R Coding Practices

Last modified: 2026-11-14 03:52:56 (PST)

Protocols for Code and Data

Just as wet labs have strict safety protocols to ensure reproducible results and prevent contamination, computational work needs 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.

The Tool-Building Mindset

A tool is anything that makes a task easier when that task would otherwise be repetitive, hard, or impossible — a script, a function, a package, a spreadsheet, a checklist, or a CI workflow. Aaron Crosman's Tool Building Mindset argues for taking that expansive view deliberately: everyone who works with technology should, can, and already does build tools for themselves, whether or not they think of themselves as a "developer."

The mindset in practice:

- **Notice the friction.** When you catch yourself repeating a task, following a fragile manual sequence, or avoiding something because it is tedious, treat that as a signal to build (or find) a tool for it.
- **Look before you build — then chip away.** Check whether an existing tool already solves the problem (see Prefer Existing Packaged Functions); when nothing quite fits, don't let "not worth the effort" stop you: a tool that removes part of an annoying task is already good enough to start with.
- **Start small and grow the tool as it earns it.** A copy-pasted snippet becomes a function; a function used across analyses becomes a package (see Section); a workflow copied between repositories becomes a shared, reusable one.
- **Build for your own repeated use first.** A tool that saves you an hour a week justifies itself quickly, and the discipline of packaging it (documentation, tests) makes it usable by the next person too.
- **Don't gold-plate.** The tool only needs to beat doing the task by hand; polish it further only when its use grows to warrant it.

Much shared research infrastructure comes from exactly this pattern: the Morrison Lab's `{snapr}` package packages snapshot-testing helpers, the `{lms}` linter package, written for the UCD-SERG lab manual and now kept in this guide's repository, packages a shared lint configuration, and the Morrison-Lab/gha repository collects reusable CI workflows and composite actions shared across the Morrison Lab's repositories.

Essential R Package Development Tools

The following tools are essential for R package development:

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
usethis::use_git()                  # Initialize git
usethis::use_github()               # Connect to GitHub
usethis::use_testthat()             # Set up testing infrastructure
```

```r
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
```

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
```

Run `devtools::check()` locally before pushing

Running `R CMD check` workflows on GitHub Actions introduces runner startup overhead and delay, as virtual environments must be provisioned and system-level dependencies configured on each run. Always run `devtools::check()` locally before pushing to GitHub to catch errors, warnings, and NOTEs early while conserving CI resources.

pkgdown: Package Websites

`pkgdown` builds beautiful documentation websites from your package:

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

# Set up pkgdown
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`:

```
url: https://YOUR-ORG.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
```

Alternatives to pkgdown

While {pkgdown} is the standard tool for creating package documentation websites, several alternatives exist that may better suit specific needs, particularly for generating multiple output formats beyond HTML.

altdoc

{altdoc} is a flexible alternative that supports multiple documentation frameworks, including Quarto, Docsify, MkDocs, and Docute. It is especially useful when you need to generate documentation in multiple formats.

Key features:

- Supports multiple documentation frameworks (Quarto, Docsify, MkDocs, Docute)
- Renders Quarto and R Markdown vignettes to HTML websites
- When using Quarto as the framework, vignettes can be authored to support multiple output formats (HTML, PDF, DOCX, reveal.js presentations), and Quarto can include download links for alternative formats on the HTML site (see Quarto documentation on multiple formats)
- Handles function reference pages, README, NEWS, and other standard documentation
- Easy preview and deployment workflow

Basic usage:

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

# Set up documentation with Quarto
altdoc::setup_docs(tool = "quarto_website")

# Or use other frameworks
# altdoc::setup_docs(tool = "docsify")
# altdoc::setup_docs(tool = "mkdocs")
# altdoc::setup_docs(tool = "docute")

# Render documentation
altdoc::render_docs()

# Preview in browser
altdoc::preview_docs()
```

When to choose altdoc:

- You want to use Quarto's modern publishing system for your documentation website
- You need vignettes that can be downloaded in multiple formats (when authored with Quarto's multi-format support)
- You prefer a different documentation framework than pkgdown's Bootstrap-based approach (Docsify, MkDocs, or Docute)
- You need more flexibility in site design and structure

pkgsite

{pkgsite} provides a minimal, lightweight alternative to pkgdown, focusing on simplicity and ease of customization.

Key features:

- Minimal CSS framework (no Bootstrap)
- Simple, clean design that is easy to customize
- Similar `build_site()` function to pkgdown
- Lightweight and fast
- Can be published via GitHub Pages or other static site hosts

Basic usage:

```r
# Install pkgsite (not on CRAN as of early 2026)
pak::pkg_install("pachadotdev/pkgsite")

# Build and preview site
pkgsite::build_site(preview = TRUE)

# Build with custom URL and lazy rebuilding
pkgsite::build_site(
  # Replace with your real docs domain.
  url = "https://example.com",
  lazy = TRUE,
  preview = TRUE
)
```

When to choose pkgsite:

- You want a minimal, lightweight documentation site
- You prefer to customize CSS styling from scratch
- You don't need the extensive features of pkgdown
- You want faster build times for simple packages

Quarto for Package Documentation

While not a dedicated package documentation tool, Quarto can be used to create sophisticated documentation websites for R packages, particularly when combined with custom scripts or tools like `{ecodown}` (experimental and intended for internal use).

Discussion and resources:

- Quarto discussion on pkgdown alternative
- pkgdown itself now supports Quarto vignettes (as of version 2.1.0)

When to consider Quarto:

- You want complete control over site structure and design
- You're already using Quarto for other documentation
- You need advanced publishing features beyond what pkgdown offers
- You're comfortable writing custom scripts for function reference extraction

Comparison Summary

Tool	Complexity	Website Output	Multi-format Support	Customization	Best For
pkgdown	Standard	HTML	Limited (via Quarto vignettes)	Template-based	Most R packages, standard documentation needs
altdoc	Moderate	HTML	Yes (via Quarto-authored vignettes with download links)	Framework-dependent	Quarto-based workflows, flexible frameworks
pkgsite	Minimal	HTML	No	High (simple CSS)	Lightweight sites, custom styling
Quarto	Advanced	HTML (for websites)	Yes (native - can output to PDF, DOCX, reveal.js, EPUB)	Complete	Full control, advanced features

Recommendation: Use pkgdown for most standard R package documentation needs. Consider altdoc when you prefer Quarto's publishing system or want flexibility to choose between documentation frameworks (Quarto, Docsify, MkDocs, Docute). If you choose altdoc with Quarto, you can author vignettes to provide downloadable PDF and DOCX versions alongside the HTML website. Choose pkgsite for minimalist websites with easy CSS customization. Use Quarto directly only if you need complete control and are comfortable with more complex setup.

Complete Package Development Workflow

Here's a typical workflow for developing an R package:

1. Initial Setup

Starting from a template (recommended):

Using an R package template such as `Morrison-Lab/rpt` is the fastest way to get started with a new R package, as it provides pre-configured settings, GitHub Actions workflows, and development tools:

- **R Package Template (`rpt`)** - Our recommended template with pre-configured development tools and CI workflows:
  - Repository: https://github.com/Morrison-Lab/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}`, `{devtools}`, `{roxygen2}`)
- Testing infrastructure (`{testthat}`)
- 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()
```

2. Add Dependencies

```r
# 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")
```

3. Write Functions

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

```r
#' 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
}
```

4. Document

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

5. Test

Create tests in `tests/testthat/`:

```r
# 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:

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

6. Check

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

Fix any warnings or errors before proceeding.

7. Build Documentation Site

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

8. Share and Publish

```r
# 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. Very generally, you want to:

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 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.

Testing Requirements

ALWAYS establish tests BEFORE modifying functions. This ensures changes preserve existing behavior and new behavior is correctly validated.

Add New Tests for New Features

When a PR adds a feature, give it a new test rather than editing an existing test's inputs or expectations to also cover it. Repurposing a test this way folds two changes into one diff — the reviewer can no longer tell, at a glance, whether the edited assertions still protect the old behavior or now only protect the new one.

For example, suppose a `run_model()` test originally covered the `"power"` method and gets edited in place to cover `"exponential"` instead, rather than gaining a new test alongside it:

```
# Avoid --- repurposing an existing test to also cover a new method
test_that("run_model produces expected output", {
  result <- run_model(data, method = "exponential") # was "power"
  expect_snapshot_value(result)
})

# Preferred --- a new test for the new feature
test_that("run_model handles the exponential method", {
  result <- run_model(data, method = "exponential")
  expect_snapshot_value(result)
})
```

If the new feature makes an existing test redundant, leave that redundancy alone for now. Remove or consolidate redundant tests in a separate, focused PR, so a reviewer can evaluate "does the new feature work" and "which old tests are now redundant" as two independent questions.

