R Code Style

Last modified: 2026-11-10 10:40:46 (PST)

Follow these code style guidelines for all R code:

General Principles

We recommend the Google R Style Guide, which in turn mostly follows the tidyverse style guide (Wickham 2023). The following principles apply to all R code:

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

Many of these rules can be enforced automatically with a `.lintr.R` configuration file; see the UCD-SERG lab manual's example. See Section for details on automated style checking.

Clean Code Principles

Robert C. Martin's *Clean Code* (Martin 2008) provides timeless principles for writing readable, maintainable, and robust software. While originally framed in object-oriented languages, these core tenets translate directly to scientific computing and R programming:

Meaningful Names

- **Use intention-revealing names**: A variable or function name should tell you why it exists, what it does, and how it is used (e.g., `days_since_exposure` rather than `d`).
- **Make meaningful distinctions**: Avoid arbitrary suffixes like `data1` and `data2` or `info` and `data`.
- **Use pronounceable and searchable names**: Names should be easily communicated in conversations and found via search tools.
- **Functions should use verbs**: Function names should describe actions (e.g., `calculate_incidence()`, `fit_model()`, `plot_titers()`).

Functions

- **Small and focused**: Functions should do one thing, do it well, and do it only.
- **Single level of abstraction**: Keep operations in a function at the same conceptual level rather than mixing high-level orchestration with low-level data munging.
- **Few arguments**: Aim for 0 to 2 arguments where practical; if a function requires many parameters, group them into a configuration object or list.
- **Avoid side effects**: Functions should return values predictably without unexpectedly mutating global state.

Comments

- **Explain why, not what**: As detailed in Section , code should explain what it does through expressive naming and clear structure. Use comments to document the reasoning, statistical rationale, edge cases, or scientific domain decisions behind an implementation.
- **Do not leave commented-out code**: Rely on version control (Git) to preserve historical code rather than leaving dead blocks in source files.

The Boy Scout Rule

- **Leave the codebase cleaner than you found it**: When editing a script or chapter, make small improvements in stride—fix typos, improve a variable name, or break up a long expression. Over time, continuous incremental cleanup prevents code degradation.

Don't Repeat Yourself (DRY)

- **Eliminate duplication**: Originating in *The Pragmatic Programmer* (Hunt and Thomas 1999) and emphasized in *Clean Code*, the DRY principle states that every piece of knowledge must have a single, unambiguous representation. Duplicate logic is a primary source of bugs and maintenance overhead. When the same transformation or calculation appears in multiple places, extract it into a reusable helper function (see also Section).

Function Structure and Documentation

Every function should follow this pattern:

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

See also Documenting your code for general code documentation practices.

Explicit Return Statements

Following the Google R Style Guide's recommendation, always use explicit `return()` statements in functions, even when R's implicit return would work. This makes the function's intent clear and improves readability.

Example:

```
# Good: Explicit return
calculate_mean <- function(x) {
  result <- mean(x, na.rm = TRUE)
  return(result)
}

# Less clear: Implicit return (avoid)
calculate_mean <- function(x) {
  mean(x, na.rm = TRUE)
}
```

Note: The example `.lintr.R` configuration disables the `return_linter` to allow flexibility in older code, but we still recommend explicit returns in new code.

Assertions and Input Validation

Validating function arguments and preconditions early (often called "failing fast") prevents bugs from propagating deep into analysis pipelines or statistical models. R offers several strategies and packages for writing assertions, each suited to different stages of development:

Base R: `stopifnot()`

Base R provides `stopifnot()`, which halts execution if any of the supplied expressions are not all `TRUE`:

```
calculate_rate <- function(count, person_years) {
  stopifnot(
    is.numeric(count),
    length(count) == 1,
    !is.na(count),
    count >= 0,
    is.numeric(person_years),
    length(person_years) == 1,
```

```
    !is.na(person_years),
    person_years > 0
  )
  return(count / person_years)
}
```

- **Pros**: Zero dependencies; built into every R installation.
- **Cons**: Error messages can be cryptic in older R versions (though R 4.0+ improved messages to display the failing expression); does not provide prepackaged assertion helpers for complex data types.

