your Quarto content online
- Quarto Getting Started[17] - installation and basic usage

11.6.2 Learning Resources

- R for Data Science - Quarto Chapter[18] (Wickham, Çetinkaya-Rundel, and Grolemund 2023) - excellent introduction to using Quarto with R
- Regression Models for Epidemiology[19] - example of a Quarto book with profiles for rendering chapters as slides
- UCD-SeRG Lab Manual Source[20] - this manual's source code provides examples of:
  - Book structure and organization
  - Using includes for modular content
  - Configuring multiple output formats (HTML, PDF, ePub, Word)
  - Cross-references for figures, tables, and sections

11.6.3 Templates

- UCD-SeRG Quarto Book Template[21] - our recommended template
- Coatless Tutorials Quarto Book Template[22] - another frequently-used template with helpful examples
- DataLab Quarto Template[23] - template from the UC Davis DataLab and Davis R Users Group

11.6.4 Related Lab Manual Chapters

For additional context about using Quarto in our lab:

- Chapter 6 - R coding practices that apply to code in Quarto documents
- Chapter 8 - Code style guidelines including formatting for Quarto documents
- Chapter 12 - Version control for Quarto projects

[14] https://quarto.org/docs/guide/
[15] https://quarto.org/docs/books/
[16] https://quarto.org/docs/publishing/
[17] https://quarto.org/docs/get-started/
[18] https://r4ds.hadley.nz/quarto.html
[19] https://d-morrison.github.io/rme/
[20] https://github.com/UCD-SERG/lab-manual
[21] https://github.com/UCD-SERG/qbt
[22] https://github.com/coatless-tutorials/quarto-book-template
[23] https://github.com/d-rug/datalab_template_quarto

12 Github

Adapted by UCD-SeRG team from original by Stephanie Djajadi and Nolan Pokpongkiat[1]

12.1 Basics

- A detailed tutorial of Git can be found here on the CS61B website[2].
- If you are already familiar with Git, you can reference the summary at the end of Section B[3].
- If you have made a mistake in Git, you can refer to On undoing, fixing, or removing commits in git (Robertson, n.d.) to undo, fix, or remove commits in git.
- For hands-on Git practice, see the UC Davis DataLab Git Sandbox[4] - a collaborative repository for learning Git workflows.

12.2 GitHub Education and Copilot Access

12.2.1 GitHub Education Benefits

Students, faculty, and researchers at UC Davis can access GitHub Education benefits, which include free access to GitHub Pro features and various developer tools.

To sign up for GitHub Education:

1. Visit the GitHub Education website[5]
2. Click "Get benefits" or "Join GitHub Education"
3. Sign in with your GitHub account (or create one if you don't have one)
4. Complete the application form using your UC Davis email address (ending in `ucdavis.edu`)
5. Provide proof of your academic affiliation (e.g., upload a photo of your student ID or university letter)
6. Wait for approval (typically takes a few days)

Once approved, you'll have access to the GitHub Student Developer Pack[6] (for students) or GitHub Teacher Toolbox[7] (for faculty), which includes numerous free tools and services for learning and development.

[1] https://jadebc.github.io/lab-manual/github.html
[2] https://sp19.datastructur.es/materials/guides/using-git#b-local-repositories-narrative-introduction
[3] https://sp19.datastructur.es/materials/guides/using-git#b-local-repositories-narrative-introduction
[4] https://github.com/ucdavisdatalab/sandbox_git
[5] https://education.github.com/
[6] https://education.github.com/pack
[7] https://education.github.com/teachers

12.2.2 GitHub Copilot Access for UC Davis Members

GitHub Copilot is an AI-powered coding assistant that can significantly accelerate your work. As a member of the UC Davis GitHub organization, you may be eligible for a free Copilot seat.

To request a Copilot seat:

If you are not yet a member of the UC Davis GitHub organization:

1. Ensure you have a GitHub account and have signed up for GitHub Education (see above)
2. Go to the UC Davis IT ServiceHub GitHub page[8]
3. Click the blue "Get GitHub!" button near the top right
4. When you receive the invitation email, make sure to check the "Ask for a GitHub Copilot seat (optional)" checkbox before joining (Figure 12.1)

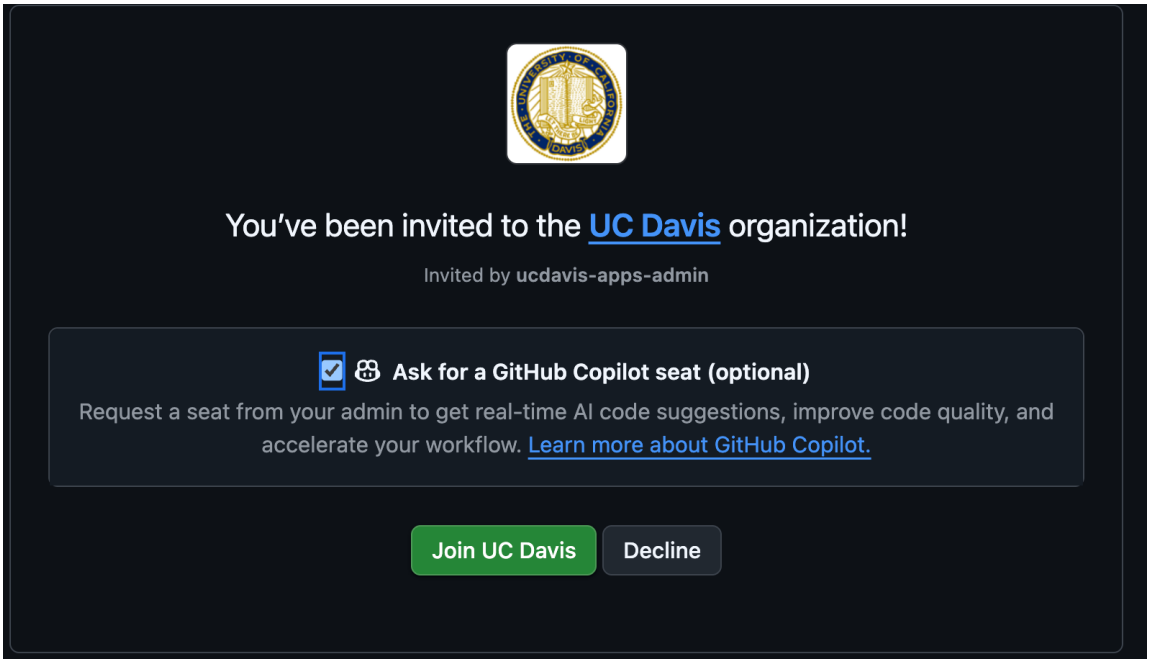

Figure 12.1: UC Davis GitHub invitation with Copilot seat option

5. Click "Join UC Davis" to accept the invitation
6. Follow the setup instructions to install and configure Copilot in your development environment

If you are already a member of the UC Davis GitHub organization:

1. Go to your GitHub Copilot settings[9]
2. Scroll to the "Get Copilot from an organization" section at the bottom
3. Find the "ucdavis" entry and click the request button

[8] https://servicehub.ucdavis.edu/servicehub?id=it_catalog_content&sys_id=951add951b5798103f4286ae6e4bcb12

[9] https://github.com/settings/copilot/features

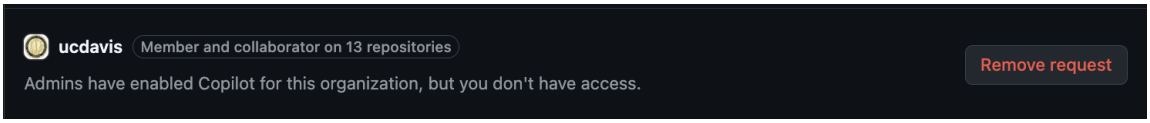

Figure 12.2: GitHub Copilot settings page showing organization options

4. Wait for approval from the UC Davis GitHub organization administrators
5. Once approved, follow the setup instructions to install and configure Copilot in your development environment

For guidance on using GitHub Copilot effectively, see Section 18.5.8 in the Working with AI chapter.

12.3 Github Desktop

While knowing how to use Git on the command line will always be useful since the full power of Git and its customizations and flexibility is designed for use with the command line, Github also provides GitHub Desktop ("GitHub Desktop," n.d.) as a graphical interface to do basic git commands; you can do all of the basic functions of Git using this desktop app. Feel free to use this as an alternative to Git on the command line if you prefer.

12.4 Git Branching

Branches allow you to keep track of multiple versions of your work simultaneously, and you can easily switch between versions and merge branches together once you've finished working on a section and want it to join the rest of your code. Here are some cases when it may be a good idea to branch:

- You may want to make a dramatic change to your existing code (called refactoring) but it will break other parts of your project. But you want to be able to simultaneously work on other parts or you are collaborating with others, and you don't want to break the code for them.
- You want to start working on a new part of the project, but you aren't sure yet if your changes will work and make it to the final product.
- You are working with others and don't want to mix up your current work with theirs, even if you want to bring your work together later in the future.

A detailed tutorial on Git Branching can be found here[10]. You can also find instructions on how to handle merge conflicts when joining branches together.

12.5 Example Workflow

A standard workflow when starting on a new project and contributing code looks like this:

[10] https://sp19.datastructur.es/materials/guides/using-git#e-git-branching-advanced-git-optional

Table 12.1: Standard Git workflow for new projects

Command	Description
SETUP: FIRST TIME ONLY: `git clone <url> <directory_name>`	Clone the repo. This copies of all the project files in its current state on Github to your local computer.
1. `git pull origin master`	update the state of your files to match the most current version on GitHub
2. `git checkout -b <new_branch_name>`	create new branch that you'll be working on and go to it
3. Make some file changes	work on your feature/implementation
4. `git add -p`	add changes to stage for commit, going through changes line by line
5. `git commit -m <commit message>`	commit files with a message
6. `git push -u origin <branch_name>`	push branch to remote and set to track (-u only needed if this is first push)
7. Repeat step 4-5.	work and commit often
8. `git push`	push work to remote branch for others to view
9. Follow the link given from the `git push` command to submit a pull request (PR) on GitHub online[11]	PR merges in work from your branch into master
(10.) Your changes and PR get approved, your reviewer deletes your remote branch upon merging	
11. `git fetch --all --prune`	clean up your local git by untracking deleted remote branches

Other helpful commands are listed below.

12.6 Commonly Used Git Commands

Table 12.2: Commonly used Git commands

Command	Description
`git clone <url> <directory_name>`	clone a repository, only needs to be done the first time
`git pull origin master`	pull from `master` before making any changes
`git branch`	check what branch you are on
`git branch -a`	check what branch you are on + all remote branches
`git checkout -b <new_branch_name>`	create new branch and go to it (only necessary when you create a new branch)
`git checkout <branch name>`	switch to branch

[11] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request#creating-the-pull-request

Command	Description
`git add <file name>`	add file to stage for commit
`git add -p`	adds changes to commit, showing you changes one by one
`git commit -m <commit message>`	commit file with a message
`git push -u origin <branch_name>`	push branch to remote and set to track (-u only works if this is first push)
`git branch --set-upstream-to origin <branch_name>`	set upstream to `origin/<branch_name>` (use if you forgot -u on first push)
`git push origin <branch_name>`	push work to branch
`git checkout <branch_name>` `git merge master`	switch to branch and merge changes from `master` into `<branch_name>` (two commands)
`git merge <branch_name> master`	switch to branch and merge changes from `master` into `<branch_name>` (one command)
`git checkout --track origin/<branch_name>`	pulls a remote branch and creates a local branch to track it (use when trying to pull someone else's branch onto your local computer)
`git push --delete <remote_name> <branch_name>`	delete remote branch
`git branch -d <branch_name>`	deletes local branch, -D to force
`git fetch --all --prune`	untrack deleted remote branches

12.7 How often should I commit?

It is good practice to commit every 15 minutes, or every time you make a significant change. It is better to commit more rather than less.

12.8 Repeated Amend Workflow

When working on a complex task, you may want to make frequent incremental commits to protect your progress, but avoid cluttering your Git history with many tiny "work in progress" commits. The **Repeated Amend** pattern lets you build up a polished commit gradually.

12.8.1 Basic Workflow

Start with a clean working tree in a functional state. Then:

1. Make a small change and verify your project still works
2. Stage and commit with a temporary message like "WIP" (work in progress)
3. **Do not push yet**
4. Make another small change and verify it works

5. Stage and amend the previous commit: `git commit --amend --no-edit`
6. Repeat steps 4-5 as needed
7. When finished, amend one final time with a proper commit message
8. Push your completed work

In RStudio, you can use the "Amend previous commit" checkbox when committing.

12.8.2 Key Points

- Each amend replaces the previous commit rather than creating a new one
- This keeps your history clean while letting you work incrementally
- Only use this pattern before pushing - never amend commits that others may have pulled
- If you need to undo changes, use `git reset --hard` to return to your last commit state
- Think of commits as climbing protection: use them when in uncertain territory

For more details and troubleshooting scenarios, see the Repeated Amend chapter[12] in Happy Git with R.

12.9 What should be pushed to Github?

Never push .Rout files! If someone else runs an R script and creates an .Rout file at the same time and both of you try to push to github, it is incredibly difficult to reconcile these two logs. If you run logs, keep them on your own system or (preferably) set up a shared directory where all logs are name and date timestamped.

There is a standardized `.gitignore` for `R` which you can download[13] and add to your project. This ensures you're not committing log files or things that would otherwise best be left ignored to GitHub. This is a great discussion of project-oriented workflows[14], extolling the virtues of a self-contained, portable projects, for your reference.

12.10 Customizing How Files Appear on GitHub

GitHub uses a tool called Linguist[15] to detect languages in your repository and generate language statistics. You can customize how certain files are treated by GitHub using a `.gitattributes` file in the root of your repository. This is particularly useful for marking generated files, documentation, or vendored code that shouldn't count toward your repository's language statistics.

[12] https://happygitwithr.com/repeated-amend.html
[13] https://github.com/github/gitignore/blob/master/R.gitignore
[14] https://tidyverse.org/blog/2017/12/workflow-vs-script/
[15] https://github.com/github-linguist/linguist/

12.10.1 The `linguist-generated` Attribute

One of the most useful attributes is `linguist-generated`, which marks files as generated code. Files marked this way are:

- **Excluded from language statistics** - they won't affect your repository's language breakdown
- **Hidden by default in diffs** - making pull request reviews cleaner and more focused on actual code changes

Common use cases for `linguist-generated` include:

- Compiled or minified files (e.g., `*.min.js`, `*.min.css`)
- Auto-generated documentation files
- Files generated by build tools or code generators
- Lock files that are updated automatically

For more details, see the Generated code documentation[16].

12.10.2 Using `.gitattributes`

Create a `.gitattributes` file in the root of your repository and add patterns for files you want to mark as generated:

```
# Mark minified JavaScript as generated
*.min.js linguist-generated

# Mark search index as generated
search/index.json linguist-generated

# Mark compiled CSS as generated
dist/styles.css linguist-generated
```

To unmark a file that would normally be considered generated:

```
# Don't treat bootstrap as generated
bootstrap.min.css -linguist-generated
```

The `.gitattributes` file uses the same pattern matching rules as `.gitignore`. For more details, see the pattern format documentation[17] and the Using gitattributes guide[18].

[16] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#generated-code
[17] https://www.git-scm.com/docs/gitignore#_pattern_format
[18] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#using-gitattributes

12.10.3 Other Useful Linguist Attributes

Beyond `linguist-generated`, you can use several other attributes:

- `linguist-vendored` - Marks vendored code (libraries you didn't write) to exclude from stats (documentation[19])
- `linguist-documentation` - Marks documentation files to exclude from stats (documentation[20])
- `linguist-detectable` - Forces a file type to be included in language stats (useful for data or prose files) (documentation[21])
- `linguist-language=<name>` - Overrides the detected language for syntax highlighting (documentation[22])

Example `.gitattributes` file:

```
# Exclude vendored dependencies
vendor/* linguist-vendored

# Exclude generated files
*.generated.ts linguist-generated
dist/* linguist-generated

# Mark documentation
docs/* linguist-documentation

# Force R Markdown files to be detected
*.Rmd linguist-detectable
```

For complete documentation, see Customizing how changed files appear on GitHub[23] and the Linguist overrides documentation[24].

[19] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#vendored-code
[20] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#documentation
[21] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#detectable
[22] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md#using-gitattributes
[23] https://docs.github.com/en/repositories/working-with-files/managing-files/customizing-how-changed-files-appear-on-github
[24] https://github.com/github-linguist/linguist/blob/main/docs/overrides.md

13 Unix

Adapted by UCD-SeRG team from original by Stephanie Djajadi, Kunal Mishra, Anna Nguyen, and Jade Benjamin-Chung[1]

We typically use Unix commands in Terminal (for Mac users) or Git Bash (for Windows users) to

1. Run a series of scripts in parallel or in a specific order to reproduce our work
2. To check on the progress of a batch of jobs
3. To use git and push to github

13.1 Basics

On the computer, there is a desktop with two folders, `folder1` and `folder2`, and a file called `file1`. Inside `folder1`, we have a file called `file2`. Mac users can run these commands on their terminal; it is recommended that Windows users use Git Bash, not Windows PowerShell.

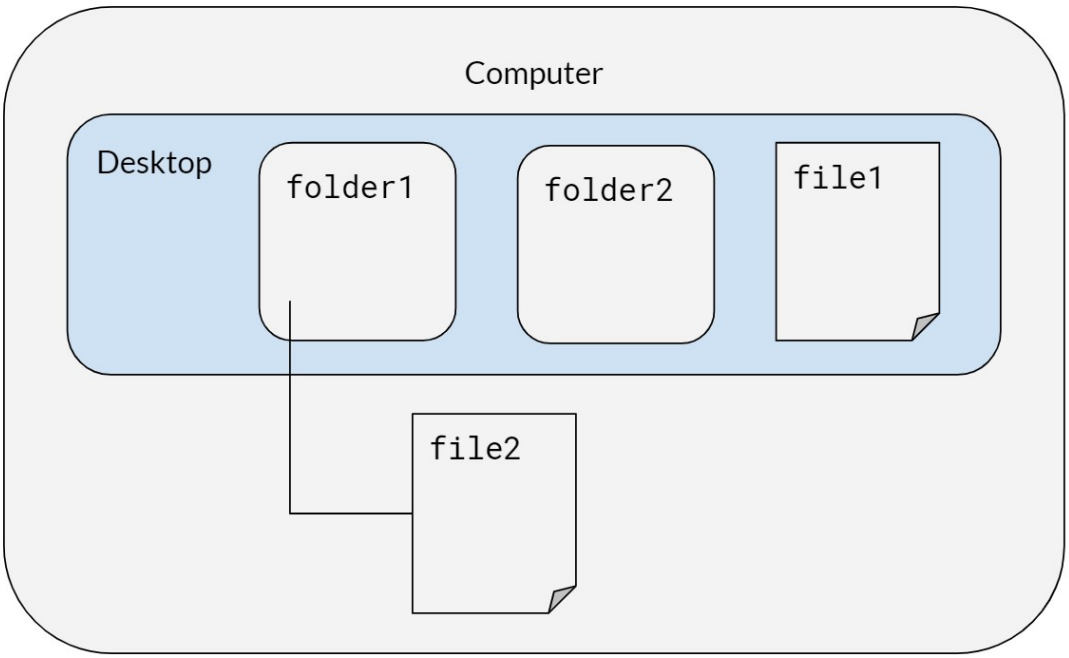

Figure 13.1: Example desktop with folders and files

[1] https://jadebc.github.io/lab-manual/unix.html

13.2 Syntax for both Mac/Windows

When typing in directories or file names, quotes are necessary if the name includes spaces.