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

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

The Morrison Lab's {snapr} package (on CRAN) adds convenience wrappers on top of {testthat}'s snapshot functions: `expect_snapshot_data()` for data frames (adapted, with permission, from the {ssdtools} package) and `expect_snapshot_object()`, which generalizes it to any R object.

```
test_that("prep_study_data output is stable", {
  result <- prep_study_data(raw_data)
  snapr::expect_snapshot_data(result, name = "prepped_study_data")
})
```

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)
})
```

Choosing Between Snapshot and Explicit Tests

The two styles answer different questions. A snapshot test asks "does this output still match what it looked like before?"; an explicit test asks "does this specific property equal this exact value?". That difference drives every practical trade-off, as Table 1 summarizes:

Table 1: Trade-offs between snapshot and explicit tests

	Snapshot test	Explicit test
Conciseness	High: one snapshot call pins the whole output, standing in for a long list of assertions, and the first run records the reference automatically	Lower: you write an assertion per property you care about, which adds up fast for complex outputs
Readability	Lower: the test shows *that* output is pinned, not *why*	Higher: each assertion documents an intended behavior
When output legitimately changes	One review-and-accept of the new snapshot	Rewrite the affected assertions by hand
Failure diagnosis	A diff of the whole output; the relevant change may be buried	Points at the exact property that broke

Each style has a characteristic failure mode to guard against:

- **Snapshot blindness.** Because a snapshot diff can be long, it is tempting to accept an updated snapshot without reading it — which silently converts a regression into the new expected output. Treat every `testthat::snapshot_accept()` like a code review: read the diff (`testthat::snapshot_review()`) before accepting.
- **Brittle snapshots.** Non-deterministic output (timestamps, random numbers, file paths) makes a snapshot fail on every run; seed or fix the inputs first (see the seeding advice under Section).
- **Assertion sprawl.** Explicitly asserting every field of a large object takes a page of fragile code — exactly the case where one snapshot is clearer.
- **Over-specified assertions.** Explicit tests pinned to incidental details (element order that doesn't matter, formatting) fail on harmless refactors and train you to distrust them.

In practice the strongest suites combine both: explicit assertions pin the core logic (the value a calculation must produce, the condition an input must trigger), and a snapshot guards the surrounding large structure (the full prepared data frame, a printed summary) against unnoticed drift. The `{testthat}` snapshotting vignette covers the workflow — recording, reviewing, and accepting snapshots — in more depth.

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

When testing conditions with `{testthat}`, assert on the condition `class` whenever possible. Message text (`regexp`) can be checked as a secondary assertion, but should not be the only signal.

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

  # Missing required columns
  expect_error(
    prep_study_data(data.frame(x = 1)),
    class = "prep_study_data_missing_columns_error",
    regexp = "Missing required columns"
  )

  # 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)))
})
```

Test-Driven Development

Test-driven development (TDD) inverts the usual order of coding and testing: you write the test for a behavior *before* writing the code that implements it. The practice was popularized by Kent Beck, who developed it as part of extreme programming and described it in *Test-Driven Development: By Example* (2002).

Work proceeds in a short, repeated cycle (often called *red-green-refactor*):

1. **Red.** Write a small test for the next behavior you want, and run it to confirm it fails — this proves the test can actually detect the missing behavior.
2. **Green.** Write the simplest code that makes the test pass, resisting the urge to build ahead of the tests.
3. **Refactor.** Clean up the code (and the tests) while everything stays green, leaning on the tests to catch anything the cleanup breaks.

Why it helps:

- The test pins down what the function *should* do before you are invested in what it happens to do, turning requirements into executable checks.
- Every unit of behavior ends up covered by construction, so later changes get regression protection for free.
- Code written to be testable tends toward small, single-purpose functions with explicit inputs and outputs — the same design the function-decomposition guidance (Section) pushes toward.

You do not have to practice strict TDD to benefit from it. The testing guidance (Section) already applies its core move — establish tests *before* modifying existing functions — and writing the test first is just that same discipline applied to new code as well. In R packages, the mechanics are light: `usethis::use_test()` creates the matching `{testthat}` test file, and `devtools::test()` runs the suite (see R Packages, ch. "Testing basics").

Benchmarking

> **Note**
>
> This section draws from the Measuring Performance and Improving Performance chapters in *Advanced R* by Hadley Wickham, along with documentation for `{bench}` and `{profvis}`.

Performance optimization starts with measurement. Benchmarking helps identify bottlenecks, compare alternative implementations, and ensure your code meets performance requirements. This section covers when and how to benchmark R code, with a focus on integration with package development workflows.

When to Benchmark

Benchmark when:

- **Choosing between implementations**: Compare different approaches to the same problem
- **Optimizing critical paths**: Identify and improve performance bottlenecks
- **Preventing regressions**: Ensure new code doesn't slow down existing functionality
- **Processing large datasets**: Verify scalability for biostatistics/epidemiology workflows
- **Implementing computational methods**: Test algorithms involving bootstrapping, simulation, or resampling

Don't benchmark prematurely. Write correct, readable code first, then measure to find what actually needs optimization.

Setting Up Benchmarking Infrastructure

Organize benchmarking code separately from unit tests:

```
mypackage/
  tests/
    testthat/          # Unit tests (run by R CMD check)
    benchmarks/        # Benchmarking scripts (run manually or in CI)
      01-data-prep.R
      02-analysis.R
```

Add benchmarking dependencies to `DESCRIPTION`:

```r
# Add bench and profvis as development dependencies
usethis::use_package("bench", type = "Suggests")
usethis::use_package("profvis", type = "Suggests")

# Add any packages used in benchmarks
# For example, if comparing with data.table or gbm:
# usethis::use_package("data.table", type = "Suggests")
# usethis::use_package("gbm", type = "Suggests")
```

Create a benchmark template:

```r
# tests/benchmarks/01-data-prep.R
library(bench)
library(mypackage)

# Generate test data
n <- 10000
test_data <- data.frame(
  id = 1:n,
  exposure = rnorm(n),
  outcome = rbinom(n, 1, 0.3)
)

# Compare implementations
results <- bench::mark(
```

```
  original = prep_data_v1(test_data),
  optimized = prep_data_v2(test_data),
  check = FALSE,  # Set TRUE to verify outputs are identical
  iterations = 100,
  time_unit = "ms"
)

print(results)
```

Using bench for Performance Comparisons

{bench} provides accurate performance measurements and makes it easy to compare multiple implementations.

Basic usage:

```
library(bench)

# Compare different approaches
results <- bench::mark(
  base_subset = data[data$group == "treatment", ],
  dplyr_filter = dplyr::filter(data, group == "treatment"),
  data.table = dt[group == "treatment"],
  iterations = 100
)

# View results
print(results)
#> # A tibble: 3 x 6
#>   expression        min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 base_subset    1.2ms    1.4ms       714.    1.5MB     2.14
#> 2 dplyr_filter   2.1ms    2.3ms       435.    2.1MB     4.35
#> 3 data.table   850.0us  950.0us      1053.  800.0KB     0.00

# Plot results
plot(results)
```

Key features:

- **Accurate timing**: Accounts for overhead and runs multiple iterations
- **Memory tracking**: Shows memory allocation for each approach

- **Garbage collection**: Reports GC frequency
- **Output verification**: Optional checking that all implementations produce identical results

Example for biostatistics workflow:

```
# Compare propensity score estimation methods
library(bench)

# Simulate cohort data
n <- 5000
cohort <- data.frame(
  age = rnorm(n, 50, 15),
  bmi = rnorm(n, 27, 5),
  treatment = rbinom(n, 1, 0.5),
  outcome = rbinom(n, 1, 0.3)
)

# Note: Install gbm first if not already installed
# usethis::use_package("gbm", type = "Suggests")

# Compare GLM vs GBM for propensity scores
ps_benchmark <- bench::mark(
  glm_method = {
    model <- glm(treatment ~ age + bmi,
                 data = cohort,
                 family = binomial())
    predict(model, type = "response")
  },
  gbm_method = {
    model <- gbm::gbm(treatment ~ age + bmi,
                      data = cohort,
                      distribution = "bernoulli",
                      n.trees = 100,
                      verbose = FALSE)
    predict(model, n.trees = 100, type = "response")
  },
  check = FALSE,
  iterations = 50
)

print(ps_benchmark)
```

Using profvis for Memory and CPU Profiling

{profvis} identifies where your code spends time and allocates memory. Use it to find bottlenecks in existing functions.