{checkmate}: Fast, comprehensive argument checks

The {checkmate} package (on CRAN) is widely recommended for package development and production pipelines:

```
calculate_rate <- function(count, person_years) {
  checkmate::assert_number(count, lower = 0)
  # lower bound is inclusive; use smallest positive double to require person_years > 0
  checkmate::assert_number(person_years, lower = .Machine$double.xmin, finite = TRUE)
  return(count / person_years)
}
```

- **Pros**: Extremely fast (written in C); comprehensive family of assertion functions (`assert_*`, `check_*`, `test_*`); generates clear, informative error messages automatically.
- **Cons**: Adds a package dependency.

{cli} and {rlang}: Custom, friendly error conditions

Within the Tidyverse, {cli} and {rlang} are the modern standard for emitting structured, formatted error messages:

```
calculate_rate <- function(count, person_years, call = rlang::caller_env()) {
  if (!is.numeric(count) || length(count) != 1 || is.na(count)) {
    cli::cli_abort(
      "{.arg count} must be a single non-missing number, not {.obj_type_friendly {count}}.",
      call = call
    )
  }
  if (count < 0) {
```

```
    cli::cli_abort(
      "{.arg count} must be non-negative ({.val {count}} was provided).",
      call = call
    )
  }
  if (!is.numeric(person_years) || length(person_years) != 1 || is.na(person_years)) {
    cli::cli_abort(
      "{.arg person_years} must be a single non-missing number, not {.obj_type_friendly {pers
      call = call
    )
  }
  if (person_years <= 0) {
    cli::cli_abort(
      "{.arg person_years} must be positive ({.val {person_years}} was provided).",
      call = call
    )
  }
  return(count / person_years)
}
```

- **Pros**: Outstanding user feedback with inline formatting (`{.arg}`, `{.val}`); properly attributes the error to the caller environment via `call = rlang::caller_env()`; supports signaling custom condition classes (via `class = ...`) that callers can catch programmatically.
- **Cons**: Requires writing manual validation branches rather than one-line assertion calls.

Decomposing validation into helpers

When a function validates multiple inputs with the same pattern, extract the repeated checks into a helper function rather than repeating the `if`/`cli_abort` block for each input. See Decomposing validation blocks for a worked example.

Note on `{assertthat}`

The `{assertthat}` package, created by Hadley Wickham, popularized user-friendly assertions in R. While still widely used across existing codebases, it has not seen active development since 2019. For new code, prefer `{checkmate}` for comprehensive argument checking or `{cli}` paired with `cli::cli_abort()` for custom error messages.

Avoiding Deep Nesting

When writing code, do not nest function calls. Where feasible, also avoid nested function definitions:

- Do not pass one function call's result straight into another call, as in `f(g(h(x)))`. Give each intermediate result a name, or chain the steps with a pipe (`|>` in R); pipes are fine. Naming each step makes the data flow read top-to-bottom and leaves intermediate values inspectable in a debugger.
- Prefer standalone, top-level function definitions over functions defined inside other functions. Nested definitions hide reusable logic, complicate unit testing, and obscure scope.

The nested-call rule is a policy, not a default: a nested call is a code-review finding. It covers a call whose result is a value handed to the outer call as an argument. It does not cover the steps of a pipe chain, which already name their order, nor an argument the outer function captures and interprets itself, as in these examples, which are not a closed list: `{tidyselect}` helpers (`all_of()`, `any_of()`, `starts_with()`), data-masked expressions inside data-masking verbs such as `mutate()`, `filter()`, or `summarise()`, `aes()` mappings, and formulas. Those are part of the outer call's own syntax, and pulling them out would break or obscure it. The exemption stops at the captured argument itself. In `select(df, all_of(names(x)))`, `select()` captures `all_of()`, but `all_of()` evaluates `names(x)` as an ordinary value, so that inner call still gets a name:

```
keep_cols <- names(x)
select(df, all_of(keep_cols))
```

The nested-definition rule is a readability and maintainability default, not an absolute rule — keep the nesting when flattening it would be more convoluted (a trivial one-argument wrapper, or a closure that genuinely needs the enclosing scope).