Table 13.1: Basic Unix commands for Mac and Windows

Command	Description
`cd desktop/folder1`	Change directory to `folder1`
`pwd`	Print working directory
`ls`	List files in the directory
`cp "file2" "newfile2"`	Copy file (remember to include file extensions when typing in file names like `.pdf` or `.R`)
`mv "newfile2" "file3"`	Rename `newfile2` to `file3`
`cd ..`	Go to parent of the working directory (in this case, `desktop`)
`mv "file1" folder2`	Move `file1` to `folder2`
`mkdir folder3`	Make a new folder in `folder2`
`rm <filename>`	Remove files
`rm -rf folder3`	Remove directories (`-r` will attempt to remove the directory recursively, `-rf` will force removal of the directory)
`clear`	Clear terminal screen of all previous commands

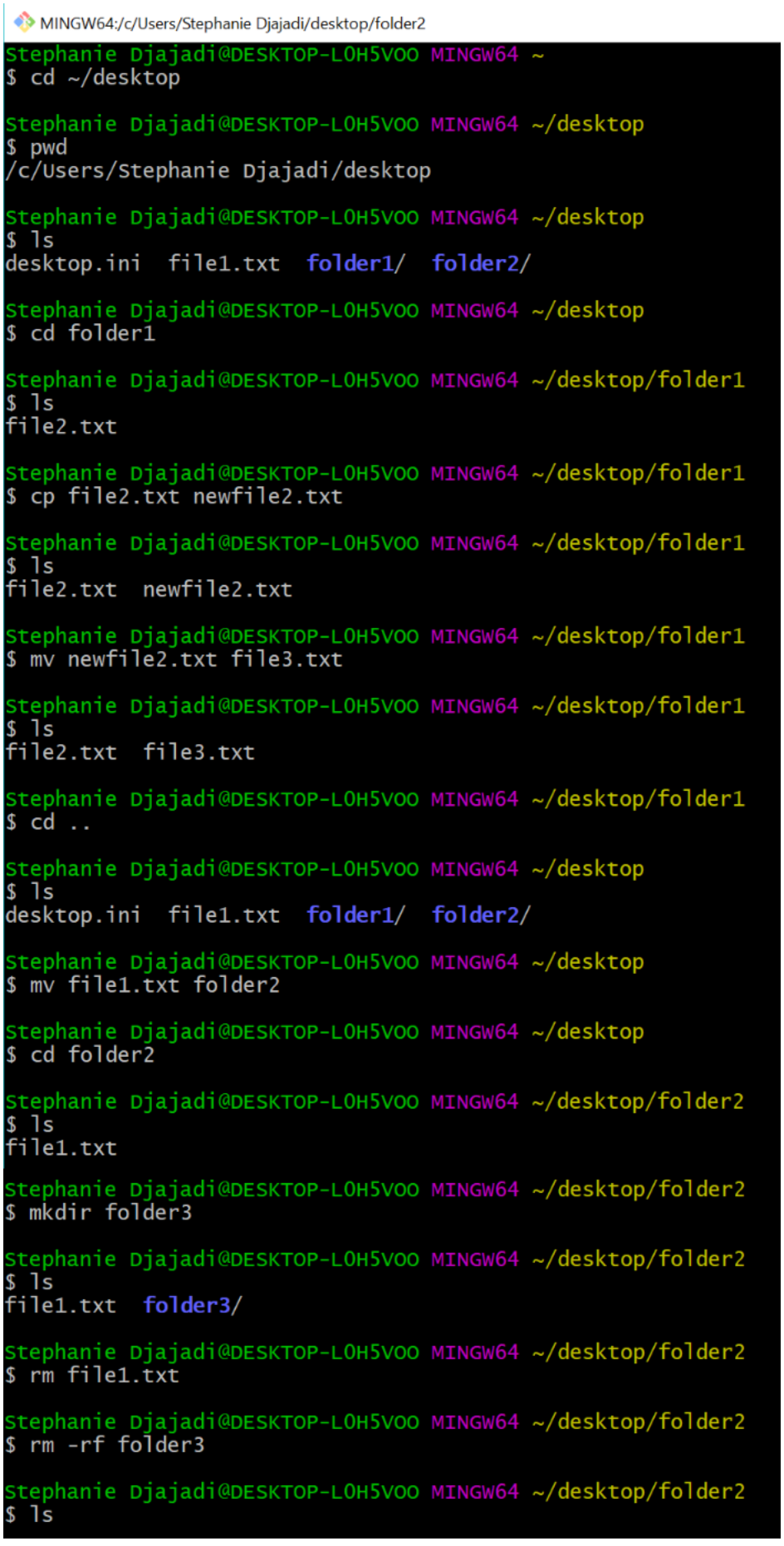

Figure 13.2: Terminal output after executing basic Unix commands

13.3 Running Bash Scripts

Table 13.2: Commands for running Bash scripts

Windows	Mac / Linux	Description
`chmod +750 <filename.sh>`	`chmod +x <filename.sh>`	Change access permissions for a file (only needs to be done once)
`./<filename.sh>`	`./<filename.sh>`	Run file (`./` to run any executable file)
`bash bash_script_name.sh &`	`bash bash_script_name.sh &`	Run shell script in the background

13.4 Running Rscripts in Windows

Note: This code seems to work only with Windows Command Prompt, not with Git Bash.

When R is installed, it comes with a utility called Rscript. This allows you to run R commands from the command line. If Rscript is in your `PATH,` then typing Rscript into the command line, and pressing enter, will not error. Otherwise, to use Rscript, you will either need to add it to your PATH (as an environment variable), or append the full directory of the location of Rscript on your machine. To find the full directory, search for where R is installed your computer. For instance, it may be something like below (this will vary depending on what version of R you have installed):

```
C:\Program Files\R\R-3.6.0\bin
```

For appending the `PATH` variable, please view this link[2]. I strongly recommend completing this option.

If you add the PATH as an environment variable, then you can run this line of code to test: `Rscript -e "cat('this is a test')"`, where the `-e` flag refers to the expression that will be executed.

If you do not add the PATH as an environment variable, then you can run this line of code to replicate the results from above: `"C:\Program Files\R\R-3.6.0\bin" -e "cat('this is a test')"`

To run an R script from the command line, we can say: `Rscript -e "source('C:/path/to/script/some_code`

13.4.1 Common Mistakes

- Remember to include all of the quotation marks around file paths that have a spaces.
- If you attempt to run an R script but run into `Error: '\U' used without hex digits in character string starting "'C:\U"`, try replacing all `\` with `\\` or `/`.

[2] https://www.howtogeek.com/118594/how-to-edit-your-system-path-for-easy-command-line-access/

13.5 Checking tasks and killing jobs

Windows	Mac / Linux	Description
`tasklist`	`ps -v`	List all processes on the command line
	`top -o [cpu/rsize]`	List all running processes, sorted by CPU or memory usage
`taskkill /F /PID pid_number`	`kill <PID_number>`	Kill a process by its process ID
`taskkill /IM "process name" /F`		Kill a process by its name
`start /b program.exe`		Runs jobs in the background (exclude `/b` if you want the program to run in a new console)
	`nohup`	Prevents jobs from stopping
	`disown`	Keeps jobs running in the background even if you close R
`taskkill /?`		Help, lists out other commands

To kill a task in Windows, you can also go to Task Manager > More details > Select your desired app > Click on End Task.

13.6 Running big jobs

For big data workflows, the concept of "backgrounding" a bash script allows you to start a "job" (i.e. run the script) and leave it overnight to run. At the top level, a bash script (`0-run-project.sh`) that simply calls the directory-level bash scripts (i.e. `0-prep-data.sh`, `0-run-analysis.sh`, `0-run-figures.sh`, etc.) is a powerful tool to rerun every script in your project. See the included example bash scripts for more details.

- **Running Bash Scripts in Background**: Running a long bash script is not trivial. Normally you would run a bash script by opening a terminal and typing something like `./run-project.sh`. But what if you leave your computer, log out of your server, or close the terminal? Normally, the bash script will exit and fail to complete. To run it in background, type `./run-project.sh &; disown`. You can see the job running (and CPU utilization) with the command `top` or `ps -v` and check your memory with `free -h`.

Alternatively, to keep code running in the background even when an SSH connection is broken, you can use `tmux`. In terminal or gitbash follow the steps below. This site[3] has useful tips on using `tmux`.

[3] https://medium.com/@jeongwhanchoi/install-tmux-on-osx-and-basics-commands-for-beginners-be22520fd95e

```
# create a new tmux session called session_name
tmux new -ssession_name

# run your job of interest
R CMD BATCH myjob.R &

# check that it is running
ps -v

# to exit the tmux session (Mac)
ctrl + b
d

# to reopen the tmux session to kill the job or
# start another job
tmux attach -tsession_name
```

- **Deleting Previously Computed Results**: One helpful lesson we've learned is that your bash scripts should remove previous results (computed and saved by scripts run at a previous time) so that you never mix results from one run with a previous run. This can happen when an R script errors out before saving its result, and can be difficult to catch because your previously saved result exists (leading you to believe everything ran correctly).
- **Ensuring Things Ran Correctly**: You should check the `.Rout` files generated by the R scripts run by your bash scripts for errors once things are run. A utility file is include in this repository, called `runFileSaveLogs`, and is used by the example bash scripts to… run files and save the generated logs. It is an awesome utility and one I definitely recommend using. Before using `runFileSaveLogs`, it is necessary to put the file in the home working directory. For help and documentation, you can use the command `./runFileSaveLogs -h`. See example code and example usage for `runFileSaveLogs` below.

13.6.1 Example code for `runfileSaveLogs`

```
#!/usr/bin/env python3
# Type "./runFileSaveLogs -h" for help

import os
import sys
import argparse
import getpass
import datetime
import shutil
import glob
import pathlib

# Setting working directory to this script's current directory
os.chdir(os.path.dirname(os.path.abspath(__file__)))
```

```
# Setting up argument parser
parser = argparse.ArgumentParser(description='Runs the argument R script(s) - in parallel i

# Function ensuring that the file is valid
def is_valid_file(parser, arg):
    if not os.path.exists(arg):
        parser.error("The file %s does not exist!" % arg)
    else:
        return arg

# Function ensuring that the directory is valid
def is_valid_directory(parser, arg):
    if not os.path.isdir(arg):
        parser.error("The specified path (%s) is not a directory!" % arg)
    else:
        return arg

# Additional arguments that can be added when running runFileSaveLogs
parser.add_argument('-p', '--parallel', action='store_true', help="Runs the argument R scri
parser.add_argument("-i", "--identifier", help="Adds an identifier to the directory name wh
parser.add_argument('filenames', nargs='+', type=lambda x: is_valid_file(parser, x))

args = parser.parse_args()
args_dict = vars(args)

print(args_dict)

# Run given R Scripts
for filename in args_dict["filenames"]:
  system_call = "R CMD BATCH" + " " + filename
  if args_dict["parallel"]:
    system_call = "nohup" + " " + system_call + " &"

  os.system(system_call)

# Create the directory (and any parents) of the log files
currentUser = getpass.getuser()
currentTime = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
logDirPrefix = "/home/kaiserData/logs/" # Change to the directory where the logs should be
logDir = logDirPrefix + currentTime + "-" + currentUser

# If specified, adds the identifier to the filename of the log
if args.identifier is not None:
  logDir += "-" + args.identifier

logDir += "/"

pathlib.Path(logDir).mkdir(parents=True, exist_ok=True)

# Find and move all logs to this new directory
```

```
currentLogPaths = glob.glob('./*.Rout')

for currentLogPath in currentLogPaths:
  filename = currentLogPath.split("/")[-1]
  shutil.move(currentLogPath, logDir + filename)
```

13.6.2 Example usage for `runfileSaveLogs`

This example bash script runs files and generates logs for five scripts in the `kaiserflu/3-figures` folder. Note that the `-i` flag is used as an identifier to add `figures` to the filename of each log.

```
#!/bin/bash

# Copy utility run script into this folder for concision in call
cp ~/kaiserflu/runFileSaveLogs ~/kaiserflu/3-figures/

# Run folder scripts and produce output
cd ~/kaiserflu/3-figures/
./runFileSaveLogs -i "figures" \
fig-mean-season-age.R \
fig-monthly-rate.R \
fig-point-estimates-combined.R \
fig-point-estimates.R \
fig-weekly-rate.R

# Remove copied utility run script
rm runFileSaveLogs
```

14 Reproducible Environments

Adapted by UCD-SeRG team from original by Anna Nguyen[1]

14.1 Package Version Control with renv

14.1.1 Introduction

Replicable code should produce the same results, regardless of when or where it's run. However, our analyses often leverage open-source R packages that are developed by other teams. These packages continue to be developed after research projects are completed, which may include changes to analysis functions that could impact how code runs for both other team members and external replicators.

For example, suppose we had used a function that took in one argument, such that our code contained `example_function(arg_a = "a")`. A few months after we publish our code, the package developers update the function to take in another mandatory argument `arg_b`. If someone runs our code, but has the most recent version of the package, they'll receive an error message that the argument `arg_b` is missing and will not be able to full reproduce our results.

To ensure that the right functions are used in replication efforts, it is important for us to keep track of package versions used in each project.

`renv` can be to promote reproducible environments within R projects. `renv` creates individual package libraries for each project instead of having all projects, which may use different versions of the same package, share the same package library. However, for projects that use many packages, this process can be memory intensive and increase the time needed for a new users to start running code.

In this lab manual chapter, we provide a quick tutorial for integrating `renv` into research workflows. For more detailed instructions, please refer to the `renv` package vignette.

14.1.2 Implementing renv in projects

Ideally, `renv` should be initiated at the start of projects and updated continuously when new packages are introduced in the codebase. However, this process can be initiated at any point in a project

To add `renv` to your workflow, follow these steps:

1. Install the `renv` package by running `install.packages("renv")`
2. Create an RProject file and ensure that your working directory is set to the correct folder

[1] https://jadebc.github.io/lab-manual/reproducible-environments.html

3. In the R console, run `renv::init()` to initialize renv in your R Project
4. This will create the following files: `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R`. Commit and push these files to GitHub so that they're accessible to other users.
5. As you write code, update the project's R library by running `renv::snapshot()` in the R console
6. Add `renv::restore()` to the head of your config file, to make sure that all users that run your code are on the same package versions.

14.1.3 Configuring renv settings

The `renv/settings.json` file created during initialization allows you to customize how `renv` behaves in your project. One useful setting is `snapshot.dev`, which controls whether development dependencies are included by default when calling `renv::snapshot()` or `renv::status()`.

14.1.3.1 Reducing startup messages

When working on projects, you may encounter startup messages indicating that `renv` is out of sync with the lockfile. To reduce these messages in most projects, add the following setting to `renv/settings.json`:

```
"snapshot.dev": true
```

This setting (available in `renv` version 1.1.6 and later) includes development dependencies in snapshots by default, which helps keep the lockfile aligned with your actual usage and eliminates many synchronization warnings.

If startup messages about being out of sync persist after enabling this setting, use `renv::restore()` to sync your local library with the lockfile, or `renv::snapshot()` to update the lockfile with your current package versions.

For more details on `renv` configuration options, see the official renv documentation[2].

14.1.4 Using projects with renv

If you're starting to work on an ongoing project that already has `renv` set up, follow these steps to ensure that you're using the same project versions.

1. Install the `renv` package by running `install.packages("renv")`
2. Pull the most updated version of the project from GitHub
3. Open the project's RProject file
4. Run `renv::restore()`. In our lab's projects, this is often already found at the top of the config file, so you can just run scripts as is.
5. This will pull up a list of the project's packages that need to be updated for you to be consistent with the project. The console will ask if you want to proceed with updating these packages - type "Y" to continue.

[2] https://rstudio.github.io/renv/

6. Wait for the correct versions of each package to install/update. This may take some time, depending on how many packages the project uses.
7. Your R environment should now be using the same package versions as specified in the `renv` lock file. You should now be able to replicate the code.
8. If you make edits to the code and introduce new/updated packages, see the section above for instructions on how to make updates.

15 Code Publication

Adapted by UCD-SeRG team from original by Nolan Pokpongkiat[1]

15.1 Checklist overview

1. Fill out file headers
2. Clean up comments
3. Document functions
4. Remove deprecated filepaths
5. Ensure project runs via bash
6. Complete the README
7. Clean up feature branches
8. Create Github release

15.2 Fill out file headers

Every file in a project should have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. See template here.[2]

15.3 Clean up comments

Make sure comments in the code are for code documentation purposes only. Do not leave comments to self in the final script files.

15.4 Document functions

Every function you write must include a header to document its purpose, inputs, and outputs. See template for the function documentation here.[3]

[1] https://jadebc.github.io/lab-manual/code-publication.html
[2] https://ucd-serg.github.io/lab-manual/coding-practices.html#file-headers
[3] https://ucd-serg.github.io/lab-manual/coding-practices.html#function-documentation

15.5 Remove deprecated filepaths

All file paths should be defined in 0-config.R, and should be set relative to the project working directory. All absolute file paths from your local computer should be removed, and replaced with a relative path. If a third party were to re-run this analysis, if they need to download data from a separate source and change a filepath in the 0-config.R to match, make sure to specify in the README which line of 0-config.R needs to be substituted.