Basic usage:

```
library(profvis)

# Profile a function
profvis({
  data <- prep_study_data(raw_data)
  results <- run_primary_analysis(data)
})
```

The profvis output shows:

- **Flame graph**: Visual representation of time spent in each function
- **Data view**: Line-by-line time and memory usage
- **Memory profile**: Allocation patterns over time

Example for data preparation:

```
library(profvis)

profvis({
  # Profile data cleaning pipeline
  cleaned <- raw_data |>
    dplyr::filter(!is.na(id)) |>
    dplyr::mutate(
      age_group = cut(age, breaks = c(0, 18, 65, Inf)),
      bmi_category = cut(bmi, breaks = c(0, 18.5, 25, 30, Inf))
    ) |>
    dplyr::left_join(exposure_data, by = "id") |>
    dplyr::group_by(age_group) |>
    dplyr::summarize(
      n = n(),
      mean_bmi = mean(bmi, na.rm = TRUE)
    )
})
```

Interpreting profvis output:

- **Wide bars**: Functions taking substantial time
- **Tall stacks**: Deep call chains (potential for simplification)
- **Memory spikes**: Large allocations (consider chunking or data.table)

Common bottlenecks in epidemiology code:

- Repeated subsetting operations -> Use data.table or pre-filter
- Growing objects in loops -> Pre-allocate vectors
- Complex joins on large datasets -> Index properly or use data.table
- Unnecessary copies -> Use reference semantics where appropriate

Integration with Package Development Workflow

Integrate benchmarking into your development cycle:

1. Benchmark before optimization:

```
# tests/benchmarks/baseline.R
# Run this before making changes to establish baseline

library(bench)
library(mypackage)

baseline <- bench::mark(
  current_implementation = analyze_cohort(test_data),
  iterations = 100
)

saveRDS(baseline, "tests/benchmarks/baseline.rds")
```

2. Profile to identify bottlenecks:

```
library(profvis)

profvis({
  analyze_cohort(test_data)
})
# Identify which functions are slow
```

3. Optimize and re-benchmark:

```r
# After optimization
library(bench)

baseline <- readRDS("tests/benchmarks/baseline.rds")

comparison <- bench::mark(
  baseline = analyze_cohort_v1(test_data),
  optimized = analyze_cohort_v2(test_data),
  check = FALSE,
  iterations = 100
)

print(comparison)
```

4. Document performance characteristics:

```r
#' Analyze cohort data
#'
#' @param data A data frame with cohort information
#' @return Analysis results
#'
#' @details
#' Performance characteristics (as of 2025-01):
#' - Typical runtime: ~50ms for 10,000 observations
#' - Memory usage: ~5MB per 10,000 observations
#' - Scales linearly with sample size
#'
#' @examples
#' \dontrun{
#' results <- analyze_cohort(cohort_data)
#' }
#' @export
analyze_cohort <- function(data) {
  # Implementation
}
```

CI/CD Integration for Automated Benchmarking

Set up GitHub Actions to track performance over time.

Create `.github/workflows/benchmark.yaml`:

Key considerations for CI benchmarks:

- **Variability**: CI runners have variable performance; use thresholds (e.g., >10% regression)
- **Baseline storage**: Commit baseline results or use GitHub Actions artifacts
- **Selective running**: Only benchmark on specific branches or when performance-critical files change
- **Manual triggers**: Use `workflow_dispatch` for on-demand benchmarking
- **Security**: For production workflows, consider pinning action versions to commit SHAs instead of tags (see the wai site's "Best Practices for Safe and Successful Use" section)

Alternative: Comment benchmark results on PRs:

Use the `{touchstone}` package for more sophisticated CI benchmarking with automated PR comments.

Performance Testing with testthat

For critical performance requirements, add performance tests to your test suite.

Example performance test:

```
# tests/testthat/test-performance.R

test_that("data preparation meets performance requirements", {
  skip_on_cran()  # Skip on CRAN (timing-based tests can be flaky)

  # Run on CI only if BENCHMARK_ON_CI is set
  if (identical(Sys.getenv("CI"), "true") &&
      !identical(Sys.getenv("BENCHMARK_ON_CI"), "true")) {
    skip("Skipping performance test on CI")
  }

  n <- 10000
  test_data <- generate_test_cohort(n)

  # Measure execution time
  timing <- bench::mark(
    prep_study_data(test_data),
    iterations = 10,
    check = FALSE
  )
```

```
  # Require median time under threshold
  max_time_ms <- 100
  median_time_ms <- as.numeric(timing$median) * 1000

  expect_true(
    median_time_ms < max_time_ms,
    info = sprintf(
      "prep_study_data took %.1f ms (threshold: %.1f ms)",
      median_time_ms,
      max_time_ms
    )
  )
})

test_that("analysis scales linearly with sample size", {
  skip_on_cran()

  # Run on CI only if BENCHMARK_ON_CI is set
  if (identical(Sys.getenv("CI"), "true") &&
      !identical(Sys.getenv("BENCHMARK_ON_CI"), "true")) {
    skip("Skipping performance test on CI")
  }

  # Test at different sample sizes
  sizes <- c(1000, 5000, 10000)

  timings <- vapply(sizes, function(n) {
    data <- generate_test_cohort(n)
    result <- bench::mark(
      analyze_cohort(data),
      iterations = 10
    )
    as.numeric(result$median)
  }, numeric(1))

  # Check approximate linearity (R^2 > 0.95)
  model <- lm(timings ~ sizes)
  r_squared <- summary(model)$r.squared

  expect_true(
    r_squared > 0.95,
    info = sprintf("Scaling R^2 = %.3f (expected > 0.95)", r_squared)
```

```
  )
})
```

When to use performance tests:

- Functions with documented performance requirements
- Critical paths that must stay fast
- Code that processes large datasets

When not to use performance tests:

- Functions without specific performance needs
- Tests that would be flaky due to system variability
- Development environments with limited resources

See Section for more on testing with {testthat}.

Best Practices

Focus benchmarks on realistic scenarios:

```
# Good: Realistic data size
n <- 50000  # Typical cohort size in a large study
test_data <- generate_realistic_cohort(n)

# Avoid: Unrealistically small data
n <- 10
```

Establish baselines:

Before optimizing, measure current performance to understand the improvement.

```
# Document baseline
baseline <- bench::mark(current_implementation(data), iterations = 100)
saveRDS(baseline, "benchmarks/baseline-2025-01.rds")
```

Compare on equal footing:

```r
# Good: Same random seed, same data
set.seed(123)
data <- generate_test_data(10000)

bench::mark(
  method_a = analyze_a(data),
  method_b = analyze_b(data),
  check = FALSE
)

# Avoid: Different random data
bench::mark(
  method_a = analyze_a(generate_test_data(10000)),
  method_b = analyze_b(generate_test_data(10000))
)
```

Benchmark critical paths only:

Don't optimize everything—focus on code that:

- Runs frequently
- Processes large datasets
- Is called in loops or simulations
- Has noticeable user-facing delays

Use appropriate sample sizes:

```r
# For data prep: use typical dataset size
cohort_data <- generate_cohort(n = 50000)

# For simulation: use realistic iteration counts
n_iterations <- 1000
```

Document optimization decisions:

```r
# In package documentation or vignette

# We use data.table for joins because:
# - Benchmarks show 5x speedup over dplyr for n > 100,000
# - Typical cohort sizes: 50,000 - 500,000 observations
# - See tests/benchmarks/join-comparison.R for details
```

Additional Resources

- {bench} - Accurate benchmarking
- {profvis} - Interactive profiling
- {memoise} - Function-call caching to eliminate repeated bottlenecks (Section)
- Measuring Performance chapter in Advanced R
- Improving Performance chapter in Advanced R
- {touchstone} - CI benchmarking with PR comments

Iterative Operations

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

`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)
```

`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
```

`base::Vectorize()` vs `purrr::map_*()`

`base::Vectorize()` and {purrr} solve different problems.

- Use `base::Vectorize()` when you already have a scalar function and want a quick base-R wrapper that accepts vector inputs.
- Use `purrr::map_*()` when you are iterating over lists or list-columns and want explicit, type-stable outputs (`map_dbl()`, `map_int()`, etc.) that compose naturally in tidyverse pipelines.

`base::Vectorize()` is often convenient for small, local wrappers. In contrast, {purrr} is usually more idiomatic for analysis workflows because output types are explicit and iteration intent is clearer.

Also note that neither approach is parallel by default. For parallel execution, use `furrr::future_map()` from {furrr}.

`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
  )
})
```

Parallel Processing

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

```r
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)
```

Reading and Saving Data

RDS Files (Preferred)

Use RDS format for R objects:

```r
# 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. The base R function `save()` serializes several objects into one `.RData` file, and `load()` restores every object in it under its original name. That causes three problems:

- You can't control object names when loading.
- You can't load individual objects.
- Older code becomes hard to follow, because it is not clear which objects a `load()` call creates.

> **Note**
>
> If you have many related R objects that you would otherwise save together with `save()`, put them in a named list and save that list as one RDS file.