Name `if()` and `while()` conditions before testing them

Compute every `if()` and `while()` condition ahead of the test: assign it to a named logical variable first, then test that name, so the condition inside the parentheses is always a bare name (or a literal, as in `while (TRUE)`). This holds even for a condition with no nested call, such as `is.na(strat)` or a single comparison like `x > 0`, because the name is what documents the test.

```
# Good: the name states what the test means
n_plots <- trace_strat_list |> lengths() |> sum()
single_plot <- n_plots == 1
if (single_plot) {
```

```r
  # ...
}

# Avoid: the inline expression hides a misplaced parenthesis
if (sum(lengths(trace_strat_list) == 1)) {
  # ...
}
```

For a `while()` loop, compute the named condition before the loop and recompute it at the end of the loop body.

The inline form above was a real bug, fixed in {serodynamics} PR #326. `sum(lengths(x) == 1)` counts the strata that hold exactly one plot, so the single-plot branch ran whenever any stratum held one plot, not only when the whole result was one plot. Splitting the condition into named steps makes the author say what each step should mean (`n_plots`, then `single_plot`), which puts the comparison on its own line where a misplaced parenthesis is visible in review, and the named value can be inspected in a debugger before the branch runs.

Prefer more, simpler steps over fewer, denser ones

The named-intermediate rule above operates within one expression. The same trade runs one level up, across a pipeline: given a choice, do less per step and take more steps.

Advanced R makes the observation while comparing a `purrr` pipeline against the base-R and `for`-loop versions of the same task, in Purrr style:

> It's interesting to note that as you move from purrr to base apply functions to for loops you tend to do more and more in each iteration. In purrr we iterate 3 times (`map()`, `map()`, `map_dbl()`), with apply functions we iterate twice (`lapply()`, `vapply()`), and with a for loop we iterate once. I prefer more, but simpler, steps because I think it makes the code easier to understand and later modify.

The gain is the same one named intermediates buy: each step is separately readable, separately testable, and separately replaceable, and a change lands in one step rather than in the middle of a compound one. Cost only shows up when a step is traversed enough times for the extra passes to matter — which is a claim to settle with performance benchmarking, not by assumption.

Lambdas in map() and apply-family calls

The nested-definition rule applies to `purrr::map*()` / `pmap*()` / `lapply()`-family call sites too: do not wrap a named function in an anonymous function (lambda) just to fix constant

arguments. Pass the mapped elements positionally and the constants through the mapping function's `...`:

```
# Preferred --- hr/power match schoenfeld_events()'s leading parameters
# positionally; the constant fractions ride along via map2's `...`
purrr::map2_dbl(
  df$hr, df$power, schoenfeld_events,
  p1 = frac_op, p2 = frac_nonop
)

# Avoid --- a lambda that only fixes constant arguments
purrr::map2_dbl(df$hr, df$power, function(h, p) {
  schoenfeld_events(hr = h, power = p, p1 = frac_op, p2 = frac_nonop)
})
```

`purrr`'s own documentation mildly recommends shorthand lambdas over `...`-passing. This preference deliberately overrides that: when the mapped elements line up with the callee's leading parameters, use `...` and skip the wrapper.

When a wrapper genuinely is necessary — the mapped element is not the callee's leading argument, is used more than once in the body, or the body is a real expression rather than a single call — define a named wrapper function in its own file rather than an inline lambda. The one exception is a demonstrated performance reason to define the wrapper nested inside the calling function (e.g. it must close over a large enclosing-scope object that would otherwise be passed repeatedly); that is the same closure escape hatch as the nested-definition rule above.

Prefer Existing Packaged Functions

Before writing a function, look for an existing packaged one that already does the job — and prefer it over rolling your own:

- Check, roughly in this order: base R and the tidyverse / r-lib packages, then a focused, well-maintained CRAN package, then packages your own group maintains (for the Morrison Lab, e.g. `{bcs}`, `{ettbc}`, `{gha}`, and the shared workflows there). Packages can depend on each other, so reuse across your own repos is fine.
- Reach for the packaged version unless it is genuinely unfit — the wrong API, a heavy dependency for a one-liner, or it does not quite do what you need.