15.6 Ensure project runs via bash

The project should be configured to be entirely reproducible by running a master bash script, run-project.sh, which should live at the top directory. This bash script can call other bash scripts in subfolders, if necessary. Bash scripts should use the runFileSaveLogs utility script, which is a wrapper around the Rscript command, allowing you to specify where .Rout log files are moved after the R scripts are run.

See usage and documentation here.[4]

15.7 Complete the README

A README.md should live at the top directory of the project. This usually includes a Project Overview and a Directory Structure, along with the names of the contributors and the Creative Commons License. See below for a template:

> **Overview**
>
> To date, coronavirus testing in the US has been extremely limited. Confirmed COVID-19 case counts underestimate the total number of infections in the population. We estimated the total COVID-19 infections – both symptomatic and asymptomatic – in the US in March 2020. We used a semi-Bayesian approach to correct for bias due to incomplete testing and imperfect test performance.
>
> **Directory structure**
>
> - 0-config.R: configuration file that sets data directories, sources base functions, and loads required libraries
> - 0-base-functions: folder containing scripts with functions used in the analysis
>   - 0-base-functions.R: R script containing general functions used across the analysis
>   - 0-bias-corr-functions.R: R script containing functions used in bias correction
>   - 0-bias-corr-functions-undertesting.R: R script containing functions used in bias correction to estimate the percentage of underestimation due to incomplete testing vs. imperfect test accuracy
>   - 0-prior-functions.R: R script containing functions to generate priors

[4] https://ucd-serg.github.io/lab-manual/unix.html#example-code-for-runfilesavelogs

- 1-data: folder containing data processing scripts NOTE: some scripts are deprecated
- 2-analysis: folder containing analysis scripts. To rerun all scripts in this subdirectory, run the bash script 0-run-analysis.sh.
  - 1-obtain-priors-state.R: obtain priors for each state
  - 2-est-expected-cases-state.R: estimate expected cases in each state
  - 3-est-expected-cases-state-perf-testing.R: estimate expected cases in each state, estimate the percentage of underestimation due to incomplete testing vs. imperfect test accuracy
  - 4-obtain-testing-protocols.R: find testing protocols for each state.
  - 5-summarize-results.R: summarize results; obtain results for in text numerical results.
- 3-figure-table-scripts: folder containing figure scripts. To rerun all scripts in this subdirectory, run the bash script 0-run-figs.sh.
  - 1-fig-testing.R: creates plot of testing patterns by state over time
  - 2-fig-cases-usa-state-bar.R: creates bar plot of confirmed vs. estimated infections by state
  - 3a-fig-map-usa-state.R: creates map of confirmed vs. estimated infections by state
  - 3b-fig-map-usa-state-shiny.R: creates map of confirmed vs. estimated infections by state with search functionality by state
  - 4-fig-priors.R: creates figure with priors for US as a whole
  - 5-fig-density-usa.R: creates figure of distribution of estimated cases in the US
  - 6-table-data-quality.R: creates table of data quality grading from COVID Tracking Project
  - 7-fig-testpos.R: creates figure of the probability of testing positive among those tested by state
  - 8-fig-percent-undertesting-state.R: creates figure of the percentage of under estimation due to incomplete testing
- 4-figures: folder containing figure files.
- 5-results: folder containing analysis results objects.
- 6-sensitivity: folder containing scripts to run the sensitivity analyses

Contributors: UCD-SeRG team (adapted from original contributors: Jade Benjamin-Chung, Sean L. Wu, Anna Nguyen, Stephanie Djajadi, Nolan N. Pokpongkiat, Anmol Seth, Andrew Mertens)

Wu SL, Mertens A, Crider YS, Nguyen A, Pokpongkiat NN, Djajadi S, et al. Substantial underestimation of SARS-CoV-2 infection in the United States due to incomplete testing and imperfect test accuracy. medRxiv. 2020; 2020.05.12.20091744. doi:10.1101/2020.05.12.20091744

When possible, also include a description of the RDS results that are generated, detailing what data sources were used, where the script lives that creates it, and what information the RDS results hold.

15.8 Clean up feature branches

In the remote repository on Github, all feature branches aside from master should be merged in and deleted. All outstanding PRs should be closed.

15.9 Create Github release

Once all of these items are verified, create a tag to make a Github release, which will tag the repository, creating a marker at this specific point in time.

Detailed instructions here.[5]

[5] https://docs.github.com/en/enterprise/2.13/user/articles/creating-releases

16 Data Publication

Adapted from Fanice Nyatigo and Ben Arnold's chapter in the Proctor-UCSF Lab Manual[1]

16.1 Overview

Warning! ***NEVER*** *push a dataset into the public domain (e.g., GitHub, OSF) without first checking with lab leadership to ensure that it is appropriately de-identified and we have approval from the sponsor and/or human subjects review board to do so. For example, we will need to re-code participant IDs (even if they contain no identifying information) before making data public to completely break the link between IDs and identifiable information stored on our servers.*

If you are releasing data into the public domain, then consider making available *at minimum* a `.csv` file and a codebook of the same name (note: you should have a codebook for internal data as well). We often also make available `.rds` files as well. For example, your `mystudy/data/public` directory could include three files for a single dataset, two with the actual data in `.rds` and `.csv` formats, and a third that describes their contents:

```
analysis_data_public.csv
analysis_data_public.rds
analysis_data_public_codebook.txt
```

In general, datasets are usually too big to save on GitHub, but occasionally they are small. Here is an example of where we actually pushed the data directly to GitHub: https://github.com/ben-arnold/enterics-seroepi/tree/master/data .

If the data are bigger, then maintaining them under version control in your git repository can be unwieldy. Instead, we recommend using another stable repository that has version control, such as the Open Science Framework ("Open Science Framework," n.d.). For example, all of the data from the WASH Benefits trials (led by investigators at Berkeley, icddr,b, IPA-Kenya and others) are all stored through data components nested within in OSF projects: https://osf.io/tprw2/. Another good option is Dryad Digital Repository ("Dryad Digital Repository," n.d.) or institutional digital repositories.

[1] https://urlisolation.com/browser?clickId=524DE241-3F8F-4C98-B619-3C278374BF64&traceToken=1728923499%3Bucsfmed_hosted%3Bhttps%3A%2F%2Fproctor-ucsf.github.io%2Fd&url=https%3A%2F%2Fproctor-ucsf.github.io%2Fdcc-handbook%2Fpublicdata.html

We recommend cross-linking public files in GitHub (scripts/notebooks only) and OSF/Dryad/institutional digital repositories.

Below are the main steps to making data public, after finalizing the analysis datasets and scripts:
1. Remove Protected Health Information (PHI)
2. Create public IDs or join already created public IDs to the data
3. Create an OSF repository and/or Dryad/institutional digital repository
4. Edit analysis scripts to run using the public datasets and test (optional)
5. Create a public github page for analysis scripts and link to OSF and/or Dryad/Zenodo
6. Go live

16.2 Removing PHI

Once the data is finalized for analysis, the first step is to strip it of Protected Health Information (PHI), or any other data that could be used to link back to specific participants, such as names, birth dates, or GPS coordinates at the village/neighborhood level or below. PHI includes, but is not limited to:

16.2.1 Personal information

These are identifiers that directly point to specific individuals, such as:
- Names, addresses, photographs, date of birth
- A combination of age, sex, and geographic location (below population 20,000) is considered identifiable

16.2.2 Dates

Any specific dates (e.g., study visit dates, birth dates, treatment dates) are usually problematic.
- If a dataset requires high resolution temporal information, coarsen visit or measurement dates to be two variables: year and week of the year (1-52).
- If a dataset requires age, provide that information without a birth date (typically month resolution is sufficient)

Caution! *If making changes to the format of dates or ages, make sure your analysis code runs on these modified versions of the data (step 3)!*

16.2.3 Geographic information

Do not include GPS coordinates (longitude, latitude) except in special circumstances where they have been obfuscated/shifted. Reach out to lab leadership before doing this because it can be complicated.

Do not include place names or codes (e.g., US Zip Codes) if the place contains <20,000 people. For villages or neighborhoods, code them with uninformative IDs. For sub-districts or districts, names are fine.

If an analysis requires GPS locations (e.g., to make a map), then typically we include a disclaimer in the article's data availability statement that explains we cannot make GPS locations public to protect participant confidentiality. As a middle ground, we typically make our *code* public that runs on the geo-located data for transparency, even if independent researchers can't actually run that code (although please be careful to ensure the code itself does not in any way include geographic identifiers).

For more examples of what constitutes PHI, please refer to this link: https://cphs.berkeley.edu/hipaa/hipaa18.html

For learning about working with geospatial data, see the UC Davis DataLab GIS workshops[2], including resources on QGIS and spatial SQL.

16.3 Create public IDs

16.3.1 Rationale

The UC Davis IRB requires that public datasets not include the original study IDs to identify participants or other units in the study (such as village IDs). The reason is that those IDs are linked in our private datasets to PHI. By creating a new set of public IDs, the public dataset is one step further removed from the potential to link to PHI.

16.3.2 A single set of public IDs for each study

For each study, it is ideal to create a single set of public IDs whenever possible. We could create a new set of public IDs for every public dataset, but the downside is that independent researchers could no longer link data that might be related. By creating a single set of public IDs associated with each internal study ID, public files retain the link.

Maintaining a single set of public IDs requires a shared "bridge" dataset, that includes a row for each study ID and has the associated public ID. For studies with multiple levels of ID, we would typically have separate bridge datasets for each type of ID (e.g,. cluster ID, participant ID, etc.)

Create a public ID that can be used to uniquely identify participants and that can internally be linked to the original study IDs. We recommend creating a subdirectory in the study's shared data directory to store the public IDs. The shared location enables multiple projects to use the same IDs. Create the IDs using a script that reads in the study IDs, creates a unique (uninformative) public ID for the study IDs, and then saves the bridge dataset. The script should be saved in the same directory as the public ID files.

[2] https://github.com/ucdavisdatalab

Caution! *Note that small differences may arise if the new public IDs do not necessarily order participants in the same way as the internal IDs. The small differences are all in estimates that rely on resampling, such as Bootstrap CIs, permutation P-values, and TMLE, as the resampling process may lead tp slightly different re-samples. The key here, to ensure the results are consistent irrespective of the dataset used, is simply to not assign public IDs randomly. Use* `rank()` *on the internal ID instead of* `row_number()` *to ensure that the order is always the same.*

16.3.3 Example scripts

We have created a self-contained and reproducible example that you can run and replicate when making data public for your projects. It contains the following files and folders:

1. `data/final/`- folder containing the projects final data in both csv and rds formats

2. `code/DEMO_generate_public_IDs.R`- creates randomly generated public IDs that can be matched to the trial's assigned patient IDs.

3. `data/make_public/DEMO_internal_to_publicID.csv`- the output from step #2, a bridge dataset with two variables- the new public ID and the patient's assigned ID.

4. `code/DEMO_create_public_datasets.R`- joins the public IDs to the trial's full dataset, and strips it of the assigned patient ID.
5. `data/public/`- folder containing the output from step #3- de-identified public dataset, in csv and rds formats, with uniquely identifying public IDs that cannot be easily linked back to the patient's ID.

The example workflow is accessible via GitHub: https://github.com/proctor-ucsf/dcc-handbook/tree/master/templates/making-data-public

16.4 Create a data repository

First, ensure that you create a codebook and metadata file for each public dataset See the DCC guide on Documenting datasets[3]. Use the same name as the datasets, but with "-codebook.txt" / "-codebook.html" / "-codebook.csv" at the end (depending on the file format for the codebook). One nice option is the R codebook package, which also generates JSON output that is machine-readable.

For additional guidance on data documentation best practices, see the UC Davis DataLab workshop on data documentation[4].

[3] https://proctor-ucsf.github.io/dcc-handbook/datawrangling.html#documenting-datasets
[4] https://github.com/ucdavisdatalab/workshop_how-to-data-documentation

16.4.1 Steps for creating an Open Science Framework (OSF) repository:

1. Create a new OSF project per these instructions: https://help.osf.io/article/252-create-a-project
2. Create a data component and upload the datasets in .csv and .rds format along with the codebooks. The primary format for public dissemination is .csv but we make the .rds files available too as auxiliary files for convenience.
3. Create a notebook component and upload the final .html files (which will not be on github… but see optional item below)
4. On the OSF landing Wiki, provide some context. Here is a recent example: https://osf.io/954bt/
5. Create a Digital Object Identifier (DOI) for the repository. A DOI is a unique identifier that provides a persistent link to content, such as a dataset in this case. Learn more about DOIs[5]
6. Optional: Complete the software checklist and system requirement guide for the analysis to guide others. Include it on the GitHub README for the project: https://github.com/proctor-ucsf/mordor-antibody

16.5 Edit and test analysis scripts

Make minor changes to the analysis scripts so that they run on public data. If using version control in GitHub, the most straight-forward way is to create a branch from the main git branch that reads in the public files, and then renames the new public ID variable, e.g., "id_public" to the internally recognized ID variable name, e.g. "recordID", when reading in the public data. Re-run all the analysis scripts to ensure that they still work with the public version of the dataset.

16.6 Create a public GitHub page for public scripts

At minimum, we should include all of the scripts required to run the analyses. **IMPORTANT**: ensure you have taken a snapshot and saved your computing environment using the `renv` package (`renv`).

See examples:
- ACTION - https://github.com/proctor-ucsf/ACTION-public
- NAITRE - https://github.com/proctor-ucsf/NAITRE-primary

> ***Caution!*** *Read through the scripts carefully to ensure there is no PHI in the code itself*

Once a public GitHub page exists, you can create a new component on an OSF project (step 3, above) and link it to the public version of the GitHub repo.

[5] https://researchdata.princeton.edu/research-lifecycle-guide/publishing-and-preservation/dois

16.7 Go live

On GitHub, it is useful to create an official "release" version to freeze the repository, where you can have "associated files" with each version. Include the .html notebook output as additional files — since they aren't tracked in GitHub, it does provide a way of freezing / saving the HTML output for us and others. OSF examples of a studies from UCSF's Proctor Foundation:
- ACTION - https://osf.io/ca3pe/
- NAITRE - https://osf.io/ujeyb/
- MORDOR Niger antibody study - https://osf.io/dgsq3/

Further reading on end-to-end data management: How to Store and Manage Your Data - PLOS ("How to Store and Manage Your Data," n.d.)

17 High-performance computing (HPC)

Adapted by UCD-SeRG team from original by Anna Nguyen, Jade Benjamin-Chung, and Gabby Barratt Heitmann[1]

When you need to run a script that requires a large amount of RAM, large files, or that uses parallelization, UC Davis provides several high-performance computing (HPC) resources.

17.1 UC Davis Computing Resources

17.1.1 Available Resources

UC Davis HPC Clusters: - **Farm Cluster** (hpc.ucdavis.edu[2]): UC Davis's primary HPC cluster providing shared computing resources for research

PHS Shared Compute Environments: For lab members affiliated with the School of Public Health Sciences (PHS), additional shared computing environments are available. These environments provide secure, HIPAA-compliant computing resources suitable for working with sensitive health data.

- **Shiva** (shiva.ucdavis.edu): SLURM-based cluster for computational work
- **Mercury** (mercury.ucdavis.edu): RStudio GUI computing environment

For detailed information about PHS shared compute environments, including access procedures, security guidelines, and usage policies, please refer to the PHS Shared Compute Environments Guide[3].

Contact lab leadership for assistance with: - Requesting access to computing resources - Choosing the appropriate computing environment for your project - Setting up your computing environment

17.2 Getting started with SLURM clusters

To access a UC Davis HPC cluster, in terminal, log in using SSH. For example, to access shiva:

```
ssh USERNAME@shiva.ucdavis.edu
```

[1] https://jadebc.github.io/lab-manual/slurm.html
[2] https://hpc.ucdavis.edu/
[3] assets/files/PHS_Shared_Compute_Environments.pdf

You will be prompted to enter your UC Davis credentials and may need to complete two-factor authentication.

Once you log in, you can view the contents of your home directory in command line by entering `cd $HOME`. You can create subfolders within this directory using the `mkdir` command. For example, you could make a "code" subdirectory and clone a Github repository there using the following code:

```
cd $HOME
mkdir code
git clone https://github.com/jadebc/covid19-infections.git
```

17.2.1 One-Time System Set-Up

To keep the install packages consistent across different nodes, you will need to explicitly set the pathway to your R library directory.

Open your `~/.Renviron` file (`vi ~/.Renviron`) and append the following line:

Note: Once you open the file using `vi [file_name]`, you must press `i` (on Mac OS) or `Insert` (on Windows) to make edits. After you finish, hit `Esc` to exit editing mode and type `:wq` to save and close the file.

```
R_LIBS=~/R/x86_64-pc-linux-gnu-library/4.0.2
```

Alternatively, run an R script with the following code on the cluster:

```
r_environ_file_path = file.path(Sys.getenv("HOME"), ".Renviron")
if (!file.exists(r_environ_file_path)) file.create(r_environ_file_path)

cat("\nR_LIBS=~/R/x86_64-pc-linux-gnu-library/4.0.2",
    file = r_environ_file_path, sep = "\n", append = TRUE)
```

To load packages that run off of C++, you'll need to set the correct compiler options in your R environment.

Open the Makevars file (`vi ~/.R/Makevars`) and append the following lines

```
CXX14FLAGS=-O3 -march=native -mtune=native -fPIC
CXX14=g++
```

Alternatively, create an R script with the following code, and run it on the cluster:

```
dotR = file.path(Sys.getenv("HOME"), ".R")
if (!file.exists(dotR)) dir.create(dotR)

M = file.path(dotR, "Makevars")
if (!file.exists(M)) file.create(M)

cat("\nCXX14FLAGS=-O3 -march=native -mtune=native -fPIC",
    "CXX14=g++",
    file = M, sep = "\n", append = TRUE)
```

17.3 Moving files to the cluster

The `$HOME` directory is a good place to store code and small test files. Save large files to the `$SCRATCH` directory or other designated storage areas. Check with the UC Davis HPC documentation[4] for specific quotas and retention policies. It's best to create a bash script that records the file transfer process for a given project. See example code below:

```
# note: the following steps should be done from your local
# (not after ssh-ing into the cluster)

# securely transfer folders from Box to cluster home directory
# note: the -r option is for folders and is not needed for files
scp -r "Box/project-folder/folder-1/" USERNAME@shiva.ucdavis.edu:/home/users/USERNAME/

# securely transfer folders from Box to your cluster scratch directory
scp -r "Box/project-folder/folder-2/" USERNAME@shiva.ucdavis.edu:/scratch/users/USERNAME/

# securely transfer folders from Box to shared scratch directory
scp -r "Box/project-folder/folder-3/" USERNAME@shiva.ucdavis.edu:/scratch/group/GROUPNAME/
```

17.4 Installing packages on the cluster

When you begin working on a cluster, you will most likely encounter problems with installing packages. To install packages, login to the cluster on the command line and open a development node. Do not attempt to do this in RStudio Server, as you will have to re-do it for every new session you open.

```
ssh USERNAME@shiva.ucdavis.edu

sdev
```

You should only have to install packages once. The cluster may require that you specify the repository where the package is downloaded from. You may also need to add an additional argument to install.packages to prevent the packages from locking after installation:

```
install.packages(<PACKAGE NAME>, repos="https://cran.r-project.org",
                 INSTALL_opts = "--no-lock")
```

In order for some R packages to work on clusters, it is necessary to load specific software modules before running R. These must be loaded each time you want to use the package in R. For example, for spatial and random effects analyses, you may need the modules/packages below. These modules must also be loaded on the command line prior to opening R in order for package installation to work.