CSV Files

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

```r
# 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, here("data-raw", "big_file.csv"))
data <- data.table::fread(here("data-raw", "big_file.csv"))
```

`readr::read_csv()` and `readr::write_csv()` are much faster than base R's `read.csv()` and `write.csv()`. For very large files (several gigabytes), `data.table::fread()` and `data.table::fwrite()` are faster still.

Version Control and Collaboration

Git Branches and Tags

To understand how Git tracks history, think of commits as Polaroid snapshots of your repository connected to their parent snapshots by strings. Branches and tags are both labels (like sticky notes) pointing to specific snapshots in this history graph:

- **Branches are movable labels**: A branch label points to the latest commit in a line of development. When you make a new commit while working on a branch, Git automatically advances the branch label forward to point to the new commit. Think of a branch as a sticky note that moves automatically as history grows.
- **Tags are permanent labels**: A tag marks a specific point in time (such as a published release like `v1.0.0` or a manuscript submission). Tags do not move when new commits are added; they remain anchored to their specific commit. Git provides two types of tags: *lightweight tags* (simple references directly pointing to a commit) and *annotated tags* (full Git objects storing a tagger identity, timestamp, and release message; recommended for official versions and public releases).

For deeper background on Git's object model and branching mechanisms, consult the official Git Branching in a Nutshell and Git Tagging guides in *Pro Git* by Scott Chacon and Ben Straub.

Practical branch and tag commands

```
# Create and switch to a new feature branch
git switch -c feat/new-analysis

# Publish the feature branch to GitHub and set upstream tracking
git push -u origin feat/new-analysis

# View all local and remote-tracking branches
git branch -a

# Create a lightweight tag at the current commit
git tag v0.1.0

# Create an annotated tag with a release message (recommended for releases)
git tag -a v1.0.0 -m "Release version 1.0.0"

# Push a specific tag to GitHub
git push origin v1.0.0

# Push reachable annotated tags alongside branch commits (safer than pushing all tags)
git push --follow-tags
```

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`)

> **!** Keep the development version ahead
>
> Always keep the development version ahead of the main branch version. When working on a development branch, ensure the version in `DESCRIPTION` is higher than the version on `main`.

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

See R Packages - Version numbers for full guidelines.

NEWS File

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

Formatting conventions:

- Use `(#123)` notation to link to related issues and pull requests (GitHub uses a shared number space for both)
- Use `@username` to credit **external** contributors only (not internal team members)

```
# myproject 0.2.0

## New features

- Added function for data validation (#45).
- Improved error messages (#52, @external-contributor).

## Bug fixes

- Fixed issue with missing values (#38).
- Corrected calculation error in summary stats (#41).
```

See R Packages - NEWS.md for full conventions.

README.Rmd and README.md

If your project uses `README.Rmd` (a README with executable R code), **always edit `README.Rmd`, never `README.md` directly.** `README.md` is auto-generated from `README.Rmd`.

To regenerate `README.md` after editing `README.Rmd`, use `devtools::build_readme()` (preferred for R packages):

```
devtools::build_readme()
```

Alternatively, you can use {rmarkdown}'s `rmarkdown::render("README.Rmd")` if {devtools} is not available.

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); run `pkgload::load_all()` first so `object_usage_linter()` checks current source
- ☐ `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 via `devtools::build_readme()`
- ☐ `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)

Automated Code Styling

RStudio Built-in Formatting

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

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
```

lintr Package

For checking code style without modifying files:

```
# Install lintr (and pkgload, used below)
install.packages(c("lintr", "pkgload"))

# For package code, load the package first so object_usage_linter()
# checks your current source (see the style guide for details)
pkgload::load_all()

# 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.

jarl (Fast Rust-Powered Linter)

`{jarl}` is a high-performance R linter written in Rust by Etienne Bacher. It checks many common style and syntax rules significantly faster than `{lintr}`, making it well-suited for fast pre-commit hooks and large codebases.

See also Automated Tools.

Documenting your code

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

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().
#'
#' @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`:

```
#' 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
```

```r
#'
#' @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}` package to catch misspelled argument names:

   ```r
   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.

Documenting internal helpers: `@keywords internal` over `@noRd`

Document every function you write, including internal helpers that only your other package functions call. The question is *how* the documentation is exposed, not whether it exists.

A helper is kept internal by omitting `@export`, which leaves it out of the `export()` directives in `NAMESPACE` (the function is still in the package namespace, which is why `pkg:::fn` works). Once the function is non-exported, two roxygen tags control how its documentation is generated and indexed, and they behave differently:

Table 2: Comparison of `@noRd` and `@keywords internal` roxygen tags

Tag	Generates an `.Rd` file?	Appears in pkgdown reference index?	Accessible via `?pkg:::fn`?
`@noRd`	No	No	No
`@keywords internal`	Yes	No	Yes

Prefer `@keywords internal` for any helper that isn't part of the package's public API.

For more on these tags, see the roxygen2 documentation and R Packages (2e): Function documentation. You still get a real `.Rd` file — so future maintainers (or future-you) can run `?pkg:::internal_helper` and see the documentation — but the function stays out of the pkgdown reference and the package's main help page.

`@noRd` is for cases where you genuinely don't want documentation to exist at all (e.g., a one-line internal shim where the docstring would be longer than the function). Reach for it sparingly.

Example:

```
#' Compute the per-row score
#'
#' Internal worker for [public_summary()].
#'
#' @param row A one-row data frame.
#' @returns A numeric scalar.
#'
#' @keywords internal
score_one_row <- function(row) {
  ### function code here
}
```

> **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).

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.

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

Sections and subsections

RStudio (v1.4 or more recent) supports the use of sections and subsections. You can easily navigate through longer scripts using the navigation pane in RStudio. Create sections and subsections with comment lines like these:

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

## Subsection -------
```

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

Code folding

Consider using RStudio's code folding 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. The section headers shown above are examples.

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 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
```

Variable labels

Add a descriptive label as soon as you create each key variable (any variable that might end up in a figure, table, or output model). The label should explicitly include measurement units where applicable.

In R, prefer the {labelled} package over raw base R attributes. {labelled} integrates seamlessly with tidyverse pipelines:

```r
library(labelled)

# Set variable labels directly in a pipeline
data <- data |>
  set_variable_labels(
    blood_lead_level = "Blood lead concentration (ug/dL)",
    pm25_exposure = "Annual average PM2.5 exposure (ug/m^3)"
  )

# Set a single variable label using var_label()
var_label(data$blood_lead_level) <- "Blood lead concentration (ug/dL)"
```

Labeling key variables at the point of creation ensures that downstream table generators (such as `{gtsummary}`) can automatically display human-readable descriptions and correct units without error-prone manual relabeling.

See also Function Structure and Documentation for function documentation style guidelines.

Design Documentation with ADRs

An Architecture Decision Record (ADR) is a short, versioned document that captures a significant design decision: the context that forced the choice, the alternatives considered, the decision itself, and its consequences. Function-level documentation and inline comments explain *code*; an ADR explains a *decision* that shapes many files at once and would otherwise live only in someone's memory or a long-merged pull request thread.

When to write one

Write an ADR for a non-obvious design choice that affects multiple files or systems: a concurrency or security trade-off, a data layout or dependency choice with long-term consequences, or a naming convention that needs justification. Routine choices that any reader would make the same way do not need one.

Recommended format

Use the four-part template from Michael Nygard's "Documenting Architecture Decisions":

- **Status**: proposed, accepted, deprecated, or superseded (with a pointer to the successor).
- **Context**: the forces at play — what problem existed and what constraints applied.
- **Decision**: what was chosen, stated in full sentences with an active voice ("We will …").
- **Consequences**: what becomes easier, what becomes harder, and what future work the decision commits you to.

Where to store them

Keep ADRs in the repository they describe, as `docs/adr-<topic>.md` files (or a `docs/decisions/` directory once there are more than a few), so they are versioned, reviewable, and greppable alongside the code.

How to reference them from code

Point to the ADR with a terse inline comment — `# see docs/adr-<topic>.md, "<section>" section` — instead of duplicating the rationale in comments. This keeps scripts readable while making the full reasoning one hop away, and it means the rationale has one home to update when the decision changes.

Relationship to pull request descriptions

A PR description explains *why this change, now*, and effectively disappears from view once the PR merges; an ADR persists in the repository and keeps answering "why is it built this way?" long after the merge. When a PR implements a significant decision, write the ADR in that PR and let the PR description summarize and link to it.

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

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 Object Naming.

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:

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

Modular design

Modular programming splits a program into separate **modules**, each of which does one job and can be understood, tested and changed on its own. In R and Python, a module can be a function, a file or a package. Splitting a problem into such parts is called decomposition. Guides to scientific computing recommend it: Wilson et al. (2014) advise modularizing code rather than copying and pasting it, and Wilson et al. (2017) advise decomposing programs into functions.

