Code repositories

Last modified: 2026-11-19 23:48:51 (PST)

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 to collaborate across projects.

We recommend R package structure for most R-based projects. This provides benefits for reproducibility, collaboration, and code quality even for analysis-only projects.

Template repositories

When starting a new package, begin from a template repository, such as `Morrison-Lab/rpt`, rather than an empty repository. On GitHub, click **Use this template** to create a new repository that already includes a standard structure, CI workflows, and documentation scaffolding. Then replace the placeholder package name, `Title`, and `Description` in `DESCRIPTION` with your project's details.

A note on GitLab

GitHub-style template repositories do **not** have a direct equivalent in GitLab Community Edition (CE):

- **Built-in project templates** (Rails, Node, and similar) ship with all editions, including CE, but you cannot add your own to that list.
- **Custom project templates** (instance- or group-level) — the feature most analogous to GitHub's template repositories — are Premium/Ultimate tier only, so they are unavailable in CE.
- On CE, the practical workaround is to **fork** (or clone and re-push) a "seed" project to start new work from it.

In short: if you mirror this workflow onto a self-managed GitLab CE instance, plan to either fork a seed repository or budget for a paid tier.

Package Structure

We recommend structuring every R project as an R package, even one that is primarily an analysis project and not intended for distribution on CRAN or Bioconductor. This standardized structure provides numerous benefits:

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

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
  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)
myproject.Rproj     # RStudio project file
```

Where to Place Analysis Files

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

Additional considerations for choosing articles over vignettes:

Articles are particularly useful when (Wickham and Bryan 2023, sec. 17.4.1):

- **Code shouldn't execute on CRAN**: When your code requires authentication credentials, depends on external services, or takes too long to run
- **Demonstrating integration with other packages**: When you want to show your package working with packages you don't want to formally depend on
- **Graphics-heavy content**: When including many or large graphics would make the package too large for CRAN

- **Trade-off**: Articles are less accessible than vignettes for users with limited internet access, as they don't ship with the package and only appear on the pkgdown website

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.

Analyses with Restricted Data (inst/analyses/)

For analyses that rely on **private, sensitive, or restricted data**, place `.qmd` or `.Rmd` 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")
```

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
```

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. `Rscript` prints output and errors to the terminal and writes no log file, so check that output. Alternatively, run each script with `R CMD BATCH`, which writes a `.Rout` log file for each script, and check those files to ensure everything has run correctly.

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
```

`.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.

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.

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.R")`, 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 referenced in multiple scripts (e.g. a `merged_data_path`) can be declared in `0-config.R` and simply referred to by their variable names 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 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.

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.

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 Unix 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.

Repository Status Badges

Communicating the development and support status of a repository helps both human collaborators and automated tools understand whether a project is actively maintained, a work in progress, or an archived analysis. The repostatus.org project provides a standardized, machine-readable and human-readable convention for declaring project status through badges in the repository `README.md`.

Standard repository statuses

repostatus.org defines eight standardized statuses spanning both implementation progress and ongoing maintenance commitment:

- **Concept:** Minimal or no implementation has been done yet, or the repository is only intended to be a limited example, demo, or proof-of-concept.
- **WIP (Work in Progress):** Initial development is in progress, but there has not yet been a stable, usable release suitable for the public.
- **Suspended:** Initial development has started, but there has not yet been a stable, usable release; the project has been put on hold or stopped.
- **Abandoned:** Initial development has started, but has been abandoned; the project is no longer being worked on.
- **Active:** The project has reached a stable, usable state and is being actively developed.
- **Inactive:** The project has reached a stable, usable state but is no longer being actively developed; support/maintenance will be provided as time allows.
- **Unsupported:** The project has reached a stable, usable release, but the person or organization responsible for it has decided to abandon it or drop support for it permanently.
- **Moved:** The project has been moved to a new location, and the version at that location should be considered current and authoritative.

Adding a status badge

Place the status badge near the top of your repository `README.md`, immediately beneath the main title. The badge links to the corresponding status definition on repostatus.org.

For an active project:

```
[![Project Status: Active - The project has reached a stable, usable state and is being activ
```

For a project in initial development:

```
[![Project Status: WIP - Initial development is in progress, but there has not yet been a sta
```

For a completed analysis repository that is stable but no longer actively adding features:

```
[![Project Status: Inactive - The project has reached a stable, usable state but is no longer
```

(Note: An ASCII hyphen - is used in the badge alt text above as an ASCII-safe approximation of repostatus.org's Unicode en-dash to ensure compatibility across all plain-text and build tools.)

Lifecycle management

Update the status badge as your project evolves through its lifecycle:

1. **Start as Concept or WIP:** Begin new repositories with `Concept` (for early prototypes) or `WIP` (when building an analysis pipeline or package).
2. **Promote to Active:** Switch to `Active` once the codebase reaches a functional, reproducible state with working scripts (and, where applicable, tests).
3. **Transition upon project completion:** When an epidemiology study or manuscript is published and the analysis code is finalized, update the badge to `Inactive` or `Unsupported` so readers know whether future bug fixes or dependency updates are planned.

Wickham, Hadley, and Jennifer Bryan. 2023. *R Packages*. 2nd ed. O'Reilly Media. https://r-pkgs.org/.