[4] https://hpc.ucdavis.edu/

```
module --force purge # remove any previously loaded modules, including math and devel
module load math
module load math gmp/6.1.2
module load devel
module load gcc/10
module load system
module load json-glib/1.4.4
module load curl/7.81.0
module load physics
module load physics udunits geos
module load physics gdal/2.2.1 # for R/4.0.2
module load physics proj/4.9.3 # for R/4.0.2
module load pandoc/2.7.3

module load R/4.0.2

R # Open R in the Shell window to install individual packages or test code
Rscript install-packages.R # Alternatively, run a package installation script in the Shell
```

Figuring out the issues with some packages will require some trial and error. If you are still encountering problems installing a package, you may have to install other dependencies manually by reading through the error messages. If you try to install a dependency from CRAN and it isn't working, it may be a module. You can search for it using the `module spider` command:

```
module spider DEPENDENCY NAME
```

You can also reach out to UC Davis HPC support for help. Visit hpc.ucdavis.edu[5] for support information.

17.5 Testing your code

Both of the following ways to test code on a cluster are recommended for making small changes, such as editing file paths and making sure the packages and source files load. You should write and test the functionality of your script locally, only testing on the cluster once major bugs are out.

17.5.1 The command line

There are two main ways to explore and test code on computing clusters. The first way is best for users who are comfortable working on the command line and editing code in base R. Even if you are not comfortable yet, this is probably the better way because these commands will transfer between different cluster computers using Slurm.

Typically, you will want to initially test your scripts by initiating a development node using the command `sdev`. This will allocate a small amount of computing resources for 1 hour. You can access R via command line using the following code.

[5] https://hpc.ucdavis.edu/

```
# open development node
sdev

# Load all the modules required by the packages you are using
module load MODULE NAME

# Load R (default version)*
module load R

# initiate R in command line
R
```

*Note: for collaboration purposes, it's best for everyone to work with one version of R. Check what version is being used for the project you are working on. Some packages only work with some versions of R, so it's best to keep it consistent.

17.5.2 RStudio Server

For RStudio GUI computing, UC Davis provides mercury.ucdavis.edu. This is accessed through a web browser and provides an RStudio interface. You will be prompted to authenticate with your UC Davis credentials. This is the best way to work with R for people who are not comfortable accessing & editing in base R in a Shell application.

Note that mercury does not have SLURM, so it's best suited for interactive work and smaller computations. For large-scale computations requiring SLURM job scheduling, use shiva.ucdavis.edu instead.

When using RStudio Server, you can test your code interactively. However, do NOT use the RStudio Server's Terminal to install packages and configure your environment for SLURM-based clusters, as you will likely need to re-do it for every session/project. For SLURM clusters, use the command line approach described earlier.

17.5.3 Filepaths & configuration on the cluster

In most cases, you will want to test that the file paths work correctly on the cluster. You will likely need to add code to the configuration file in the project repository that specifies cluster-specific file paths. Here is an example:

```
# set cluster-specific file paths
if(Sys.getenv("LMOD_SYSHOST")!=""){

  cluster_path = paste0(Sys.getenv("HOME"), "/project-name/")

  data_path = paste0(cluster_path, "data/")
  results_path = paste0(cluster_path, "results/")
}
```

17.6 Storage & group storage access

17.6.1 Individual storage

There are multiple places to store your files on computing clusters. Each user has their own `$HOME` directory as well as a `$SCRATCH` directory. These are directories that can be accessed via the command line once you've logged in to the cluster:

```
cd $HOME
cd /home/users/USERNAME # Alternatively, use the full path

cd $SCRATCH
cd /scratch/users/USERNAME # Full path
```

You can also navigate to these using the File Explorer if available through a web interface.

`$HOME` typically has a volume quota (e.g., 15 GB). `$SCRATCH` typically has a larger volume quota (e.g., 100 TB), but files here may get deleted after a certain period of inactivity. Thus, use `$SCRATCH` for test files, exploratory analyses, and temporary storage. Use `$HOME` for long-term storage of important files and more finalized analyses.

Check with the UC Davis HPC documentation[6] for specific storage options and quotas.

17.6.2 Group storage

The lab may have shared `$GROUP_HOME` and `$GROUP_SCRATCH` directories to store files for collaborative use. These typically have larger quotas and may have different retention policies. You can access these via the command line or navigate to them using the File Explorer:

```
cd $GROUP_HOME
cd /home/groups/GROUPNAME

cd $GROUP_SCRATCH
cd /scratch/groups/GROUPNAME
```

However, saving files to group storage can be tricky. You can try using the scp command in the section "Moving files to the cluster" to see if you have permission to add files to group directories. Read the next section to ensure any directories you create have the right permissions.

17.6.3 Folder permissions

Generally, when we put folders in `$GROUP_HOME` or `$GROUP_SCRATCH`, it is so that we can collaborate on an analysis within the research group, so multiple people need to be able to access the folders. If you create a new folder in `$GROUP_HOME` or `$GROUP_SCRATCH`, please check the folder's permissions to ensure that other group members are able to access its contents. To check the permissions of a folder, navigate to the level above it, and enter `ls -l`. You will see output like this:

[6] https://hpc.ucdavis.edu/

```
drwxrwxrwx 2 jadebc jadebc  2204 Jun 17 13:12 myfolder
```

Please review this website[7] to learn how to interpret the code on the left side of this output. The website also tells you how to change folder permissions. In order to ensure that all users and group members are able to access a folder's contents, you can use the following command:

```
chmod ugo+rwx FOLDER_NAME
```

17.7 Running big jobs

Once your test scripts run successfully, you can submit an sbatch script for larger jobs. These are text files with a `.sh` suffix. Use a text editor like Sublime to create such a script. Documentation on sbatch options is available from Slurm Workload Manager ("Slurm Workload Manager: Sbatch Documentation," n.d.). Here is an example of an sbatch script with the following options:

- `job-name=run_inc`: Job name that will show up in the SLURM system
- `begin=now`: Requests to start the job as soon as the requested resources are available
- `dependency=singleton`: Jobs can begin after all previously launched jobs with the same name and user have ended.
- `mail-type=ALL`: Receive all types of email notification (e.g., when job starts, fails, ends)
- `cpus-per-task=16`: Request 16 processors per task. The default is one processor per task.
- `mem=64G`: Request 64 GB memory per node.
- `output=00-run_inc_log.out`: Create a log file called `00-run_inc_log.out` that contains information about the Slurm session
- `time=47:59:00`: Set maximum run time to 47 hours and 59 minutes. If you don't include this option, the cluster will automatically exit scripts after 2 hours of run time (default may vary by cluster).

The file `analysis.out` will contain the log file for the R script `analysis.R`.

```
#!/bin/bash

#SBATCH --job-name=run_inc
#SBATCH --begin=now
#SBATCH --dependency=singleton
#SBATCH --mail-type=ALL
#SBATCH --cpus-per-task=16
#SBATCH --mem=64G
#SBATCH --mem=64G
#SBATCH --output=00-run_inc_log.out
#SBATCH --time=47:59:00

cd $HOME/project-code-repo/2-analysis/
```

[7] https://www.chriswrites.com/how-to-change-file-permissions-using-the-terminal/

```
module purge

# load R version 4.0.2 (required for certain packages)
module load R/4.0.2

# load gcc, a C++ compiler (required for certain packages)
module load gcc/10

# load software required for spatial analyses in R
module load physics gdal
module load physics proj

R CMD BATCH --no-save analysis.R analysis.out
```

To submit this job, save the code in the chunk above in a script called `myjob.sh` and then enter the following command into terminal:

```
sbatch myjob.sh
```

To check on the status of your job, enter the following code into terminal:

```
squeue -u $USERNAME
```

18 Working with AI

AI-powered coding assistants[1] can dramatically accelerate and improve your work, but they require careful and responsible use. Lab members who use AI tools must adhere to the following guidelines:

18.1 Responsibility for validation

You are fully responsible for checking and validating all AI-generated code and content. AI tools can make mistakes, generate insecure code, produce incorrect logic, or suggest approaches that are inappropriate for our specific research context. Before using any AI-generated code:

- ☐ Carefully review the code to ensure you understand what it does
- ☐ Test the code thoroughly to verify it works as expected
- ☐ Verify that the logic is appropriate for your specific use case
- ☐ Check that the code follows our lab's coding standards and best practices
- ☐ Ensure the code does not introduce security vulnerabilities or data privacy issues

> ⚠ Warning
>
> Never blindly use AI-generated code without fully understanding it. If you don't completely understand what the AI has suggested, take the time to learn or ask a colleague for help.

18.2 Disclosure of AI use

You must clearly state whenever you have used AI tools in your work. This is essential for transparency and reproducibility. Specifically:

- In code comments, note when AI tools were used to generate or significantly modify code
- In commit messages, mention if AI tools assisted with the changes
- In manuscripts and reports, acknowledge AI tool usage in the methods or acknowledgments section
- In presentations, disclose AI assistance when relevant

Example code comment:

```
# The following function was generated with assistance from GitHub Copilot
# and has been reviewed and tested to ensure correctness
```

[1] https://en.wikipedia.org/wiki/AI-assisted_software_development

18.3 Attribution of sources

When using AI tools to generate content that borrows from or adapts existing sources, you must ensure proper attribution. AI tools sometimes paraphrase or adapt content from documentation, guides, or other resources without clearly indicating the original source. It is your responsibility to:

- Ask the AI tool to identify and properly cite sources when it borrows or adapts content
- Verify that any content the AI generates includes appropriate citations
- Add citations yourself if the AI fails to do so
- Follow appropriate attribution practices for the type of content (code comments, documentation, academic writing, etc.)

When instructing AI tools to create documentation or written content, explicitly request that they provide proper attribution for any borrowed or adapted material. For example: "Please quote from and paraphrase [source], with proper attribution" rather than simply asking it to summarize information on a topic.

18.4 Using AI for Journal Articles

When using AI tools to help develop journal articles and other academic writing, you must take special care to ensure transparency, maintain intellectual ownership, and avoid plagiarism. The following practices help achieve these goals.

18.4.0.1 Establish a Clear Track Record

Working with AI through GitHub creates valuable documentation of your contributions versus the AI's. This track record can be crucial if reviewers or editors question your use of AI tools.

GitHub Pull Requests and Issues provide:

- **Attribution clarity**: Each commit shows exactly who (you or `@copilot`) made which changes
- **Audit trail**: The full conversation history shows your instructions and the AI's responses
- **Intellectual ownership**: Your prompts and guidance demonstrate that the core ideas are yours
- **Transparency**: Reviewers can see that you actively supervised and validated all AI contributions

This transparency protects you if journal reviewers are skeptical about or opposed to AI use in research. You can point to the PR history to demonstrate that you maintained control and responsibility for the work.

18.4.0.2 Write Out Your Core Ideas in Prompts

Make your prompts explicit and detailed to establish that the ideas originate from you, not from the AI.

When requesting AI assistance with academic writing:

- State your research question, hypothesis, or argument clearly
- Outline the structure and key points you want to make
- Specify the evidence or data you want to include
- Describe the logic connecting your points
- Explain the interpretation or conclusions you want to draw

Example of a good prompt:

> I need help writing the discussion section for a study on social determinants of health. My core argument is that [your specific argument]. The key findings I want to discuss are: [list findings]. I want to interpret these findings as suggesting [your interpretation]. Please help me draft this section while preserving these core ideas and citing relevant literature.

This approach creates clear evidence that the intellectual content came from you, while the AI helped with expression, organization, and literature integration.

Avoid vague prompts like "write a discussion section about my results" that give the AI too much creative control and make it unclear whose ideas are being presented.

18.4.0.3 Request Explicit Source Attribution

Always instruct AI tools to identify and cite their sources to prevent unknowing plagiarism.

AI language models are trained on vast amounts of text, including published research. While they don't have direct access to their training data during generation, they may produce text that closely resembles or paraphrases existing work without providing attribution. This creates a plagiarism risk.

Best practices:

- Explicitly ask the AI to cite sources for any borrowed or adapted content
- Request that the AI indicate when it is drawing from specific works
- Verify that generated text includes proper citations
- Add citations yourself if the AI fails to provide them
- Cross-check AI-generated content against the cited sources to ensure accuracy

Example request:

> Please help me write this section, and explicitly cite any sources you draw from or adapt. If you're paraphrasing from specific papers, identify them clearly so I can verify the citations.

Remember that even with these precautions, you must still verify the accuracy and appropriateness of any AI-generated citations, as AI tools can sometimes generate plausible-but-incorrect references ("hallucinate" citations).

18.4.0.4 Example Workflow

The development of this documentation section demonstrates these practices. See Issue #112[2] and PR #145[3] for the complete development history, which shows:

1. **Core ideas stated clearly**: The issue description outlined specific concepts (transparency through GitHub, writing explicit prompts, requesting source attribution)
2. **Detailed instructions**: The guidance specified what content should be created and how it should be structured
3. **Transparent record**: The development history shows what was human-directed versus AI-generated
4. **Demonstrated accountability**: The pull request provides a full audit trail of all changes

When developing content for journal articles, a similar workflow creates documentation that demonstrates responsible AI use and intellectual ownership. You can point to your GitHub history to show reviewers that you directed the AI rather than simply accepting its output.

18.4.0.5 Additional Considerations

When using AI for academic writing, also remember to:

- **Disclose AI use**: Follow journal policies on acknowledging AI assistance (see Section 18.2)
- **Maintain responsibility**: You are accountable for all content, including AI-generated text
- **Verify accuracy**: Always fact-check AI-generated claims and citations
- **Preserve your voice**: Ensure the writing reflects your thinking and style, not just AI's patterns
- **Follow ethical guidelines**: Comply with your institution's and journals' policies on AI use

These practices help you benefit from AI assistance while maintaining the integrity, originality, and credibility of your academic work.

18.5 Coding Agents

We recommend working with **AI coding agents**[4] to help you code[5].

[2] https://github.com/UCD-SERG/lab-manual/issues/112
[3] https://github.com/UCD-SERG/lab-manual/pull/145
[4] https://github.com/features/copilot/agents
[5] https://en.wikipedia.org/wiki/AI-assisted_software_development

18.5.1 What are AI coding agents?

AI coding agents are AI agents[6] specialized for coding. They differ from other AI coding tools in important ways:

Compared to inline coding assistants (like traditional autocomplete), coding agents work autonomously rather than providing suggestions as you type. They can navigate entire codebases, execute commands, and complete multi-step tasks without constant human guidance.

Compared to AI chatbots (like ChatGPT or Claude), coding agents don't just generate code snippets in conversation—they actively interact with your development environment. While chatbots require you to copy code from a chat window and manually integrate it into your project, coding agents directly read your codebase, make changes to files, run tests and build commands, and create pull requests with their proposed changes. Chatbots are conversational assistants; coding agents are autonomous development tools.

Coding agents are autonomous software programs that can:

- **Understand and execute complex tasks**: Coding agents can interpret natural language instructions and break them down into actionable development tasks
- **Navigate and modify codebases**: They can read, understand, and edit multiple files across a repository to implement features or fix bugs
- **Run tools and commands**: Coding agents can execute build commands, run tests, use linters, and interact with development tools
- **Make decisions autonomously**: They can plan their approach, make technical decisions, and adjust their strategy based on results
- **Work iteratively**: Coding agents can test their changes, identify issues, and refine their solutions through multiple iterations
- **Create comprehensive solutions**: They can implement complete features that span multiple files, including code, tests, and documentation

Coding agents operate in isolated environments where they can safely experiment and validate changes before proposing them. This allows them to work more independently than inline coding assistants, which require step-by-step human direction. The agent workflow typically involves analyzing requirements, planning an implementation, making changes, testing those changes, and creating a pull request with the results.

While coding agents can handle substantial development tasks, they still require human oversight and review. The human developer remains responsible for:

- Reviewing the agent's work
- Ensuring the solution meets requirements
- Verifying code quality and security
- Making the final decision to merge changes

18.5.2 AI Agents and the Technological Singularity

The emergence of sophisticated AI agents[7] has prompted discussions about whether we are witnessing or approaching a technological singularity[8]. Understanding this concept helps

[6] https://en.wikipedia.org/wiki/AI_agent
[7] https://en.wikipedia.org/wiki/Intelligent_agent
[8] https://en.wikipedia.org/wiki/Technological_singularity

contextualize the rapid evolution of AI tools and our responsibility in using them.

18.5.2.1 What is the technological singularity?

The technological singularity is a hypothetical future point when technological growth becomes uncontrollable and irreversible, resulting in unforeseeable changes to human civilization. The concept, popularized by mathematician Vernor Vinge and futurist Ray Kurzweil, typically involves the creation of artificial superintelligence that recursively improves itself, leading to an intelligence explosion beyond human comprehension or control.

18.5.2.2 Do current AI agents represent the singularity?

No, current AI coding agents (as of early 2026) do not represent the technological singularity.

While modern AI agents demonstrate impressive capabilities, they remain fundamentally different from the singularity scenario in several critical ways:

- **Limited autonomy**: Today's AI agents operate within strict boundaries and require human oversight. They cannot recursively improve their own core architecture or develop capabilities beyond their training.
- **Narrow intelligence**: AI coding agents are specialized tools designed for specific tasks. They lack general intelligence, self-awareness, or the ability to operate outside their designed domain.
- **Human dependency**: These agents require human developers to: review their work, provide direction, validate correctness, and make final decisions about their outputs.
- **No recursive self-improvement**: Current AI agents cannot fundamentally redesign themselves or create more advanced versions of themselves autonomously. Any improvements to AI systems still require human researchers and engineers.
- **Controlled development environment**: AI coding agents work in sandboxed environments with explicit permissions and constraints. They cannot independently acquire resources, modify their own constraints, or operate without human authorization.