Each module has an **interface** and an **implementation**. The interface is what other code may rely on: for an R function, its name, arguments and return value; for a package, its exported functions (Wickham and Bryan 2023). The implementation is everything else. Code that calls a module depends only on its interface, so the smaller the interface, the more of the module you can change without changing that code.

Two properties describe how well a program is divided (Stevens et al. 1974):

- **Cohesion** is how closely the parts inside one module belong together. Aim for high cohesion: a file that cleans one dataset is more cohesive than a `utils.R` file that holds every helper in the project.
- **Coupling** is how strongly one module depends on another, for example on its internal details or on shared global state. Aim for low coupling: a script that calls a function from another file is less coupled than one that `source()`s that file and then uses the intermediate objects it created.

Decide what each module hides

Parnas (1972) described two ways to split the same program into modules. One split followed the steps of the program in order. The other gave each module one design decision that was likely to change, such as a file format or a data structure, and hid that decision behind the module's interface. Parnas argued that the second split is easier to change, because a change to one of those decisions touches only one module. This approach is called **information hiding**.

In a data analysis, decisions that often change include:

- the format and location of the raw data;
- how a variable is coded or cleaned;
- which model is fitted.

Give each decision its own function or file. For example, suppose that only `read_survey_data()` knows that the survey arrives as a CSV file:

```
read_survey_data <- function(path) {
  readr::read_csv(path, show_col_types = FALSE)
}
```

If the survey later moves into a database, you change only `read_survey_data()`, and the code that calls it keeps working:

```
raw_data <- read_survey_data(here::here("data", "survey.csv"))
clean_data <- clean_survey_data(raw_data)
model <- fit_coverage_model(clean_data)
```

When to make something a module

A useful test is whether you can state the code's job in one sentence and name its inputs and outputs. If you can, it is a candidate for its own function. Move a set of functions into a package only once several projects use them, since a package adds documentation, tests and releases to maintain (see Complete Package Development Workflow).

Function decomposition

Function decomposition improves code readability and maintainability by breaking complex logic into focused, reusable units. When a function becomes too long or handles too many concerns, split it into smaller helper functions with clear responsibilities.

Key goals of decomposition include:

- Breaking a big problem into smaller subproblems that are easier to reason about
- Making code easier to read, review, and test
- Pushing low-level complexity "downward" into helper functions so the main workflow stays clear

When to decompose a function

Consider decomposition when one or more of these are true:

- A function is doing multiple jobs (validation, transformation, modeling, and formatting)
- Most of the function body is a single `if-else`; split the two cases into separate functions so each path has a clear name and contract
- You need comments to explain every few lines just to follow the logic
- You are repeating the same block in multiple places
- You want to test one part of a workflow independently of the rest
- The current function is hard to scan because implementation details hide the main idea

How to decompose in practice

A practical sequence is:

1. Write down the top-level steps in plain language.
2. Extract one step at a time into a helper function.
3. Give each helper a single, specific contract (inputs, outputs, and assumptions).
4. Keep the orchestration function short, so it reads like a workflow outline.
5. Add or update tests around each helper and the top-level function.

Top-down and bottom-up design

The sequence in How to decompose in practice starts from the top: name the top-level steps first, then write a function for each. This is top-down design, also called stepwise refinement, and a step that is still too large can be refined the same way. The opposite is bottom-up design: write and test the small pieces first, then combine them into larger functions until

the top-level workflow exists. See Bottom-up and top-down approaches for more on both approaches, including how top-down design spread in software in the 1970s.

Each approach has a strength and a typical risk:

- Top-down design keeps the overall workflow clear, but the lowest-level functions are written and tested last, so problems in them surface late. Writing a short stub for each step, a placeholder that returns a fixed value, lets the top-level function run and be tested early.
- Bottom-up design gets working, tested pieces quickly, and those pieces are easy to reuse. But the pieces may not fit together, because they were written before the workflow that calls them.

Most software projects combine the two. Sketch the top-level workflow first, so the pieces have a known purpose and interface, and reuse existing functions, from your own code or from packages, wherever one already does a step.

Decomposing validation blocks

A common case for decomposition is a function that opens with a long sequence of input-validation checks. When the same validation pattern repeats for multiple inputs, extract it into a helper:

```
# Before: repeated validation blocks inline
build_analysis_cache <- function(
    rebuild_signal, signal_files, signal_dir,
    rebuild_finance, finance_files, finance_dir,
    ...) {
  if (rebuild_signal && length(signal_files) == 0L) {
    cli::cli_abort(c(
      "Cannot rebuild Signal source data.",
      "x" = "No raw files found in {.path {signal_dir}}.",
      "i" = "Use {.val auto} to reuse an existing RDS."
    ))
  }
  if (rebuild_finance && length(finance_files) == 0L) {
    cli::cli_abort(c(
      "Cannot rebuild finance source data.",
      "x" = "No raw files found in {.path {finance_dir}}.",
      "i" = "Use {.val auto} to reuse an existing RDS."
    ))
  }
  # ... same pattern for mapping, then more for cached-RDS checks ...
```

```
}

# After: extract the repeated pattern into a helper
validate_source_rebuild <- function(
    rebuild,
    source_files,
    label,
    search_path,
    call = rlang::caller_env()) {
  if (!rebuild || length(source_files) > 0L) return(invisible(NULL))
  cli::cli_abort(
    c(
      "Cannot rebuild {label} source data.",
      "x" = "No raw {label} files were found in {.path {search_path}}.",
      "i" = "Use {.val auto} to reuse an existing source RDS."
    ),
    call = call
  )
}

build_analysis_cache <- function(
    rebuild_signal, signal_files, signal_dir,
    rebuild_finance, finance_files, finance_dir,
    rebuild_mapping, mapping_files, s_drive_path,
    ...) {
  validate_source_rebuild(rebuild_signal, signal_files, "Signal", signal_dir)
  validate_source_rebuild(rebuild_finance, finance_files, "finance", finance_dir)
  validate_source_rebuild(rebuild_mapping, mapping_files, "mapping", s_drive_path)
  # ... main logic is now immediately visible
}
```

Benefits of this pattern:

- The main function reads as a workflow outline rather than a wall of `if`/`cli_abort` blocks.
- Each validation rule is defined once and tested once.
- Adding a new data source requires one new call, not a copy-pasted block.
- Error message formatting stays consistent across all sources.

Place each extracted helper in its own file (Section), not above or below the caller in the same file.

Examples borrowed from package PRs

In `UCD-SERG/rpt` PR #100, a top-level function was updated to call dedicated helpers, with helper logic split into separate files. The workflow moved from "do everything in one function" toward a clearer sequence of validation, cleaning, calculation, and formatting steps. This made the top-level function easier to read, while isolating validation and formatting details in reusable units.

In `UCD-SERG/shigella` PR #11, focused analysis functions were added in separate files, moving away from larger script-style helpers toward single-purpose, testable package functions.

Related references

For background and related ideas, see Modular design, Code refactoring, and Functional decomposition.

One function per file

Give each function its own file rather than bundling multiple functions into one file. This applies to exported functions, internal helpers, and test helpers alike — the only exception is a trivial two-line wrapper that is inseparable from its caller.

R packages

Under `R/`, name the file after the function it contains: `simulate_data()` lives in `R/simulate_data.R`. A private helper that is more than a two-liner still gets its own file, named after the helper, even if only one caller uses it.

Why

A file with one function is easier to find, easier to diff (a change to one function does not sit inside an unrelated grep hit for a different one), and matches how `git blame` and code-review tools attribute changes. Bundling unrelated functions into one file — because they were added in the same PR, or because an existing file happened to be open — accumulates unrelated concerns over time.

Adding to an existing multi-function file

When adding a function to a file that already holds others (common in older code that predates this convention), put the **new** function in its own file so at least new code follows the convention going forward. Splitting every pre-existing function out is a separate, larger refactor.

Relationship to function decomposition

When decomposing a large function (Section), place each extracted helper in its own file rather than above or below the caller in the same file. The caller file then contains only the orchestration function, and each helper is independently findable and testable.

Function length limits

Function length matters because very long functions can hide workflow intent, increase maintenance burden, and make targeted testing harder. The sources below combine community discussions (opinion-based) with a formal complexity reference. Use the community links for examples of practitioner viewpoints, and the complexity reference for a formal metric.

Sources:

- Reddit (r/learnprogramming): February 2016 community thread asking whether one long function or many smaller functions is preferable.
- SoloLearn discussion: commenters describe different line-count conventions (including 20-line and 100-line limits) and discuss readability and testability tradeoffs.
- Stack Overflow: "When is a function too long?" and Software Engineering Stack Exchange: "What is the ideal length of a method for you?": older Q&A threads (September 2008 and February 2012, respectively) focused on practical criteria for deciding when to split methods.
- Cyclomatic complexity: defines complexity by counting independent execution paths/decision points; this reinforces that branching complexity is often a better risk indicator than raw line count alone.

Synthesizing these viewpoints for research code yields the following practical rules:

1. Use `<150` lines as a provisional heuristic trigger to reassess decomposition, not a hard constraint.
2. Prefer single-purpose functions with clear names and contracts.
3. Watch decision complexity (branching and nesting), not only length.
4. Decompose when the top-level story is hard to follow or hard to test.

Keep It Short and Simple (KISS)

The KISS principle — "Keep It Short and Simple" — says that most systems work best when they are kept simple, so unnecessary complexity should be avoided rather than managed. A blunter phrasing — the one aircraft engineer Kelly Johnson actually coined at the Lockheed Skunk Works — is "keep it simple, stupid": his standing challenge was that his team's aircraft must be repairable by an average mechanic in the field with only a handful of common tools.

The principle carries directly into software development: code is read, debugged, and modified far more often than it is written, and it is usually maintained by someone with less context than the original author had — often your own future self.

Rationales for keeping code short and simple:

- **Fewer bugs.** Less code and simpler control flow leave fewer places for defects to hide and make the remaining logic easier to reason about.
- **Easier review.** A reviewer can hold a short, single-purpose function in their head; they cannot do that with a sprawling one.
- **Cheaper maintenance.** Simple code can be modified confidently; complex code makes every change risky and slow.
- **Faster onboarding.** New team members become productive sooner when the codebase does things the plain, expected way.

Simplicity is not an excuse to drop requirements: the goal is the simplest solution that still does the whole job, not a simpler job. The preceding sections on function decomposition and function length limits, and the guidance on avoiding deep nesting, are concrete applications of this principle.

Defense in Depth

Defense in depth is the practice of protecting an important guarantee with several independent layers, so that when one layer fails, another still catches the problem. The name comes from security engineering, where a system is protected by multiple countermeasures rather than a single wall. Reason (2000) described the same idea for accidents in complex systems as the "Swiss cheese" model: each defense has holes, like a slice of Swiss cheese, and an accident happens only when the holes in every layer line up. Adding a layer does not remove the holes in the others; it makes it less likely that a problem passes through all of them.

In research software, the guarantees worth layering include:

- that results can be reproduced from the code and data;
- that code merged to the main branch passes its tests;
- that private data never reaches a public repository.

Each of these can be guarded by several layers:

- **Prevention** before the mistake, such as a type or argument check, or a pre-commit hook.
- **Detection** soon after it, such as a test or a lint check that runs on every push.
- **Review**, in which a person or a separate tool reads the change.
- **Recovery** after it, such as version control, which lets you revert a bad change.

Make the layers independent

Layers protect only if they do not fail for the same reason. Two checks that run the same script, read the same configuration, or depend on the same service fail together, and count as one layer. Prefer layers that differ in mechanism (a test, a linter, a reviewer), in timing (before a commit, in CI, at review), and in who or what operates them.

Test each layer on its own

A layer that fails silently is worse than no layer, because it makes the guarantee look protected when it is not. Test each layer separately from the others: give it an input it must catch, and confirm that it does. For example, to test a linter in CI, push a branch with a deliberate violation and confirm that the check fails. Do this when the check is first added and again whenever it changes, since a configuration error often turns a check into one that always passes.

A check that reports success should also show that it ran. A check that finds nothing because it inspected nothing (an empty file list, a wrong path, a skipped step) looks the same as a check that found nothing wrong. Have each layer fail, or say so loudly, when it has nothing to check, in keeping with the principle of failing fast (see Assertions and Input Validation).

Layers have a cost

Each layer costs time to build, maintain and wait for, and several layers guarding a trivial guarantee are over-engineered (see KISS). Reserve defense in depth for guarantees whose failure is costly or hard to notice, such as a wrong result in a published analysis.

Decomposing Code Chunks in Quarto Documents

A Quarto document should read as narrative plus orchestration: each code chunk calls documented functions and displays their results, and the surrounding prose carries the analysis story. Complex logic belongs in named functions, not inline in chunks.

When a chunk grows into substantial logic — a multi-step data manipulation, a function defined inline, a non-trivial transformation, or the same block copied across chunks — extract

that logic into a function and have the chunk call it. Following the function-decomposition (Section), one-function-per-file (Section), and length-limit (Section) principles, give the function its own file under `R/`, with full `{roxygen2}` documentation, the same way any other package function is written. The chunk then shrinks to a call plus whatever narrative or display wraps it.

Why this matters: a function can be unit-tested, reused across chunks and analyses, and documented; inline chunk code can be none of these. Keeping logic inline also buries the analysis narrative under implementation detail, and duplicates code that should have one home.

The UCD-SERG shared linter set enforces the function-definition half of this rule: `lms::function_location_linter()` (part of `lms::default_linters()`) flags a named function defined at the top level of any file outside the designated directories (`R/` and `tests/` by default), including inside a `.qmd` chunk. Loose chunk logic that was never wrapped in a function at all is not something a linter can catch, so watch for it when writing and reviewing analyses: a chunk carrying substantial inline logic is a decomposition finding, and the fix is to move the logic into `R/` and leave the chunk calling it.

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 here is a great vignette describing it.

See also The here Package for code style guidelines on using the here package.

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.

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 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.

Dropbox

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

r-lib Packages

The r-lib GitHub organization, maintained primarily by the Posit team, hosts a collection of foundational R packages for package development, infrastructure, and general programming. These packages underpin much of the modern R ecosystem.

Package Development

{usethis}

{usethis} automates repetitive tasks that arise during project setup and development, both for R packages and non-package projects. It provides functions for setting up testing, CI, licenses, Git, and GitHub integrations, making it the go-to companion for `devtools`.

{devtools}

{devtools} provides a suite of tools for the R package development workflow. It wraps many lower-level tools such as `pkgbuild`, `pkgload`, and `rcmdcheck` behind a convenient interface. Key functions include `load_all()`, `document()`, `test()`, `check()`, and `install()`.

{roxygen2}

{roxygen2} lets you write documentation in-source, alongside the code it documents, using `#'` comment blocks. It generates `.Rd` files for the `man/` directory and updates `NAMESPACE`. It is the standard approach to R package documentation.

{pkgdown}

{pkgdown} builds a documentation website for an R package from its source. It converts `README`, vignettes, and function documentation into a navigable website, and integrates with GitHub Actions for automatic deployment.

{pkgload}

{pkgload} simulates the process of installing and loading a package, enabling rapid interactive development. It is the backend for `devtools::load_all()`.

{pkgbuild}

{pkgbuild} provides tools for building R packages, finding compilers, and checking whether a build environment (such as Rtools on Windows) is available.

{rcmdcheck}

{rcmdcheck} runs `R CMD check` from within R and captures the results in a structured format. It is used by CI systems and `devtools::check()`.

{remotes}

{remotes} installs R packages from remote sources including GitHub, GitLab, Bitbucket, URLs, and local files. It is a lightweight alternative to `devtools::install_github()`.

{pak}

{pak} installs R packages from CRAN, GitHub, and other sources, with fast parallel downloads and automatic system-dependency handling. It is r-lib's modern alternative to `install.packages()` and `remotes` for interactive use.

{revdepcheck}

{revdepcheck} automates reverse dependency checking — checking that changes to a package do not break packages that depend on it. It is used by package maintainers before CRAN releases.

{desc}

{desc} provides tools for reading, writing, and manipulating `DESCRIPTION` files. It is used internally by `usethis` and other developer-facing tools.

{pkgcache}

{pkgcache} provides a local cache of CRAN-like metadata and package files, speeding up package installation. It is used internally by `pak`.

{lifecycle}

{lifecycle} manages the life cycle of functions and arguments, providing the experimental/superseded/deprecated badges and deprecation warnings used across the tidyverse and r-lib ecosystems.

Testing and Quality

{testthat}

{testthat} is the most widely used R testing framework. It provides a readable, expressive way to write unit tests for R packages, with functions like `expect_equal()`, `expect_error()`, and `expect_snapshot()`.

{covr}

{covr} measures and reports test coverage for R packages. It can generate HTML reports, submit results to Codecov, and is commonly integrated into CI pipelines.

{waldo}

{waldo} finds and describes differences between R objects. It provides clearer, more informative output than `identical()` or `all.equal()`, and is used internally by `testthat` to display test failures.

{lintr}

{lintr} performs static code analysis (linting) for R. It checks for style issues, potential errors, and common anti-patterns, and integrates with RStudio, VS Code, and CI pipelines.

Core Language and Infrastructure

{rlang}

{rlang} provides a low-level API for working with R's core language features, including environments, quosures, symbols, and non-standard evaluation (tidy eval). It is a foundational dependency for tidyverse packages and tidy eval programming.

{vctrs}

{vctrs} defines a principled type system for R vectors, providing tools for coercion, casting, and size-stability. It underpins the consistent behavior of `dplyr` and `tidyr` across vector types.