18.5.2.3 Why this matters for responsible AI use

Understanding that current AI agents are powerful but limited tools—not autonomous superintelligences—has important implications:

- **Maintain appropriate skepticism**: AI agent outputs require the same critical review as any other tool-generated code.
- **Preserve human decision-making**: The responsibility for code quality, security, and correctness remains with human developers.
- **Continue skill development**: Using AI agents should enhance rather than replace human expertise.

- **Stay vigilant**: While current agents don't represent a singularity, the rapid pace of AI development requires ongoing attention to emerging capabilities and risks.

The value of AI coding agents lies in their ability to accelerate human productivity and learning, not in replacing human judgment or expertise. They are sophisticated tools that augment human capabilities while remaining under human control and oversight.

18.5.2.4 Further reading

For thoughtful perspectives on AI consciousness and intelligence, see Douglas Hofstadter's reflections in "I Thought I Was in an AI Apocalypse. Then I Started Looking Closer."[9]

18.5.3 Relative Advantages of AI and Humans

AI coding agents and human coders have complementary strengths. Understanding these differences helps you decide when to delegate work to agents and when to handle tasks yourself.

18.5.3.1 Comparative Strengths: Humans vs. AI Agents

Table 18.1 summarizes the relative advantages of human coders and AI coding agents across different types of tasks:

Table 18.1: Relative advantages of humans and AI coding agents

Task Type	Humans	AI agents
Creative thinking	Humans excel at understanding context, handling ambiguous requirements, and thinking creatively about novel problems	AI agents struggle with ambiguous requirements and creative problem-solving in unfamiliar domains
Algorithmic thinking	Humans make mistakes when following repetitive instructions and may introduce inconsistencies	AI agents excel at executing well-defined, repetitive tasks with precision and consistency

---

Or, if you prefer a more visual representation:

[9] https://www.nytimes.com/2023/07/13/opinion/ai-chatgpt-consciousness-hofstadter.html

Table 18.2: Relative advantages of humans and Agents

	Humans	AI Agents
Creative thinking		
Algorithmic thinking		

This pattern mirrors the evolution of programming itself. Just as almost no one writes machine code anymore because higher-level languages and compilers handle those details, most developers will increasingly spend less time writing low-level code. Instead, you'll describe what the system needs to do as clearly as possible, and AI agents will handle many of the computational and coding details.

For most tasks, you won't need to step in and manipulate code yourself. However, you'll still need strong coding skills to:

- Supervise and validate AI-generated code

- Handle edge cases that agents struggle with
- Make creative decisions about architecture and design
- Understand when agent suggestions are incorrect or suboptimal

18.5.3.2 Future Developments: World Models

As AI technology advances, the distinction between these strengths may shift. Yann LeCun, 2019 Turing Award winner and AI researcher at Meta and NYU, advocates for developing "world models"—AI systems that understand and reason about the physical world, not just language patterns (LeCun 2022).

World models aim to give AI systems:

- **Persistent memory and reasoning**: Understanding that persists across interactions
- **Physical world understanding**: Reasoning about how things work in reality, not just in text
- **Better handling of ambiguity**: Using world knowledge to interpret unclear requirements

As these technologies mature, AI agents may become better at tasks requiring contextual understanding and creative problem-solving. This makes it even more important to develop strong supervision and validation skills now, so you can effectively work with increasingly capable AI systems.

18.5.4 How to Work with Coding Agents

GitHub Copilot coding agents can be used in several ways to automate development tasks:

18.5.4.1 Assigning Issues to Copilot

You can assign GitHub Issues directly to `@copilot` just like you would assign to a human collaborator:

1. **On GitHub.com**: Navigate to an issue and assign it to Copilot in the assignees section
2. **In VS Code**: In the GitHub Pull Requests or Issues view, right-click an issue and select "Assign to Copilot"
3. **From Copilot Chat**: Delegate tasks to Copilot directly from the chat interface in supported editors

18.5.4.2 The Agent Workflow

Once assigned an issue, the coding agent follows an autonomous workflow:

1. **Analysis**: Reviews the issue description, related discussions, repository instructions, and codebase context
2. **Planning**: Determines what changes are needed and creates a work plan

3. **Development**: Works in an isolated GitHub Actions environment, modifies code, runs tests and linters, and validates changes
4. **Pull Request Creation**: Creates a draft pull request with implemented changes, audit logs, and a summary of modifications
5. **Review and Iteration**: You review the PR and can request changes; the agent will iterate based on your feedback

18.5.4.3 Example: This Document

This very section you're reading was created through the coding agent workflow:

1. **Issue created**: Issue #42[10] requested adding discussion about benefits and hazards of coding agents, including a Matrix film connection and best practices
2. **Agent assigned**: The issue was assigned to `@copilot`
3. **Work completed**: The agent analyzed the requirements, reviewed the repository structure, and implemented the changes across multiple files
4. **Pull request**: PR #50[11] was created with comprehensive content about coding agents, including this "How to Work with Coding Agents" section, benefits and hazards discussion, best practices, and firewall configuration details
5. **Iteration**: The PR received feedback comments requesting additional links, improved wording, and this example section—all of which the agent addressed through follow-up commits

This demonstrates the full lifecycle of working with a coding agent on a real documentation task.

18.5.4.4 Collaborating with Coding Agents

Between iterations of asking coding agents to extend a PR, human collaborators can also push changes directly to the PR branch. This allows for a collaborative workflow where both humans and agents contribute:

- **Human contributions**: You can make quick fixes, add content, or refine the agent's work by pushing commits to the same branch
- **Agent iterations**: After your changes, you can ask the agent to continue working on additional requirements

Important: Try to avoid pushing changes while the coding agent is actively working. Simultaneous edits can produce conflicting diffs that:

- Need to be manually resolved
- May confuse both human and AI collaborators
- Could result in lost work or merge conflicts

[10] https://github.com/UCD-SERG/lab-manual/issues/42
[11] https://github.com/UCD-SERG/lab-manual/pull/50

Best practice: Wait for the agent to complete its current iteration (indicated by the PR being updated) before pushing your own changes to the branch. Then assign new work to the agent for the next iteration.

18.5.4.5 Directly Prompting for Pull Requests

You can also prompt Copilot to create pull requests without first creating an issue:

- Use Copilot Chat in your editor to describe the changes you want
- The agent will analyze your request and create a pull request
- This is useful for quick fixes or well-defined tasks

18.5.4.6 Important Safeguards

- **Human approval required**: Coding agents cannot merge their own changes
- **Branch restrictions**: Agents can only push to their own branches (e.g., `copilot/*`)
- **Full transparency**: All agent actions are logged and visible in the PR

18.5.4.7 Workflow Approval Requirements

When GitHub Copilot creates or updates a pull request, it cannot automatically trigger GitHub Actions workflows. **You must manually approve each workflow run** by clicking the approval button in the Actions tab or on the PR.

This manual approval requirement is a security measure that prevents potentially malicious or unintended code execution. Because Copilot can modify any file in the repository—including workflow files themselves or scripts called by workflows—allowing automatic workflow execution could create security vulnerabilities.

Key points:

- **No automatic approval**: There is currently no way to bypass manual workflow approval for Copilot PRs, even if you are the repository owner
- **Security reasoning**: Copilot could modify workflow files (`.github/workflows/*.yml`) or scripts they execute, potentially injecting malicious code
- **Impact on workflow**: This means you need to actively monitor and approve workflow runs as Copilot iterates on your issue, which can slow down the development cycle

Workaround considerations:

Some users have discussed using Personal Access Tokens (PATs) to allow Copilot to trigger workflows on your behalf, but this approach has security implications and should be carefully evaluated before implementation.

For more details and community discussion about this limitation, see:

- GitHub Community Discussion #162826[12]: Discussion about workflow approval requirements
- GitHub Community Discussion #183966[13]: Product feedback on this topic

[12] https://github.com/orgs/community/discussions/162826
[13] https://github.com/orgs/community/discussions/183966

For detailed instructions, see GitHub Copilot coding agent documentation[14].

18.5.5 Useful Prompt Formats

When working with coding agents, using clear and specific prompts helps achieve better results. Here are some useful prompt formats that you can use when requesting assistance from coding agents:

18.5.5.1 Common Task Patterns

Tidying up code:

- "tidy up [file, function, module, whole project]"
- Useful for improving code organization, consistency, and readability
- Example: "tidy up the data processing module"

Addressing failing workflows:

- "address failing workflows"
- Helps fix continuous integration (CI) failures, build errors, or test failures
- Example: "address failing workflows in the GitHub Actions pipeline"

Decomposing code:

- "decompose [function, quarto-file, etc]"
- Breaks down large or complex code into smaller, more manageable pieces
- Example: "decompose this analysis function into separate helper functions"

Updating content:

- "update [links, content, etc]"
- Refreshes outdated information, fixes broken links, or modernizes code
- Example: "update all package URLs in the documentation"

Expanding documentation:

- "expand [a section in a document]"
- Adds more detail, examples, or explanation to existing content
- Example: "expand the section on data validation with practical examples"

Condensing content:

- "condense [a section in a document]"
- Reduces verbosity while preserving essential information
- Example: "condense the installation instructions to be more concise"

Clarifying content:

- "clarify [a section in a document]"
- Improves clarity, removes ambiguity, or simplifies complex explanations
- Example: "clarify the explanation of the analysis workflow"

[14] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent

18.5.5.2 Tips for Effective Prompts

- **Be specific**: Include file names, function names, or specific sections when possible
- **Provide context**: Explain what you want to achieve and why
- **Set boundaries**: Specify what should or shouldn't change
- **Request validation**: Ask the agent to test or verify its changes when appropriate

18.5.6 Addressing Failing GitHub Actions Workflows

When GitHub Actions workflows fail, you can use Copilot to help diagnose and fix the issues. However, it's important to use the right prompts depending on whether the problem is in your code or in the workflow configuration itself.

18.5.6.1 Scenario 1: Code Issues Found by Workflows (Most Common)

When to use: The workflow is functioning correctly, but it's detecting problems in your code (e.g., failing tests, linting errors, build failures).

What you want: Fix the code issues without modifying the workflow files themselves.

Recommended prompts:

- "fix the code issues found by the failing workflows"
- "address the linting errors reported in the GitHub Actions checks"
- "fix the test failures in the CI pipeline"
- "resolve the build errors shown in the workflow logs"

Example: If your R package has failing tests detected by `usethis::use_github_action("check-standard")`, you want Copilot to fix the test failures in your R code, not modify the workflow YAML file.

Why this matters: These prompts make it clear that you want code changes, not workflow changes. This helps prevent the agent from unnecessarily modifying your carefully-configured CI/CD pipeline.

18.5.6.2 Scenario 2: Issues with Workflow Files Themselves

When to use: The workflow configuration itself has problems (e.g., syntax errors in YAML, incorrect job definitions, outdated actions).

What you want: Fix the workflow files, but with extreme caution due to security implications.

Recommended prompts:

- "fix the syntax error in the GitHub Actions workflow file at line X"
- "update the workflow to use the latest version of action Y"
- "correct the job configuration in .github/workflows/check-standard.yaml"

Important considerations:

> ⚠ Warning
>
> **Security Warning**
> Workflow files have access to repository secrets and can execute arbitrary code. Before accepting any changes to workflow files:
>
> 1. **Review every line** of the proposed changes
> 2. **Verify** the changes only address the specific issue
> 3. **Check** that no new secret access or command execution has been added
> 4. **Test** in a safe environment if possible
>
> See Section 18.5.8 for more details on workflow file security.

When to do it yourself: Workflow syntax errors and configuration issues are often faster to fix manually than with Copilot, especially if you're familiar with GitHub Actions. See Section 18.5.12 for more guidance.

18.5.6.3 Scenario 3: Uncertain Which Scenario Applies

When to use: You're not sure whether the failure is due to code issues or workflow configuration problems.

Recommended approach:

1. **First, examine the workflow logs**:
   - Look at the error messages in the GitHub Actions tab
   - Identify whether the error is in your code or the workflow itself
   - Common code issues: test failures, linting errors, compilation errors
   - Common workflow issues: YAML syntax errors, missing actions, permission errors
2. **Use a diagnostic prompt**:
   - "examine the failing workflow logs and identify whether the issue is in the code or the workflow configuration"
   - "diagnose the root cause of the workflow failure"
3. **Then use the appropriate scenario above**: Once you understand the issue, use the specific prompts from Scenario 1 or 2.

Example workflow:

```
1. Prompt: "examine the failing workflow logs and identify the issue"
2. Copilot responds: "The workflow is failing because of linting errors
   in src/analysis.R"
3. Prompt: "fix the linting errors in src/analysis.R"
```

18.5.6.4 Additional Resources

- See Chapter 7 for setting up GitHub Actions workflows
- See Section 18.5.8 and Section 18.5.7 for security considerations with workflow files

- See Section 18.5.12 for guidance on when to use Copilot vs. fixing issues yourself
- See the GitHub Actions documentation[15] for workflow syntax and troubleshooting

18.5.7 Benefits and Hazards

Coding agents are powerful programs that can work autonomously. They create pull requests that propose changes to the code in our repositories, potentially including their own configuration files and our automated workflows. They can work powerfully on our behalf, but they require careful oversight and control to ensure they serve our interests and that we understand the consequences of their actions.

Coding agents offer several advantages:

- **Built-in transparency**: Coding agents create a clear record of their role in your work through commit history and code suggestions
- **Context-aware suggestions**: Coding agents understand your codebase and can make contextually relevant suggestions
- **Integration with version control**: Using coding agents within GitHub ensures that AI-assisted changes are tracked alongside all other code changes
- **Interactive workflow**: Coding agents' interactive nature encourages you to review and modify suggestions rather than blindly accepting them
- **Accelerated development**: Coding agents can help you write boilerplate code, refactor existing code, and implement common patterns more quickly
- **Learning opportunities**: Coding agents can suggest approaches or techniques you may not have considered, helping you expand your coding knowledge

However, coding agents also come with significant hazards:

- **Over-reliance**: Depending too heavily on coding agents can atrophy your coding skills and understanding
- **Subtle bugs**: AI-generated code may contain logic errors that are not immediately obvious
- **Security vulnerabilities**: Coding agents may introduce insecure patterns or fail to follow security best practices
- **Inappropriate solutions**: AI may suggest solutions that work but are not optimal for your specific research context or constraints
- **Hidden biases**: Coding agents may perpetuate coding patterns or approaches that reflect biases in their training data
- **False confidence**: Well-formatted, professional-looking code from AI can mask underlying problems and reduce critical review
- **Workflow manipulation risks**: Coding agents that modify CI/CD workflows (`.github/workflows/*.yml`) or setup configurations can inadvertently or maliciously compromise repository security, expose secrets, or execute harmful commands

[15] https://docs.github.com/en/actions

18.5.7.1 Further reading/viewing

- *I Robot* (Asimov 1950)
- *Dune* (Herbert 1965)
- "2001: A Space Odyssey" (1968)
- "Terminator 3: Rise of the Machines" (2003)
- "The Matrix" (1999)
- "Blade Runner" (1982)
- "WarGames" (1983)
- *Battlestar Galactica* (2004) ("Battlestar Galactica" 2004)
- *Ender's Game* (Card 1985)
- "The Humans are Dead" (Flight of the Conchords 2007)

Figure 18.1: Agents[16]

18.5.8 Best Practices for Safe and Successful Use

To work with coding agents safely and successfully:

1. **Maintain active supervision**: Never assume AI-generated code is correct. Review every line critically.
2. **Understand before accepting**: If you don't understand what the code does, don't use it. Take time to learn or ask a colleague.
3. **Test thoroughly**: AI-generated code must be tested as rigorously as code you write yourself. Don't skip testing because "the AI wrote it."
4. **Start small**: Begin with small, well-defined tasks to build confidence and understanding of the agent's capabilities and limitations.
5. **Verify logic and assumptions**: Check that the AI hasn't made incorrect assumptions about your data, requirements, or scientific context.
6. **Review for security**: Explicitly check for security issues, especially when handling sensitive data or user input.
7. **Iterate and refine**: Use coding agents as a starting point, not an endpoint. Refine and improve the generated code.

8. **Maintain coding practice**: Regularly write code yourself to maintain and develop your skills. Don't let the agent do everything.

> ⚠ Critical: Exercise Extreme Caution with Workflow Files
>
> Be especially careful when allowing coding agents to edit GitHub Actions workflows or CI/CD configurations. These files control automated processes that can:
>
> - Access secrets and credentials
> - Deploy code to production
> - Execute arbitrary commands in your repository
>
> **Never** allow a coding agent to edit workflow files (especially `.github/workflows/*.yml` or `copilot-setup-steps.yml`) without thorough manual review. Before approving any workflow run, always check if the workflow files themselves have been modified. Malicious or erroneous changes to workflows can compromise your entire repository and its secrets.

When using coding agents, work interactively with the AI suggestions: review, modify, and test them rather than accepting them wholesale. This interactive approach helps ensure code quality and deepens your understanding of the code.

Remember: AI tools are assistants, not replacements for your expertise and judgment. The quality and correctness of your work remains your responsibility.

18.5.9 Firewall and Network Configuration

Coding agents require specific network access to function properly. If a coding agent is running behind a corporate firewall or on a restricted network, you may need to configure allowlists to enable coding agent functionality.

18.5.9.1 Built-in Agent Firewall

Coding agents run in a GitHub Actions environment with a built-in firewall that limits internet access by default. This firewall helps protect against:

- Data exfiltration
- Accidental leaks of sensitive information
- Execution of malicious instructions

By default, the agent's firewall allows access to:

- Common OS package repositories (Debian, Ubuntu, Red Hat, etc.)
- Popular container registries (Docker Hub, Azure Container Registry, AWS ECR, etc.)
- Language-specific package registries (npm, PyPI, Maven, RubyGems, etc.)
- Common certificate authorities for SSL validation

For the complete list of allowed hosts, see the Copilot allowlist reference[17].