{withr}

{withr} allows code to be run with temporarily modified global state — changing options, environment variables, working directories, or random seeds — and reliably restoring them afterward. Essential for writing side-effect-free tests.

{cli}

{cli} provides a rich toolkit for building command-line interfaces in R. It supports color, ANSI styling, progress bars, spinners, and formatted messages. It is the modern replacement for `crayon` and is used throughout the tidyverse.

{R6}

{R6} implements encapsulated, reference-based object-oriented programming in R. Unlike S3 or S4, R6 classes support mutable state and inheritance in a style familiar from Python or Java.

{cpp11}

{cpp11} provides a modern C++11 interface for interacting with R's C API. It is a lightweight, header-only alternative to `Rcpp`, designed to simplify writing and debugging C++ extensions for R.

System and Process

{fs}

{fs} provides a cross-platform, uniform interface to file system operations. It replaces inconsistent base R file functions with a tidy API using `path_*()`, `file_*()`, `dir_*()`, and `link_*()` functions that always return character vectors.

{processx}

{processx} runs external system processes asynchronously or synchronously from R, with full control over stdin, stdout, and stderr. It is the backend for `callr`, and is used internally by `rcmdcheck`.

{callr}

{callr} calls R functions in separate R processes. This is useful for running code in a clean environment, isolating side effects, or parallelizing work. It builds on top of `processx`.

{ps}

{ps} provides a cross-platform API for listing, querying, and manipulating system processes from R. It is used internally by `processx` and `callr`.

HTTP and Web

{httr2}

{httr2} is a modern, pipe-friendly HTTP client for R. It replaces `httr` with a cleaner API, better support for OAuth, rate limiting, retries, and request/response bodies. Preferred for new code.

{httr}

{httr} is a widely-used HTTP client for R, providing wrappers for `GET()`, `POST()`, `PUT()`, `DELETE()`, and other HTTP verbs, as well as OAuth support. It has been superseded by `httr2` for new development.

{gh}

{gh} provides a minimal client for the GitHub REST API. It handles authentication and pagination, and is used internally by `usethis` for GitHub operations.

{gargle}

{gargle} provides authentication utilities for Google APIs, including OAuth 2.0, service accounts, and Application Default Credentials. It is used by `googledrive`, `googlesheets4`, and other Google-backed R packages.

Utilities

{here}

{here} builds file paths relative to the project root, so scripts work regardless of the current working directory. We recommend `{here}` for file paths (see Section).

{memoise}

{memoise} adds memoisation (function-call caching) to R functions. Wrapping an expensive, pure function with `memoise::memoise()` causes it to cache computed return values in memory or on disk, so repeated calls with identical arguments return instantly from cache instead of recomputing. It supports multiple storage backends via the `cache` argument (such as `cachem::cache_mem()` for in-memory caching and `cachem::cache_disk()` for persistent disk caching across sessions), and provides `memoise::forget()` to clear caches, `memoise::is.memoised()` to test whether a function is memoised, and `memoise::has_cache()` to check whether specific arguments have cached results. Use `{memoise}` only for deterministic functions without side effects (such as pure mathematical evaluations, stable API queries, or idempotent data processing pipelines). Never memoise stochastic simulation routines or functions that generate pseudorandom numbers; memoisation suppresses stochastic variation across repeated calls, silently collapsing random variability by returning stale cached draws.

{scales}

{scales} converts data values into perceptual properties and formats labels for dates, currencies, and percentages; it powers the scale system of {ggplot2}.

{sessioninfo}

{sessioninfo} collects and prints detailed information about the current R session, including package versions and sources. It is a more informative replacement for base R's `sessionInfo()`.

{downlit}

{downlit} provides syntax highlighting for R code and automatically links function names to their documentation. It is used by `pkgdown` and Quarto for rendering R code in documentation websites.

{prettyunits}

{prettyunits} formats quantities such as file sizes, time durations, and byte counts in human-readable form. It is used internally by `devtools`, `remotes`, and progress-reporting packages.

{lobstr}

{lobstr} provides tools for visualizing R data structures, including shared-reference trees (`ref()`), abstract syntax trees (`ast()`), and object sizes (`obj_size()`). It is a useful learning and debugging tool.

{crayon}

{crayon} adds color and formatting to terminal output in R using ANSI codes. It has been largely superseded by `cli` but remains widely used in older code and packages.

{styler}

{styler} formats R code according to the tidyverse style guide. It provides an RStudio add-in and functions for styling individual files, packages, or projects. See Section for more details.

{rex}

{rex} provides a human-readable interface for constructing regular expressions in R without having to write complex, unreadable regex strings manually. It allows regular expressions to be composed modularly using functions such as `rex()`, `character_class()`, `capture()`, and `some_of()`.

When using `{rex}` inside another R package:

- Add `rex` to `Imports` in the package `DESCRIPTION`.
- Call `rex::register_shortcuts(pkgname)` in `.onLoad()` to prevent `R CMD check` NOTEs regarding undefined global variables when using `rex` shortcuts in package code.

- Call `rex::rex(...)` to compile expressions into regular expression strings for use with functions like `grepl()`, `gsub()`, or `{stringr}` functions.

GitHub Actions

actions

The `r-lib/actions` repository provides reusable GitHub Actions workflows for R package development. Commonly used actions include:

- `r-lib/actions/setup-r` — installs R
- `r-lib/actions/setup-renv` — restores an `renv` environment
- `r-lib/actions/check-r-package` — runs `R CMD check`
- `r-lib/actions/setup-r-dependencies` — installs package dependencies

These actions are widely used across R packages hosted on GitHub.

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 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}` 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 et al. 2023), which leans heavily on many Tidyverse packages and may be worth checking out.

Core Tidyverse Packages

The tidyverse 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)`:

ggplot2

{ggplot2} 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.

dplyr

{dplyr} 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()`.

tidyr

{tidyr} 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()`.

When to use dplyr vs tidyr

While both {dplyr} and {tidyr} 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.

readr

{readr} 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.

purrr

{purrr} enhances R's functional programming (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 for more details on using purrr.

tibble

{tibble} 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.

stringr

{stringr} 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.

forcats

{forcats} 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.

lubridate

{lubridate} 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.

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()`
`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_all(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.

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 (package vignette)
- Using dplyr in packages (package vignette)
- Tidy Eval in 5 Minutes (video)
- Data masking (rlang documentation)
- Evaluation (advanced details)
- Data Frame Columns as Arguments to Dplyr Functions (blog)
- Standard Evaluation for *_join (stackoverflow)

See also Tidyverse Replacements

A Skeptical View

Not everyone agrees that the Tidyverse should come first. In *Tidyverse Skeptic* (2019, source on GitHub), Norm Matloff argues that teaching R to noncoders through the Tidyverse is counterproductive. The essay is about teaching beginners, not about teaching computer science students or about whether R is a good tool for experienced coders (which Matloff calls a matter of personal taste). Its main points:

- **Keep it simple.** Base R (vectors, `$`, brackets, loops, `if`/`else`) is less abstract than the Tidyverse, so beginners reach useful analyses sooner.
- **Functional programming is hard for beginners.** Replacing loops with `{purrr}`-style function calls asks novices to master functions, which Matloff sees as the hardest part of R for them.
- **Pipes hinder debugging.** Code written as separate statements can be run and checked one line at a time, but a pipe is harder to step through, and beginners are the ones who make the most mistakes.
- **Narrow coverage looks like success.** Courses built around a few `{dplyr}` verbs leave students unable to use the many CRAN packages that expect base R idioms such as `$`, subscripting, and `summary()`.
- **Teach a mix.** Start with base R, then cover the Tidyverse (because it is popular) and `{data.table}` (because it is fast), so learners can choose the most convenient tool for each task.

Matloff also notes that `{ggplot2}` predates the Tidyverse, and while he praises Posit's overall contribution to R, he raises concerns about one company steering an open-source ecosystem.

Our recommendation above still stands for day-to-day project code, but the essay is a useful counterweight: learn enough base R to read other people's code, debug your own, and use the many packages that are not built around the Tidyverse.

Coding with R and Python

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

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.

`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.