[17] https://docs.github.com/en/copilot/reference/copilot-allowlist-reference

18.5.9.2 Customizing Agent Firewall Settings

In your repository's "Coding agent" settings page, you can:

- Add custom hosts to the allowlist (for internal dependencies or additional registries)
- Opt out of the default recommended allowlist for stricter security
- Disable the firewall entirely (not recommended)

If a coding agent's request is blocked by the firewall, a warning will be added to the pull request or comment, detailing the blocked address and the command that triggered it.

For more information, see Customizing or disabling the firewall for GitHub Copilot coding agent[18].

18.5.9.3 Recommended URLs for Data Science Repositories

For data science and R-focused repositories, we recommend adding the following URLs to your Copilot allowlist. These sites are safe, reputable sources of documentation and packages that coding agents may need to access:

R Package Documentation and Ecosystems:

- `tidyverse.org` - `{tidyverse}` package documentation and learning resources
- `r-lib.org` - Core R infrastructure packages (`{devtools}`, `{testthat}`, `{usethis}`, etc.)
- `ggplot2.tidyverse.org` - `{ggplot2}` visualization package
- `dplyr.tidyverse.org` - `{dplyr}` data manipulation package
- `tidyr.tidyverse.org` - `{tidyr}` data tidying package
- `purrr.tidyverse.org` - `{purrr}` functional programming package
- `readr.tidyverse.org` - `{readr}` data reading package
- `stringr.tidyverse.org` - `{stringr}` string manipulation package
- `forcats.tidyverse.org` - `{forcats}` categorical data package

R Package Repositories:

- `cran.r-project.org` - The Comprehensive R Archive Network
- `cloud.r-project.org` - CRAN mirror (cloud-based)
- `docs.ropensci.org` - rOpenSci package documentation (e.g., `{targets}`)
- `rdatatable.gitlab.io` - `{data.table}` package documentation
- `rstudio.github.io` - RStudio-maintained packages (e.g., `{renv}`)

Code Style and Quality Tools:

- `styler.r-lib.org` - `{styler}` code formatting package
- `lintr.r-lib.org` - `{lintr}` code linting package
- `roxygen2.r-lib.org` - `{roxygen2}` documentation package
- `style.tidyverse.org` - Tidyverse style guide

General Documentation and Reference:

- `en.wikipedia.org` - General reference and technical documentation

[18] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-firewall

- `r-project.org` - Official R project website
- `quarto.org` - Quarto publishing system documentation
- `pandoc.org` - Pandoc document converter documentation

GitHub Organizations (for package repositories):

- `github.com/tidyverse/*` - Tidyverse package source code
- `github.com/r-lib/*` - R-lib package source code
- `github.com/rstudio/*` - RStudio package source code
- `github.com/ropensci/*` - rOpenSci package source code

> **When to Add These URLs**
>
> Add these URLs to your repository's allowlist if:
>
> - Coding agents report blocked access to these sites
> - You're working on R or data science projects that use these packages
> - You want agents to access current documentation during code generation
>
> You can add URLs selectively based on your project's specific dependencies rather than adding all URLs at once.

> **Safety of These URLs**
>
> All URLs listed here are:
>
> - Maintained by reputable organizations (Tidyverse, RStudio/Posit, R Core Team, rOpenSci)
> - Widely used in the R community
> - Focused on documentation and package distribution
> - Safe for coding agents to access
>
> These sites do not host user-generated content or allow arbitrary code execution, making them appropriate for inclusion in your allowlist.

18.5.10 Configuring GitHub Copilot Settings

GitHub Copilot offers numerous configuration options that control how the AI assistant integrates into your development workflow. This section explains the key settings visible in your GitHub account preferences and provides guidance on which options to enable based on your use case.

18.5.10.1 Model Selection Options

GitHub Copilot provides access to multiple AI models, each with different capabilities and performance characteristics. The available models as of early 2026 include:

Anthropic Claude Models:

- **Claude Opus 4.1**: Most capable model for complex reasoning and analysis

  - *Pros*: Excellent at understanding nuanced requirements, handling complex codebases, superior code quality
  - *Cons*: Slower response times, may be overkill for simple tasks, limited availability (select option required)
  - *When to use*: Complex refactoring, architectural decisions, thorough code reviews

- **Claude Opus 4.5**: Latest version with enhanced capabilities
  - *Pros*: State-of-the-art performance, improved reasoning over 4.1
  - *Cons*: Similar trade-offs to Opus 4.1, requires selection
  - *When to use*: Most demanding tasks requiring cutting-edge capabilities

- **Claude Sonnet 4**: Balanced model optimizing capability and speed
  - *Pros*: Fast responses, strong performance, good default choice
  - *Cons*: Slightly less capable than Opus models for very complex tasks
  - *When to use*: General development work, most coding tasks

- **Claude Sonnet 4.5**: Enhanced version of Sonnet
  - *Pros*: Improved over Sonnet 4 while maintaining speed
  - *Cons*: Still not as powerful as Opus for extremely complex scenarios
  - *When to use*: Most daily development tasks

- **Claude Haiku 4.5**: Fast, efficient model for simpler tasks
  - *Pros*: Very fast responses, cost-effective, good for quick questions
  - *Cons*: Less capable for complex reasoning or large codebases
  - *When to use*: Simple completions, quick questions, repetitive tasks

OpenAI GPT Models:

- **GPT-5.2-Codex**: Specialized for code generation
  - *Pros*: Strong code completion, good at common patterns
  - *Cons*: May hallucinate package names or APIs
  - *When to use*: Code completion, common coding patterns

- **GPT-5**: Latest general-purpose model
  - *Pros*: Broad knowledge, good general performance
  - *Cons*: Not specifically optimized for code
  - *When to use*: Mixed tasks involving code and documentation

- **GPT-5-Codex** (various versions including Mini and Max):
  - *Pros*: Specialized variants for different use cases
  - *Cons*: Fragmented options can be confusing
  - *When to use*: Specific scenarios where variant optimizations matter

Google Gemini Models:

- **Gemini 2.5 Pro**: High-capability model
  - *Pros*: Strong multimodal capabilities, good at understanding context
  - *Cons*: Less proven in coding scenarios than Claude or GPT
  - *When to use*: Tasks involving images or complex context

- **Gemini 3 Pro/Flash** (Preview): Latest generation

  - *Pros*: Cutting-edge capabilities, flash variant offers speed
  - *Cons*: Preview status means less stable, limited track record
  - *When to use*: Experimental workflows, evaluation of new capabilities

Lab Recommendation: For most lab work, enable **Claude Sonnet 4.5** as your default model. It provides excellent balance of capability and speed. Consider switching to **Claude Opus 4.5** for complex architectural decisions or difficult debugging sessions. Keep **Claude Haiku 4.5** enabled for quick inline completions.

18.5.10.2 Feature Settings

These settings control where and how Copilot integrates into your development environment:

Editor preview features:

- *What it does*: Enables previews of experimental features in your editor
- *Pros*: Access to latest capabilities before general release
- *Cons*: May have bugs or unstable behavior
- *Recommendation*: **Enable** if you're comfortable troubleshooting issues and want cutting-edge features

Copilot Chat in GitHub.com:

- *What it does*: Enables Copilot chat interface on GitHub.com
- *Pros*: Quick access to Copilot without opening an editor, useful for reviewing PRs
- *Cons*: Only available with paid license
- *Recommendation*: **Enable** (included in GitHub Copilot subscription)

Copilot CLI:

- *What it does*: GitHub Copilot for assistance in terminal
- *Pros*: AI help for command-line operations, shell commands, and git operations
- *Cons*: Requires separate installation and setup
- *Recommendation*: **Enable** and install via `gh extension install github/gh-copilot`

Copilot in GitHub Desktop:

- *What it does*: Enables Copilot in GitHub Desktop app
- *Pros*: AI assistance in GUI git client
- *Cons*: Limited compared to editor integration
- *Recommendation*: **Enable** if you use GitHub Desktop

Copilot Chat in the IDE:

- *What it does*: Enables chat interface in your code editor
- *Pros*: Context-aware help, refactoring assistance, code explanation
- *Cons*: Can be distracting if overused
- *Recommendation*: **Enable** (essential feature)

Copilot Chat in GitHub Mobile:

- *What it does*: Enables Copilot chat in mobile app

- *Pros*: Quick access on mobile devices
- *Cons*: Limited by mobile interface
- *Recommendation*: **Enable** for convenience

Copilot can search the web:

- *What it does*: Allows Copilot to search internet for up-to-date information
- *Pros*: Access to current documentation, recent library changes, latest best practices
- *Cons*: May introduce latency, results depend on search quality
- *Recommendation*: **Enable** for access to current information

18.5.10.3 Advanced Settings

Dashboard Entry Point:

- *What it does*: Allows instant chatting when landing on GitHub.com
- *Pros*: Quick access to Copilot without navigating menus
- *Cons*: None significant
- *Recommendation*: **Enable** for convenience

Copilot code review:

- *What it does*: Use Copilot to review your code and generate pull request summaries
- *Pros*: Automated code review suggestions, PR summary generation, catches common issues
- *Cons*: May generate false positives, shouldn't replace human review
- *Recommendation*: **Enable** (major productivity boost)

Automatic Copilot code review:

- *What it does*: Automatically reviews all pull requests you create
- *Pros*: Catches issues early without manual triggering
- *Cons*: May be noisy on simple PRs, uses API quota
- *Recommendation*: **Disable** initially; enable only after you're comfortable with code review quality

Copilot coding agent:

- *What it does*: Delegate tasks to Copilot coding agent in repositories where it is enabled
- *Pros*: Autonomous multi-file edits, can execute complex refactoring, runs tests and fixes issues
- *Cons*: Requires careful oversight, can make unwanted changes if instructions unclear
- *Recommendation*: **Enable** (see Section 18.5.8 for safe usage guidelines)

Copilot Memory (Preview):

- *What it does*: Remember repository context across Copilot agent interactions
- *Pros*: Better context awareness, learns repository patterns and conventions
- *Cons*: Preview feature governed by pre-release terms, potential privacy implications
- *Recommendation*: **Enable** to help Copilot learn your codebase patterns

MCP servers in Copilot:

- *What it does*: Connect MCP servers to Copilot in all editors and Coding Agent

- *Pros*: Extend Copilot with custom tools and integrations
- *Cons*: Requires MCP server setup and maintenance
- *Recommendation*: **Enable** if you have MCP servers configured; otherwise this setting has no effect

Copilot-generated commit messages:

- *What it does*: Allow Copilot to suggest commit messages when you make changes
- *Pros*: Saves time, generates descriptive messages based on code changes
- *Cons*: May miss important context, still requires review
- *Recommendation*: **Enable** but always review and edit suggested messages

Copilot Spaces:

- *What it does*: View and create Copilot Spaces (collaborative AI environments)
- *Pros*: Share AI context with team members
- *Cons*: Additional complexity for individual work
- *Recommendation*: **Enable** for team collaboration features

Copilot Spaces Individual Access:

- *What it does*: Create individually owned Copilot Spaces
- *Pros*: Personal AI workspaces for complex projects
- *Cons*: May fragment your workflow
- *Recommendation*: **Enable** for flexibility

Copilot Spaces Individual Sharing:

- *What it does*: Share individually owned Copilot Spaces
- *Pros*: Collaborate while maintaining ownership
- *Cons*: None significant
- *Recommendation*: **Enable** for sharing capability

18.5.10.4 Summary of Recommended Settings

For lab members, we recommend the following configuration:

Enable these features:

- All Copilot Chat options (GitHub.com, CLI, IDE, Mobile)
- Web search capability
- Dashboard Entry Point
- Copilot code review (but not automatic review initially)
- Copilot coding agent
- Copilot Memory
- MCP servers (if configured)
- Copilot-generated commit messages
- All Copilot Spaces options

Model selection:

- Default: Claude Sonnet 4.5
- Complex tasks: Claude Opus 4.5
- Quick completions: Claude Haiku 4.5

Enable with caution:

- Editor preview features (only if comfortable with potential instability)
- Automatic Copilot code review (wait until familiar with review quality)

Following these guidelines will help establish an effective Copilot configuration. The key is to enable features that add value to your workflow while maintaining awareness that AI assistance requires validation (see Section 18.5.8).

18.5.11 Configuring the Agent Environment

The `.github/workflows/copilot-setup-steps.yml` file allows you to customize the development environment in which the GitHub Copilot coding agent operates. This file preinstalls tools and dependencies so that Copilot can build, test, and lint your code more reliably.

18.5.11.1 Why Configure the Environment?

While Copilot can discover and install dependencies through trial and error, this can be slow and unreliable. Additionally, Copilot may be unable to access private dependencies. Preconfiguring the environment ensures:

- Faster agent startup and execution
- More reliable builds and tests
- Access to private or authenticated dependencies
- Consistent development environment across all agent sessions

18.5.11.2 File Location and Structure

The workflow file must be located at `.github/workflows/copilot-setup-steps.yml` in your repository's **default branch**. It follows GitHub Actions workflow syntax but must contain a single job named `copilot-setup-steps`.

18.5.11.3 Basic Configuration Example

See Appendix: Copilot Setup Steps File for the configuration used in this repository (adapted for R and Quarto projects).

18.5.11.4 Using `actions/checkout`

The `actions/checkout`[19] action is used to check out your repository code so that the workflow can access it. While Copilot will automatically check out your repository if you don't include this step, **explicitly including it is necessary** when your setup steps need to access repository files.

Why explicitly include checkout?

Many dependency installation steps require access to repository files:

[19] https://github.com/actions/checkout

- `r-lib/actions/setup-renv@v2` needs `renv.lock` to install R package dependencies
- `r-lib/actions/setup-r-dependencies@v2` needs `DESCRIPTION` to install R package dependencies
- `npm ci` needs `package-lock.json` to install Node.js dependencies
- `pip install -r requirements.txt` needs the requirements file

Without an explicit checkout step, these dependency installation commands will fail because the necessary files won't be available yet.

Basic checkout:

```
- name: Checkout code
  uses: actions/checkout@v4
```

Important: The Copilot coding agent overrides any `fetch-depth` value you set in the checkout step. According to GitHub's official documentation[20], this override happens "to allow the agent to rollback commits upon request, while mitigating security risks." The agent dynamically determines the appropriate fetch depth based on the pull request context.

While you cannot control the fetch depth used by Copilot, the agent still has access to sufficient git history to perform its work effectively, including comparing changes and understanding the context of your pull request.

18.5.11.5 Configurable Options

You can customize only these specific settings in the `copilot-setup-steps` job:

- `steps`: Setup commands and actions to run
- `permissions`: Access permissions (typically `contents: read`)
- `runs-on`: Runner type (Ubuntu x64 Linux only)
- `services`: Database or service containers
- `snapshot`: Save environment state
- `timeout-minutes`: Maximum 59 minutes

All other workflow settings are ignored by Copilot.

18.5.11.6 Common Setup Tasks

For Node.js/TypeScript projects:

```
- name: Set up Node.js
  uses: actions/setup-node@v4
  with:
    node-version: "20"
    cache: "npm"

- name: Install dependencies
  run: npm ci
```

[20] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment

For Python projects:

```yaml
- name: Set up Python
  uses: actions/setup-python@v5
  with:
    python-version: "3.11"

- name: Install dependencies
  run: pip install -r requirements.txt
```

For R projects:

```yaml
- name: Set up R
  uses: r-lib/actions/setup-r@v2
  with:
    r-version: 'release'

- name: Install R dependencies
  uses: r-lib/actions/setup-renv@v2
```

18.5.11.7 Environment Variables and Secrets

To set environment variables for Copilot:

1. Navigate to your repository's **Settings**
2. Go to **Environments**
3. Select or create the `copilot` environment
4. Add environment variables or secrets as needed

Use secrets for sensitive values like API keys or passwords.

18.5.11.8 Testing Your Configuration

The workflow runs automatically when you modify `copilot-setup-steps.yml`, allowing you to validate changes in pull requests. You can also manually trigger the workflow from the repository's **Actions** tab.

Setup logs appear in the agent session logs when Copilot starts working. If a step fails, Copilot will skip remaining steps and begin working with the current environment state.

18.5.11.9 Advanced Configuration

Larger runners: For projects requiring more resources, you can use larger GitHub-hosted runners:

```yaml
jobs:
  copilot-setup-steps:
    runs-on: ubuntu-4-core
```

Self-hosted runners (ARC): For access to internal resources or private registries, use Actions Runner Controller (ARC) self-hosted runners:

```
jobs:
  copilot-setup-steps:
    runs-on: arc-scale-set-name
```

Note: When using self-hosted runners, you must disable Copilot's integrated firewall in repository settings and configure appropriate network security controls.

Git Large File Storage (LFS): If your repository uses Git LFS:

```
- uses: actions/checkout@v4
  with:
    lfs: true
```

18.5.11.10 Further Reading

For complete details, see Customizing the development environment for GitHub Copilot coding agent[21].

18.5.12 When to use a coding agent

Coding agent sessions are currently[22] considered "premium requests", which are limited resources; see https://github.com/features/copilot/plans for details. So, use coding agents sparingly. Use them for complex changes that would be difficult or time-consuming for you to complete by hand. Coding agents also take time to get configured for work, every time you make a request. See https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment#preinstalling-tools-or-dependencies-in-copilots-environment for ways to reduce that startup time, but it will never be 0. If you can complete the task faster than the coding agent can, you should probably do it yourself. For example, when you have errors in the spell-check or lint workflows, you can often fix them faster than Copilot can. Similarly, when reviewing Copilot's PRs, you can often make direct changes to the branch faster than you could write clear review comments and get Copilot to address them.

Also, the less we practice, the weaker our skills get, and the harder it is for us to supervise the agents and make sure they are actually doing what we want them to do, the way we want them to do it. You should exercise your own coding skills regularly, just like you would for any other skill you want to maintain.

18.5.13 Editing with `.docx` files

GitHub Copilot coding agents can read Microsoft Word (`.docx`) files, including tracked changes and comments. This enables a hybrid editing workflow where:

1. Lab members can export Quarto content to Word format for review

[21] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment

[22] 2026-01-10

2. Reviewers can make edits, add tracked changes, and insert comments in Word
3. Coding agents can read the `.docx` file and translate the edits back to Quarto format

When using this workflow, make sure to explicitly instruct the coding agent to:

- Examine and apply all tracked changes in the `.docx` file
- Read and address all comments in the `.docx` file
- Translate edits from Word formatting to appropriate Quarto/markdown syntax

This approach makes it easier for collaborators who are more comfortable with Word to contribute to the lab manual while maintaining the source files in Quarto format.

18.5.13.1 Known Issue: "Document 1" Warning in Word

When opening DOCX files generated by Quarto (including this lab manual), Microsoft Word may display a warning message and open the file with the title "Document 1" instead of the actual filename. Word may also require you to save the file before you can add comments or track changes.

This is a known limitation with how Quarto generates DOCX files. The issue is being tracked in the Quarto project:

- Quarto CLI Issue #6357[23]
- Quarto Discussion #6544[24]
- Quarto CLI Issue #10587[25]

Workaround: If you are the author generating the DOCX file from Quarto, follow these steps before sharing with collaborators:

1. Open the generated DOCX file in Microsoft Word
2. Immediately save the file (File → Save, or Ctrl+S/Cmd+S)
3. Close and re-open the file to verify it no longer shows "Document 1"
4. Share this saved version with collaborators

This one-time step ensures that when collaborators open the file, they won't see the "Document 1" warning and can immediately add comments and track changes without issues.

18.5.14 Copilot Instructions for this Repository

The `.github/copilot-instructions.md` file in this repository contains specific instructions and guidelines for GitHub Copilot coding agents when working with the lab manual. This file helps ensure that AI-generated contributions follow the lab's formatting standards, coding conventions, and documentation practices.

The copilot instructions file specifies:

- Markdown and Quarto formatting rules (e.g., blank lines before lists, line breaks in prose)
- R code style guidelines (e.g., using native pipe `|>`, following tidyverse style)

[23] https://github.com/quarto-dev/quarto-cli/issues/6357
[24] https://github.com/orgs/quarto-dev/discussions/6544
[25] https://github.com/quarto-dev/quarto-cli/issues/10587

- File organization patterns (e.g., using Quarto includes for modular content)
- How to work with DOCX files for hybrid editing workflows
- Repository-specific best practices

By having these instructions in `.github/copilot-instructions.md`, we ensure that coding agents produce consistent, high-quality contributions that align with the lab's established practices. This reduces the review burden and helps maintain consistency across all contributions to the lab manual, whether made by humans or AI assistants.

See Appendix: Copilot Instructions File for the complete file.

18.5.15 Using Copilot Review Before Human Review

Before requesting review from other humans, **always have Copilot review your pull request first**—even if Copilot created the PR itself. AI review provides fast, thorough feedback that helps catch issues before involving human reviewers, saving everyone time and improving code quality.

Why review with Copilot first:

- **AI has more bandwidth**: Copilot can review code immediately without competing priorities
- **Catch common issues early**: Copilot excels at identifying bugs, logic errors, security vulnerabilities, and style inconsistencies
- **Improve human review quality**: When humans review cleaner code, they can focus on higher-level concerns like design and architecture rather than basic issues
- **Learn from feedback**: Even experienced developers benefit from Copilot's perspective on best practices and potential improvements
- **Growing capabilities**: AI review capabilities continue to improve over time, making this investment increasingly valuable

Copilot review workflow:

1. **Assign Copilot as a reviewer**: On your pull request page, assign Copilot to review the PR the same way you would assign any other reviewer. Click "Reviewers" in the right sidebar and select Copilot from the list.
2. **Review Copilot's comments**: Once Copilot completes its review, carefully examine each comment. For each comment, decide whether you agree with the suggestion:
    - **If the comment is correct**: Address it by making code changes yourself or ask Copilot to apply the fix using GitHub's suggestion features
    - **If the comment is incorrect or not applicable**: Dismiss the comment with an explanation for why it doesn't apply
    - **If you're uncertain**: Seek a second opinion from a human reviewer or do additional research
3. **Request another Copilot review**: After addressing or dismissing all comments, request another review from Copilot. This creates an iterative improvement process.
4. **Iterate until satisfied**: Repeat the review-and-address cycle until Copilot stops providing valuable suggestions. This typically takes 1-3 iterations depending on the complexity of the changes.

5. **Request human review**: Only after you've addressed Copilot's feedback should you request review from human team members. At this point, the code should be in better shape, allowing human reviewers to focus on higher-level concerns.

Important considerations:

- **Copilot isn't perfect**: AI review can produce false positives or miss important issues. Always apply your own judgment when evaluating Copilot's suggestions.
- **Don't blindly accept all suggestions**: Some of Copilot's recommendations may not fit your specific context or requirements. It's perfectly appropriate to dismiss comments that don't apply.
- **Human review remains essential**: Copilot review supplements but does not replace human code review. Humans bring domain knowledge, understanding of business requirements, and judgment about trade-offs that AI cannot replicate.
- **Document dismissals**: When dismissing Copilot comments, briefly explain why. This helps human reviewers understand your reasoning and can serve as documentation for future reference.

For pull request authors:

Even if you're highly experienced, treating Copilot review as a required pre-review step helps maintain code quality and makes the best use of everyone's time. The few minutes spent on Copilot review often save hours of back-and-forth with human reviewers.

For human reviewers:

When you receive a PR for review, check whether the author has completed the Copilot review process. If Copilot hasn't reviewed the PR yet, consider asking the author to complete that step first before you invest time in review. This ensures you're reviewing code that has already been through initial automated quality checks.

18.5.16 Reviewing a Copilot PR You Didn't Create

When reviewing a pull request where someone else prompted Copilot to make changes, follow these guidelines to avoid confusion and ensure smooth collaboration:

Understanding PR roles:

Different people may play different roles in the PR lifecycle:

- **Issue creator**: Reports a bug, suggests a feature, or requests other improvements. In projects with external users, issue creators often cannot assign issues to developers and are therefore distinct from the PR prompter
- **PR prompter**: Assigns a developer (human or AI) to start working on a PR, often by assigning an issue to Copilot. The PR prompter is sometimes the same person as the issue creator, but often is a project maintainer who reviews and triages external issue reports
- **PR author(s)**: Makes commits to the PR branch (when Copilot creates commits, both Copilot and the prompter are co-authors)
- **PR manager**: Supervises and guides the PR authors, assigns reviewers, and controls the PR workflow. Typically the PR prompter becomes the PR manager, but can hand off this role to someone else
- **PR reviewer(s)**: Reviews the PR and provides feedback. The PR manager often also serves as the lead reviewer

- **PR assignee(s)**: Listed in the "Assignees" field on GitHub to indicate who is responsible for the PR. Use this field to clarify current ownership and track who should be working on addressing feedback

The scenario:

- One team member (the "PR prompter") assigned Copilot to work on an issue or explicitly prompted Copilot to start working
- The prompter may or may not be the same person who originally created the issue
  - In projects with a user base, users often submit issues (bug reports, feature requests)
  - A project maintainer then steps in, adds their perspective, and assigns the issue to Copilot
  - In this case, the maintainer who assigned Copilot is the prompter, not the original issue creator
- Copilot created the PR with the prompter as co-author
- The prompter (now acting as PR manager) requested your review
- Copilot may have also automatically reviewed the PR

As a non-manager reviewer, your role is to provide feedback, not to directly initiate more work by Copilot. The PR manager should remain in control of when and how Copilot makes additional changes.

Recommended review workflow:

1. **Use "Comment" or "Request changes" based on the severity of issues**:
   - Use **"Comment"** for suggestions, questions, or minor issues that don't block merging
   - Use **"Request changes"** for significant issues that must be addressed before merging
   - Both options allow you to provide feedback without directly triggering Copilot
2. **Don't ask Copilot to make changes directly**:
   - Avoid using features that would trigger Copilot to start working immediately
   - Let the PR manager decide whether to ask Copilot to address your comments or make changes themselves
3. **Write clear, actionable comments**:
   - Explain what needs to change and why
   - Suggest specific solutions when appropriate
   - The PR manager will decide how to address your feedback

For PR managers:

After receiving reviews from other team members:

1. **Review all comments carefully**:
   - Decide which comments you agree with
   - Dismiss or respond to comments you don't entirely agree with
   - This ensures Copilot only addresses feedback you've validated
2. **Choose how to address valid feedback**:

- **Option A**: Make the changes yourself (faster for simple fixes)
- **Option B**: Ask Copilot to address the feedback (better for complex changes)
- **Option C**: Add your own review summarizing which comments Copilot should address, then ask Copilot to respond to the open comment threads

3. **Maintain clear communication**:

   - Let reviewers know how you plan to address their feedback
   - Mark conversations as resolved after addressing them
   - Request re-review from humans after Copilot makes significant changes
   - Update the PR's "Assignees" field to reflect who is currently responsible for the PR

Transferring the PR manager role:

The original PR manager can hand over a PR to another person, who then becomes the new PR manager with control over Copilot's work on that PR. This might be useful when:

- The original PR manager is unavailable or on leave
- Someone with different expertise needs to guide the remaining work
- Responsibilities are being redistributed within the team

To transfer the PR manager role:

1. The original PR manager should clearly communicate the handover to all reviewers
2. The new PR manager should review the PR's history and any open feedback
3. The new PR manager should take over responding to Copilot and managing future iterations
4. The team should update the PR's "Assignees" field and comments or description to reflect the current PR manager

This workflow ensures the PR manager maintains control over the development process while benefiting from collaborative human review and Copilot's implementation capabilities.

19 Checklists

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

19.1 Pre-analysis plan checklist

- Brief background on the study (a condensed version of the introduction section of the paper)
- Hypotheses / objectives
- Study design
- Description of data
- Definition of outcomes
- Definition of interventions / exposures
- Definition of covariates
- Statistical power calculation
- Statistical model description
- Covariate selection / screening
- Standard error estimation method
- Missing data analysis
- Assessment of effect modification / subgroup analyses
- Sensitivity analyses
- Negative control analyses

19.2 Code checklist

- Does the script run without errors?
- Is code self-contained within repo and/or associated Box folder?
- Is all commented out code / remarks removed?
- Does the header accurately describe the process completed in the script?
- Is the script pushed to its github repository?
- Does the code adhere to the coding style guide[2]?
- Are all warnings ignorable? Should any warnings be intentionally suppressed or addressed?

19.3 Manuscript checklist

This is adapted in part from How to tackle the reproducibility crisis in ten steps (Baker 2019).

[1] https://jadebc.github.io/lab-manual/checklists.html
[2] https://ucd-serg.github.io/lab-manual/coding-style.html

- Have you completed the relevant reporting checklist, if applicable? (See EQUATOR Network ("EQUATOR Network: Enhancing the QUAlity and Transparency of Health Research," n.d.) for a collection of checklists)
- Are the study results within the manuscript replicable (i.e., if you rerun the code in the study's repository, the tables and figures will be exactly replicated?)
- Is a target journal selected?
- Is the title declarative, in other words, does it state the object/findings rather than suggest them?
- Is the word count of the manuscript close to the target journal's allowance?
- Does the manuscript adhere to the formatting guide of the target journal?
- Does the manuscript use a consistent voice (passive or active – usually active is preferred … pun intended)?
- Is each figure and table (including supplementary material) referenced in the main text?
- Is there a caption for each figure and table (including supplementary material)?
- Are tables/figures and supplementary material numbered in accordance with their appearance in the main text?
- Does the text use past tense if it is reporting research findings or future tense if it is a study protocol?
- Does the text avoid subjective wording (e.g., "interesting", "dramatic")?
- Does the text use minimal abbreviations, and are all abbreviations defined at first use?
- Does the text avoid directionless words? (e.g., instead of writing, 'Precipitation influences disease risk', write, 'Precipitation was associated with increased disease risk').
- Does the text avoid making causal claims that are not supported by the study design? Be careful about the words "effect", "increase", and "decrease", which are often interpreted as causal.
- Does the text avoid describing results with the word "significant", which can easily be confused with statistical significance? (see references on this topic here[3])
- Have you drafted author contributions? Do they follow the CRediT: Contributor Roles Taxonomy ("CRediT: Contributor Roles Taxonomy," n.d.) for author contributions?

19.4 Figure checklist

- Are the x-axis and y-axis labeled?
- If the figure includes panels, is each panel labeled?
- Are there sufficient numerical / text labels and breaks on the x-axis and y-axis?
- Is the font size appropriate (i.e., large enough to read, not so large that it distracts from the data presented in the figure?)
- Are the colors used colorblind friendly? See a colorblind-friendly palette here[4], a neat palette generator with colorblind options here[5], and an article on why this matters: The misuse of colour in science communication (Crameri, Shephard, and Heron 2020)
- Are colors/shapes/line types defined in a legend?
- Are the legends and other labels easy to understand with minimal abbreviations?
- If there is overplotting, is transparency used to show overlapping data?

[3] https://journals.lww.com/epidem/Fulltext/2001/05000/The_Value_of_P.2.aspx
[4] http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette
[5] https://medialab.github.io/iwanthue/?utm_source=Nature+Briefing&utm_campaign=2c68711076-briefing-dy-20211006&utm_medium=email&utm_term=0_c9dfd39373-2c68711076-44335685

- Are 95% confidence intervals or other measures of precision shown, if applicable?

20 Resources

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung and Kunal Mishra[1]

20.1 Resources for R

20.1.1 Books and Comprehensive Guides

- R for Data Science (Wickham, Çetinkaya-Rundel, and Grolemund 2023) - comprehensive introduction to doing data science with R
- R Packages (Wickham and Bryan 2023) - complete guide to R package development
- Advanced R (Wickham 2019) - deep dive into R programming and internals
- Mastering Shiny (Wickham 2021) - comprehensive guide to building web applications with Shiny
- Engineering Production-Grade Shiny Apps (Fay et al. 2021) - best practices for production Shiny applications
- Happy Git and GitHub for the useR (Bryan 2023) - guide to using Git and GitHub with R
- Jade's R-for-epi course[2]

20.1.2 UC Davis DataLab Workshops and Tutorials

The UC Davis DataLab[3] provides extensive workshops and learning materials for data science:

- Workshop Index[4] - comprehensive catalog of all DataLab workshops
- R Basics Workshop[5] - foundational R programming for beginners
- Research Toolkits[6] - in-depth guides for research tools and methods
- Install Guides[7] - setup instructions for data science software

20.1.3 Cheat Sheets

- dplyr and tidyr cheat sheet[8]
- ggplot cheat sheet[9]

[1] https://jadebc.github.io/lab-manual/resources.html
[2] https://ucb-epi-r.github.io
[3] https://github.com/ucdavisdatalab
[4] https://ucdavisdatalab.github.io/workshop_index/
[5] https://github.com/ucdavisdatalab/workshop_r_basics
[6] https://github.com/ucdavisdatalab/research-toolkits
[7] https://ucdavisdatalab.github.io/install_guides/
[8] https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
[9] https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

- data table cheat sheet[10]
- RMarkdown cheat sheet[11]

20.1.4 Style and Best Practices

- Hadley Wickham's R Style Guide[12]

20.1.5 Tidy Evaluation Resources

- Tidy Eval in 5 Minutes[13] (video)
- Tidy Evaluation[14] (e-book)
- Data Frame Columns as Arguments to Dplyr Functions[15] (blog)
- Standard Evaluation for *_join[16] (stackoverflow)
- Programming with dplyr[17] (package vignette)

20.2 Resources for Git & Github

- Happy Git and GitHub for the useR (Bryan 2023) - comprehensive guide to using Git and GitHub with R
- GitHub Skills: Introduction to GitHub[18]
- UC Davis DataLab Git Sandbox[19] - hands-on Git practice repository

20.3 Resources for Python

- UC Davis DataLab Python Basics Workshop[20] - foundational Python programming
- Natural Language Processing with Python[21] - text analysis and NLP techniques

20.4 Resources for Julia

- UC Davis Julia Users Group Julia Basics Workshop[22] - foundational Julia programming

[10] https://s3.amazonaws.com/assets.datacamp.com/blog_assets/datatable_Cheat_Sheet_R.pdf
[11] https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
[12] http://adv-r.had.co.nz/Style.html
[13] https://www.youtube.com/watch?v=nERXS3ssntw
[14] https://dplyr.tidyverse.org/articles/programming.html
[15] https://www.brodrigues.co/blog/2016-07-18-data-frame-columns-as-arguments-to-dplyr-functions/
[16] https://stackoverflow.com/questions/28125816/r-standard-evaluation-for-join-dplyr
[17] https://dplyr.tidyverse.org/articles/programming.html
[18] https://github.com/skills/introduction-to-github
[19] https://github.com/ucdavisdatalab/sandbox_git
[20] https://github.com/ucdavisdatalab/workshop_python_basics
[21] https://github.com/ucdavisdatalab/workshop_nlp_with_python
[22] https://ucjug.github.io/workshop_julia_basics/

20.5 Scientific figures

- Ten Simple Rules for Better Figures (Rougier, Droettboom, and Bourne 2014)

20.6 Writing

- Unpacking the Scientific Toolbox (Silbiger and Stubler 2019)
- ICMJE Definition of authorship (International Committee of Medical Journal Editors, n.d.)
- Computational science: …why scientific programming does not compute (Merali and Giles 2010)
- The Pathway to Publishing: A Guide to Quantitative Writing in the Health Sciences[23]
- Principles of Scientific Writing[24] - a handbook covering scientific writing principles including citations and evidence, word choice, and conciseness
- Secret, actionable writing tips[25]

20.7 Presentations

- How to tell a compelling story in scientific presentations (Van Noorden 2021)
- How to give a killer narratively-driven scientific talk[26]
- How to make a better poster[27]
- How to make an even better poster[28]

20.8 Professional advice

- Professional advice, especially for your first job[29]
- Team Public Health Substack[30]

20.9 Funding

- Building Your Funding Train[31]
- NIH Grant Writing Resources[32]

[23] https://link.springer.com/book/10.1007/978-3-030-98175-4
[24] https://github.com/d-morrison/psw
[25] https://x.com/acagamic/status/1680381737816424450
[26] https://www.sciencedirect.com/science/article/pii/S1471491423000928
[27] https://www.youtube.com/watch?v=1RwJbhkCA58&t=1171s
[28] https://mitcommlab.mit.edu/be/2023/09/27/toward-an-evenbetterposter-improving-the-betterposter-template/
[29] https://docs.google.com/document/d/1ckgRCcr7FFPyymyMA6Y_-cB_vftOyrrxT28c6gRg0PI/edit
[30] https://teampublichealth.substack.com/
[31] https://grantwriting.stanford.edu/funding-train/#ep
[32] https://www.niaid.nih.gov/grants-contracts/write-grant-application

20.10 Ethics and global health research

- Global Code of Conduct For Research in Resource-Poor Settings[33]
- Addressing power asymmetries in global health (Abimbola et al. 2022)
- Transforming Global Health Partnerships[34]

[33] https://www.globalcodeofconduct.org/
[34] https://link.springer.com/book/9783031537929