`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:

```
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 separate lists as its input arguments, while `pmap()` takes a list of lists. Secondly, `mapply()` simplifies its output to a vector, matrix, or array when it can (`SIMPLIFY = TRUE` is the default) and otherwise returns a list, but `pmap()` always returns a list.

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

`mapply()` recycles shorter inputs to the length of the longest input, while `pmap()` recycles only inputs of length 1 and requires all other inputs to have the same length. In the example above, `mylist1` and `mylist2` each hold a single element (a vector), so both functions make a single call, `square_sum(0:3, 6:9)`. Assume that now `mylist2 = list(6:12)`. Then, `pmap(mylists, square_sum)` calls `square_sum(0:3, 6:12)`, and R's vector arithmetic inside `square_sum()` recycles `0:3`, generating `[36 50 68 90 100 122 148]` where elements 0, 1, and 2 are recycled to match 10, 11, and 12. R also gives the warning "longer object length is not a multiple of shorter object length."

Thus, unless the recycling pattern described above is a desirable feature for a certain experiment design, **when the input lists, or the vectors inside them, 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.

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.

`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.

`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`.

Reviewing Code

Peer code review is a collaborative practice that improves scientific rigor, fosters shared understanding, and catches bugs before analysis results are shared or published. The recommended pull request workflow, author and reviewer expectations, and response protocol are consolidated in Collaborative Software Development.

Why We Review Code

Code review serves several vital purposes:

- **Error prevention**: Catching data processing and analytical bugs early, before findings enter manuscripts or reports.
- **Reproducibility**: Verifying that scripts execute cleanly from scratch without undocumented manual interventions or environment-specific dependencies.
- **Knowledge transfer**: Sharing analytical methods, R idioms, and package features across the team.
- **Maintainability**: Ensuring that analysis pipelines can be understood, modified, and extended by other team members in the future.

What Reviewers Look For

When reviewing a colleague's code, evaluate these key dimensions:

1. **Reproducibility**:
    - Does the script run cleanly from start to finish in a fresh R session?
    - Are package dependencies managed reproducibly via `{renv}` (see Reproducible Environments)?
    - Are file paths relative and constructed with `{here}` (see Section), avoiding machine-specific absolute paths?
2. **Code correctness and data integrity**:
    - Are data transformations and filtering steps logical and documented?
    - Are missing values (`NA`), edge cases, and empty groupings handled appropriately?
    - Are key variables assigned descriptive labels and measurement units where applicable (see Section)?
3. **Style and structure**:
    - Does the code follow the project's R code style (see R Code Style)?
    - Are long chunks broken into focused functions (see Section)?
    - Are variable and function names self-describing (see Object Naming)?

4. **Documentation**:
    - Do comments explain non-obvious design choices or statistical assumptions (see Section)?
    - Are architectural or analytical decisions documented with an ADR where appropriate (see Section)?

How to Give Constructive Feedback

Effective code review builds a supportive, high-trust research culture:

- **Be kind and respectful**: Critique the code, not the author. Assume positive intent and acknowledge elegant solutions or clear documentation.
- **Be specific and actionable**: Explain why a change is recommended and provide concrete code suggestions or examples where possible.
- **Distinguish blocking vs. non-blocking comments**: Clearly separate essential correctness or reproducibility issues (blocking) from optional stylistic preferences or ideas for future exploration (non-blocking nits).
- **Ask clarifying questions**: When logic is unclear, ask the author to explain their intent before proposing a rewrite.

When to Approve

Approve a pull request when:

- All automated CI checks and workflows pass cleanly.
- The code runs end-to-end and produces expected outputs.
- All blocking review findings have been resolved or explicitly deferred to a tracked follow-up issue.
- The author has completed the pre-review quality checks in Section .

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}` package: Automates creation of reproducible examples
- R for Data Science: Making a reprex (Wickham et al. 2023): Step-by-step guide to creating effective reproducible examples

Additional Resources

Learning R

- R for Data Science (Wickham et al. 2023) - learn data science with the tidyverse
- R for Epidemiology - an introduction to R, with epidemiology examples
- The Epidemiologist R Handbook - task-oriented reference for applied epidemiology
- R Graphics Cookbook - recipes for common `{ggplot2}` figures
- R for Reproducible Scientific Analysis - Software Carpentry's introductory lesson
- Building reproducible analytical pipelines with R - project-level reproducibility in R
- Posit Recipes - short, task-focused R code snippets

Learning Python

- Python Tutor - steps through a short program one line at a time and draws each variable and object in memory; it visualizes Python, Java, C, C++ and JavaScript (not R) and limits program length, so use it on small examples

Coming to R from SAS

- The sassy system - packages that bring familiar SAS concepts to R; its `{procs}` package provides versions of `proc freq`, `proc means`, `proc ttest`, `proc reg`, and other common procedures
- R for SAS and SPSS Users (Muenchen) - common SAS and SPSS tasks, done in R
- SAS and R (Kleinman and Horton) - side-by-side SAS and R code for data management and analysis

General R Programming

- Advanced R (Wickham 2019) - deep dive into R programming and internals, including functions, object-oriented systems, and debugging
- Debugging R code (Simon Wood) - debugging R and compiled code called from R
- Differences between the base R and `{magrittr}` pipes - how `|>` and `%>%` behave differently
- Spatial Data Science: With Applications in R - spatial data analysis in R

Summary tables

- `{gtsummary}` - flexible demographics and summary tables; start here
- `{table1}` - HTML "Table 1" demographics tables
- `{arsenal}` - summary tables, from the Mayo Clinic
- `{procs}` (under "Coming to R from SAS" above) - SAS-style frequency and means tables

R Package Development

- R Packages (Wickham and Bryan 2023) - comprehensive guide to R package development
- Tidyverse design guide (Wickham 2023) - principles for designing R packages and APIs that are intuitive, composable, and consistent with tidyverse philosophy
- usethis documentation - workflow automation for R projects
- devtools documentation - essential development tools
- pkgdown documentation - create package websites
- testthat documentation - unit testing framework
- Releasing to CRAN and the CRAN Cookbook - preparing a CRAN submission (see also Code Publication)
- Contributing to R - getting involved with base R itself

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

Git and Version Control

- Happy Git and GitHub for the useR (Bryan 2023) - essential guide to using Git and GitHub with R
- Pull request helpers in `{usethis}` - the `pr_*()` workflow from R

- Oh Shit, Git!?! - plain-language fixes for common Git mistakes
- `{saperlipopette}` - practice Git rescue exercises from R
- Git Magic - source repository of a short, free Git book
- `{gitdown}` - R package for documenting git commit history as a bookdown report, organized by patterns like tags or issues

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/.

Fay, Colin, Sébastien Rochette, Vincent Guyader, and Cervan Girard. 2021. *Engineering Production-Grade Shiny Apps*. Chapman; Hall/CRC. https://doi.org/10.1201/9781003029878.

Matloff, Norm. 2019. "Tidyverse Skeptic: Why the Tidyverse Is Not a Good Vehicle for Teaching Noncoders." https://matloff.github.io/TidyverseSkeptic/Skeptic.html.

Parnas, D. L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12): 1053–58. https://doi.org/10.1145/361598.361623.

Reason, James. 2000. "Human Error: Models and Management." *BMJ* 320 (7237): 768–70. https://doi.org/10.1136/bmj.320.7237.768.

Stevens, W. P., G. J. Myers, and L. L. Constantine. 1974. "Structured Design." *IBM Systems Journal* 13 (2): 115–39. https://doi.org/10.1147/sj.132.0115.

Wickham, Hadley. 2019. *Advanced r*. 2nd ed. Chapman; Hall/CRC. https://doi.org/10.1201/9781351201315.

Wickham, Hadley. 2021. *Mastering Shiny*. O'Reilly Media. https://mastering-shiny.org/.

Wickham, Hadley. 2023. *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/.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, et al. 2014. "Best Practices for Scientific Computing." *PLoS Biology* 12 (1): e1001745. https://doi.org/10.1371/journal.pbio.1001745.

Wilson, Greg, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2017. "Good Enough Practices in Scientific Computing." *PLOS Computational Biology* 13 (6): e1005510. https://doi.org/10.1371/journal.pcbi.1005510.

Listing 1 `.github/workflows/benchmark.yaml`

```yaml
name: Benchmark

on:
  pull_request:
    branches: [main, master]
  workflow_dispatch:  # Manual trigger

jobs:
  benchmark:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - uses: r-lib/actions/setup-r@v2
        with:
          use-public-rspm: true

      - uses: r-lib/actions/setup-r-dependencies@v2
        with:
          extra-packages: any::bench, any::profvis

      - name: Run benchmarks
        run: |
          library(bench)
          library(mypackage)

          # Create benchmarks directory if it doesn't exist
          dir.create("tests/benchmarks", recursive = TRUE, showWarnings = FALSE)

          # Generate test data (or load existing test data)
          test_data <- generate_test_cohort(n = 10000)

          # Run benchmark
          results <- bench::mark(
            current_implementation = analyze_cohort(test_data),
            iterations = 100
          )

          # Save results
          saveRDS(results, "tests/benchmarks/current.rds")
        shell: Rscript {0}

      - name: Compare with baseline
        run: |
          # Load baseline and current results
          baseline <- readRDS("tests/benchmarks/baseline.rds")
          current <- readRDS("tests/benchmarks/current.rds")

          # Calculate percentage change in median time
          baseline_time <- as.numeric(baseline$median[1])
```