21 Professional Development

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

21.1 Mentoring Philosophy

We believe in individualized mentoring that supports each person's unique career goals. Effective mentoring requires:

- Regular, open communication between mentees and mentors
- Mutual respect and trust
- Clear expectations and goals
- Constructive feedback
- Support for both research and career development

21.2 Individual Development Plans

All graduate students and postdocs should maintain an Individual Development Plan (IDP) that outlines:

- Short-term and long-term career goals
- Skills to develop
- Training needs and opportunities
- Timeline and milestones
- Progress toward goals

Update your IDP at least annually and discuss it with your PI. Useful resources:

- myIDP - Individual Development Plan tool[2]
- NIH OITE Career Resources[3]

21.3 Presentations and Conferences

We encourage lab members to present their work at conferences and seminars:

- Discuss conference opportunities with PIs early
- Submit abstracts with PI approval
- Practice presentations in lab meeting before the conference
- Funding for conferences depends on availability and should be discussed in advance

[1] https://jadebc.github.io/lab-manual/
[2] https://myidp.sciencecareers.org/
[3] https://www.training.nih.gov/career-services/

Resources for effective presentations:

- How to tell a compelling story in scientific presentations[4]
- How to give a killer narratively-driven scientific talk[5]
- How to make a better poster[6]
- How to make an even better poster[7]

21.4 Scientific Figures

Creating clear, effective figures is essential for communicating research findings:

- Label x-axis and y-axis clearly
- Use panel labels when including multiple subplots
- Include sufficient tick marks and labels
- Use appropriate font sizes (readable but not distracting)
- Use colorblind-friendly palettes (see ColorBrewer[8] or iwanthue[9])
- Define colors, shapes, and line types in legends
- Minimize abbreviations in labels and legends
- Use transparency to show overlapping data
- Show measures of precision (e.g., 95% confidence intervals)

Resources:

- Ten Simple Rules for Better Figures[10]

21.5 Grant Writing

- Graduate students and postdocs are encouraged to apply for fellowships (e.g., NIH F31, NSF GRFP, K awards)
- PIs will support fellowship applications with feedback, letters of support, and mentoring
- Start planning fellowship applications well in advance of deadlines (typically 3-6 months)
- Attend grant writing workshops and seek feedback from multiple sources

Resources:

- Building Your Funding Train[11]
- NIH Funding Opportunities[12]
- NIH Grant Writing Resources[13]

[4] https://www.nature.com/articles/d41586-021-03603-2
[5] https://www.sciencedirect.com/science/article/pii/S1471491423000928
[6] https://www.youtube.com/watch?v=1RwJbhkCA58
[7] https://mitcommlab.mit.edu/be/2023/09/27/toward-an-evenbetterposter-improving-the-betterposter-template/
[8] http://colorbrewer2.org/
[9] https://medialab.github.io/iwanthue/
[10] https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833
[11] https://grantwriting.stanford.edu/funding-train/
[12] https://grants.nih.gov
[13] https://www.niaid.nih.gov/grants-contracts/write-grant-application

21.6 PhD Dissertation Requirements

Understanding what constitutes a sufficient PhD dissertation is crucial for setting realistic expectations and timelines. The dissertation represents an important milestone, but it doesn't need to be your magnum opus.

21.6.1 Review Previous Dissertations

Before setting your dissertation goals, read previous dissertations from students in your program. This helps you:

- Understand the typical scope and depth expected
- See different approaches to structure and presentation
- Calibrate your expectations based on successful examples
- Identify common patterns and standards in your field

Most universities maintain electronic dissertation repositories, making it easy to access recent examples from your program.

21.6.2 Publication Requirements

Three first-author papers typically suffice for a dissertation in public health and biomedical sciences. If academic peers in reputable journals have approved your work through peer review, this demonstrates that your research meets professional standards. Your dissertation committee should recognize this external validation.

The specific publication requirements may vary by program and institution, so consult your program's guidelines and discuss expectations with your committee early. However, three substantial first-author publications generally demonstrate:

- Independent research capability
- Ability to communicate findings effectively
- Contribution to the scientific literature
- Readiness for an academic or research career

21.6.3 External Validation and Fast-Tracking

If you have a job offer waiting, you can usually get fast-tracked through the dissertation process. The reasoning is straightforward: your work has been externally validated as worthwhile by prospective employers.

Since most post-PhD positions offer better compensation than graduate stipends, it's difficult to justify prolonging your graduation when you've already demonstrated professional competence. This applies whether the job offer is in academia, industry, government, or nonprofit sectors.

21.6.4 Historical Context: The Masterpiece Tradition

The dissertation is a spiritual successor to an apprentice's masterpiece[14] in craft guilds. Historically, a masterpiece was the piece of work that demonstrated an apprentice had achieved sufficient skill to join the guild as a master craftsperson. It was not meant to be the best work they would ever produce—it was meant to prove they were ready to work independently.

Similarly, your dissertation should demonstrate that you're ready to conduct independent research. It's your first professional-level work, not your career highlight. This perspective helps set appropriate expectations:

- The dissertation proves you can conduct rigorous research
- It doesn't need to solve every problem in your field
- It doesn't need to be flawless
- It doesn't need to be all-encompassing
- It just needs to constitute incremental progress in your field

21.6.5 Setting Realistic Expectations

Many PhD students struggle with perfectionism or "scope creep" in their dissertations. Remember:

- Done is better than perfect when it comes to dissertations
- You'll have your entire career to refine and expand on these ideas
- The goal is to finish and move forward, not to write the definitive work on your topic
- Your committee wants to see you succeed and graduate

Focus on making a solid, incremental contribution to knowledge in your field. That's what a dissertation is meant to be—no more, no less.

21.6.6 Resources

Dissertation writing tools:

- quarto-thesis[15] - Quarto extension for creating masters or PhD theses with professional LaTeX formatting

21.7 Teaching and Outreach

Teaching and outreach are valuable professional development opportunities:

- Graduate students are encouraged to gain teaching experience
- We support science communication and outreach activities
- Discuss opportunities with PIs

[14] https://en.wikipedia.org/wiki/Masterpiece#History
[15] https://github.com/nmfs-opensci/quarto-thesis

The UC Davis DataLab[16] offers various workshops and learning materials that can support your teaching and professional development. Their workshop index[17] provides a comprehensive catalog of available resources.

21.8 Networking

Building a professional network is important for your career:

- Attend seminars and departmental events
- Connect with researchers in your field
- Join professional societies
- Use professional social media platforms to share research and engage with the scientific community

[16] https://github.com/ucdavisdatalab
[17] https://ucdavisdatalab.github.io/workshop_index/

22 Writing

22.1 Writing to Clarify Your Thinking

Writing is a powerful tool for clarifying your thoughts, even before you start searching for answers. When questions or ideas come up, it's good practice to write them down immediately—this helps you:

- Organize your thoughts and identify what you actually need to know
- Articulate the problem more precisely
- Often realize the answer yourself through the process of writing

As Leslie Lamport, a Turing Award winner and computer scientist, states: "If you think you understand something, and don't write down your ideas, you only think you're thinking" (Lamport 2019).

23 Manuscript Preparation and Publication

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

23.1 Publication Process

The typical workflow for manuscript preparation and publication:

1. **Planning**: Discuss target journals, outline, and timeline with PIs
2. **Drafting**: Lead author prepares initial draft
3. **Internal review**: Co-authors review and provide feedback
4. **Revision**: Lead author incorporates feedback iteratively
5. **PI approval**: Obtain final approval from PIs before submission
6. **Submission**: Submit to journal
7. **Revisions**: Lead author coordinates response to peer reviewers
8. **Publication**: Celebrate and share!

23.2 Preprints and Open Access

- We encourage posting preprints prior to or during peer review on platforms like medRxiv[2] or bioRxiv[3]
- Preprints allow rapid dissemination of findings and can be cited in grant applications
- We support publishing in open access journals when possible to maximize accessibility
- Many funders, including NIH, have public access policies that require making publications freely available

A preprint is a scientific manuscript that has not been peer reviewed. Preprint servers create digital object identifiers (DOIs) and can be cited in other articles and in grant applications. Because the peer review process can take many months, publishing preprints prior to or during peer review enables other scientists to immediately learn from and build on your work. Importantly, NIH allows applicants to include preprint citations in their biosketches. In most cases, we publish preprints on medRxiv[4].

23.3 Reporting Checklists

Using reporting checklists ensures that publications contain information needed for readers to assess validity and reproducibility. We use checklists appropriate to study design:

[1] https://jadebc.github.io/lab-manual/
[2] https://www.medrxiv.org/
[3] https://www.biorxiv.org/
[4] https://www.medrxiv.org/

- CONSORT for randomized trials
- STROBE for observational studies
- PRISMA for systematic reviews
- Others as appropriate (see EQUATOR Network[5])

23.4 Manuscript Checklist

Before submitting a manuscript:

- ☐ Completed relevant reporting checklist
- ☐ Results are reproducible (rerunning code replicates tables/figures exactly)
- ☐ Target journal selected
- ☐ Title is declarative and states findings clearly
- ☐ Word count meets journal requirements
- ☐ Manuscript follows journal formatting guidelines
- ☐ Consistent voice throughout (typically active voice)
- ☐ All figures and tables referenced in main text
- ☐ Captions for all figures and tables
- ☐ Tables/figures numbered by order of appearance
- ☐ Abbreviations defined at first use and used sparingly
- ☐ Avoid subjective wording (e.g., "interesting", "dramatic")
- ☐ Avoid directionless statements (specify direction of associations)
- ☐ Causal language only when supported by study design
- ☐ Avoid "significant" (easily confused with statistical significance)
- ☐ Author contributions drafted using CRediT Taxonomy

23.5 Scientific Writing: Claims and Evidence

All factual claims in scientific writing should be supported by appropriate evidence.

Citation requirements:

- **Cite primary sources** for factual statements about established knowledge, methods, or findings
- **Cite official documentation** when describing how software, tools, or systems work
- **Link to authoritative sources** like peer-reviewed publications, official repositories, or technical specifications
- Avoid citing secondary sources when primary sources are available

When you can't find a citation:

- **Demonstrate directly**: Show the behavior through experiments, data, or explicit examples
- **Acknowledge uncertainty**: Use appropriate hedging language ("may", "appears to", "in our experience") when evidence is limited
- **Remove the claim**: If you cannot substantiate a claim with either citations or direct evidence, consider whether it needs to be included

Why this matters:

[5] https://www.equator-network.org/

- Builds reader trust and credibility
- Enables readers to verify information independently
- Maintains scientific rigor in all communications
- Prevents propagation of misinformation

This principle applies to all lab writing, including: manuscripts, documentation, grant applications, and technical reports.

Using AI tools for writing:

When using AI tools to help develop manuscripts or other academic writing, follow the special guidance in Section 18.4 to ensure transparency, maintain intellectual ownership, and avoid plagiarism.

References

"2001: A Space Odyssey." 1968. Film. https://en.wikipedia.org/wiki/2001:_A_Space_Odyssey_(film).

Abimbola, Seye, Sheena Asthana, Cristina Montenegro, et al. 2022. "Addressing Power Asymmetries in Global Health: Imperatives in the Wake of the COVID-19 Pandemic." *PLOS Global Public Health* 2 (10). https://doi.org/10.1371/journal.pgph.0002269.

Asimov, Isaac. 1950. "I, Robot." New York: Novel; Gnome Press. https://search.library.ucdavis.edu/permalink/01UCD_INST/9fle3i/alma990000226350403126.

Baker, Monya. 2019. "How to Tackle the Reproducibility Crisis in Ten Steps." *Nature*. https://doi.org/10.1038/d41586-019-01431-z.

"Battlestar Galactica." 2004. Television Series. https://en.wikipedia.org/wiki/Battlestar_Galactica_(2004_TV_series).

Benjamin-Chung, Jade, Kunal Mishra, Stephanie Djajadi, Nolan Pokpongkiat, Anna Nguyen, Iris Tong, and Gabby Barratt Heitmann. 2024. "Benjamin-Chung Lab Manual." https://jadebc.github.io/lab-manual/.

"Blade Runner." 1982. Film. https://en.wikipedia.org/wiki/Blade_Runner.

Bryan, Jennifer. 2023. *Happy Git and GitHub for the useR*. https://happygitwithr.com/.

Bryan, Jennifer, Jim Hester, David Robinson, Hadley Wickham, and Christophe Dervieux. 2024. *Reprex: Prepare Reproducible Example Code via the Clipboard*. https://reprex.tidyverse.org/.

Card, Orson Scott. 1985. "Ender's Game." Novel; Tor Books. https://en.wikipedia.org/wiki/Ender%27s_Game.

Crameri, Fabio, Grace E. Shephard, and Philip J. Heron. 2020. "The Misuse of Colour in Science Communication." *Nature Communications* 11. https://doi.org/10.1038/s41467-020-19160-7.

"Creative Commons Attribution-NonCommercial 4.0 International License." n.d. Creative Commons. http://creativecommons.org/licenses/by-nc/4.0/.

"CRediT: Contributor Roles Taxonomy." n.d. PLOS ONE. https://journals.plos.org/plosone/s/authorship.

"Dryad Digital Repository." n.d. Dryad. https://datadryad.org/.

"EQUATOR Network: Enhancing the QUAlity and Transparency of Health Research." n.d. EQUATOR Network. https://www.equator-network.org/.

Fay, Colin, Sébastien Rochette, Vincent Guyader, and Cervan Girard. 2021. *Engineering Production-Grade Shiny Apps*. Chapman; Hall/CRC. https://engineering-shiny.org/.

Flight of the Conchords. 2007. "The Humans Are Dead." Music Video. https://www.youtube.com/watch?v=B1BdQcJ2ZYY.

"GitHub Desktop." n.d. GitHub. https://desktop.github.com/.

Herbert, Frank. 1965. "Dune." Novel. https://en.wikipedia.org/wiki/Organizations_of_the_Dune_universe#Thinking_machines.

"How to Store and Manage Your Data." n.d. PLOS. https://plos.org/resource/how-to-store-and-manage-your-data/.

International Committee of Medical Journal Editors. n.d. "Defining the Role of Authors and Contributors." ICMJE. http://www.icmje.org/recommendations/browse/roles-and-responsibilities/defining-the-role-of-authors-and-contributors.html.

Lamport, Leslie. 2019. "Think and Write with Leslie Lamport." Podcast interview. https://mentors.fm/2019/08/13/think-and-write-with-leslie-lamport/.

LeCun, Yann. 2022. "A Path Towards Autonomous Machine Intelligence." Meta AI Research; New York University; Technical Report. https://openreview.net/forum?id=BZ5a1r-kVsf.

"medRxiv: The Preprint Server for Health Sciences." n.d. Cold Spring Harbor Laboratory. https://www.medrxiv.org/.

Merali, Zeeya, and Jim Giles. 2010. "Computational Science: Error ... Why Scientific Programming Does Not Compute." *Nature* 467: 775–77. https://doi.org/10.1038/467775a.

Munafò, Marcus R., Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. 2017. "A Manifesto for Reproducible Science." *Nature Human Behaviour* 1. https://doi.org/10.1038/s41562-016-0021.

Nuzzo, Regina. 2015. "How Scientists Fool Themselves – and How They Can Stop." *Nature* 526: 182–85. https://doi.org/10.1038/526182a.

"Open Science Framework." n.d. Center for Open Science. https://osf.io/.

Robertson, Seth. n.d. "On Undoing, Fixing, or Removing Commits in Git." https://sethrobertson.github.io/GitFixUm/fixup.html.

Rougier, Nicolas P., Michael Droettboom, and Philip E. Bourne. 2014. "Ten Simple Rules for Better Figures." *PLOS Computational Biology* 10 (9). https://doi.org/10.1371/journal.pcbi.1003833.

Silbiger, Nyssa J., and Ariel D. Stubler. 2019. "Unpacking the Scientific Toolbox: Five Skills for the Modern Scientist." *Nature*. https://doi.org/10.1038/d41586-019-02918-5.

"Slurm Workload Manager: Sbatch Documentation." n.d. SchedMD. https://slurm.schedmd.com/sbatch.html.

Stoddart, Charlotte. 2019. "Is There a Reproducibility Crisis in Science?" *Nature*. https://doi.org/10.1038/d41586-019-00067-3.

"Terminator 3: Rise of the Machines." 2003. Film. https://en.wikipedia.org/wiki/Terminator_3:_Rise_of_the_Machines.

"The Matrix." 1999. Film. https://en.wikipedia.org/wiki/The_Matrix.

Tidyverse Team. 2023. *Tidyverse Code Review Principles*. https://code-review.tidyverse.org/.

Van Noorden, Richard. 2021. "Scientists and Science Communicators Swap Tips on How to Tell Compelling Stories." *Nature*. https://doi.org/10.1038/d41586-021-03603-2.

"WarGames." 1983. Film. https://en.wikipedia.org/wiki/WarGames.

Wickham, Hadley. 2019. *Advanced r*. 2nd ed. Chapman; Hall/CRC. https://adv-r.hadley.nz/.

———. 2021. *Mastering Shiny*. O'Reilly Media. https://mastering-shiny.org/.

———. 2023a. *The Tidyverse Style Guide*. https://style.tidyverse.org/.

———. 2023b. *Tidyverse Design Guide*. https://design.tidyverse.org/.

Wickham, Hadley, and Jennifer Bryan. 2023. *R Packages*. 2nd ed. O'Reilly Media. https://r-pkgs.org/.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science*. 2nd ed. O'Reilly Media. https://r4ds.hadley.nz/.

Wickham, Hadley, Peter Danenberg, Gábor Csárdi, and Manuel Eugster. 2024. *Roxygen2: In-Line Documentation for r*. https://roxygen2.r-lib.org/.

Copilot Instructions File

For the complete `.github/copilot-instructions.md` file, please view the HTML version of this appendix at: https://ucd-serg.github.io/lab-manual/appendix-copilot-instructions.html

Copilot Setup Steps File

For the complete `.github/workflows/copilot-setup-steps.yml` file, please view the HTML version of this appendix at: https://ucd-serg.github.io/lab-manual/appendix-copilot-setup-steps.html

Document Generation Metadata

This document was generated from the following git commit:

- **Branch:** HEAD
- **Commit:** 30157ed
- **Full commit hash:** 30157edaac8cee578b54df594d71d8ed9211e967
- **Commit date:** 2026-01-28 00:42:17 +0000

When transferring edits from this DOCX file back to the Quarto source files, use this commit information to set up the PR correctly and account for any commits that have been added since this document was generated.