Packaged functions are tested, documented, and maintained by other people; hand-rolling an equivalent duplicates that work, adds surface area to maintain, and risks subtle bugs the package already fixed. For example, use `withr::with_seed()` to set a seed and restore the RNG stream, rather than hand-rolling a `.Random.seed` save/restore.

This is a default, not an absolute rule. A tiny, dependency-free helper can beat pulling in a package, and sometimes nothing fits — but look first, and prefer the standard, well-known way over a bespoke one.

One reason to skip a packaged function does not count: an environment your own change chose. "The script runs on a bare R, so `rlang` is unreachable" is not evidence that `rlang` does not fit when this change is what decided the script would run that way. Install the package, fix the CI job, and re-run the comparison.

- **Do:** say whether a constraint ruling out a package is external or one of ours, before letting it decide.
- **Don't:** verify a self-imposed constraint and report that as having justified the hand-rolled version.

Prefer Per-Operation Grouping

When reviewing or writing `dplyr` code, prefer **per-operation grouping** (the `.by` argument) over persistent `group_by()` / `ungroup()` pairs. Apply it when the grouping is only needed for one operation.

```
# Preferred --- grouping is scoped to this summarise(), no ungroup() needed
df |> summarise(mean_x = mean(x), .by = group_col)

# Avoid --- group_by() persists and must be manually ungroup()'d
df |> group_by(group_col) |> summarise(mean_x = mean(x)) |> ungroup()
```

Reference: https://dplyr.tidyverse.org/reference/dplyr_by.html

Flag persistent `group_by()` calls during code review when `.by` would work — that is, when the grouping feeds exactly one downstream verb and no subsequent operation needs it to persist.

Avoid Hard-Coding Data with an External Source of Truth

Avoid hard-coding data that already has a reliable external source of truth — a version number, a package list, a dependency's release date, a set of downstream consumers, a schema, or an enum's valid values. Read or generate it from that source instead of copying a snapshot into the codebase:

- **Versions and pins.** Do not retype a dependency's version in prose or a second config file when a lockfile, `DESCRIPTION`, or manifest already states it — reference that file, or generate the mention from it.

- **Generated lists.** A list of consumers, plugins, or registered items that the source system can enumerate (an API, a directory scan, a registry) should be produced by querying that system, not maintained by hand alongside it.
- **Cross-file duplication.** When the same fact must appear in two places (a usage example and a reference doc, a schema and its example), generate the second from the first, or have CI check they agree, rather than trusting two hand-edited copies to stay in sync.

Prose enumerations count, and they are the ones that rot unnoticed

The "generated lists" bullet reads as being about code, and the rule is easiest to break in a sentence. An enumeration written into documentation — "the rules below (A, B, C, …)" — is a hand-maintained copy of a directory listing, and it has no generator, no test, and no linter behind it. Nothing fails when the directory grows; the sentence simply becomes wrong and stays wrong, while still reading as authoritative.

When a prose list mirrors something the filesystem or an API already enumerates, prefer a pointer to the source over the list: "every fragment under `coding-style/`" cannot drift, while a parenthetical naming seven of them silently can. Keep an explicit list only where the *selection* is the point — a curated subset, an ordering that matters — and then say that it is a selection, so a reader knows not to trust it as complete.

Fixing a drifted list by refreshing it only resets the clock. The list will drift again on the next addition, by exactly the same mechanism. Replace it with the pointer instead, and treat "this needs updating again" as the signal that it should not have been a list.

A softened count is still a pinned count. The rule that refreshing a drifted list only resets the clock has a near-miss. The move it does not rule out is replacing the figure with a vaguer version of itself — "on the order of 90K+ tokens across the ~60 files in it" in place of "~74-92K tokens across 59 files". Hedging acknowledges the imprecision and changes nothing about the mechanism, so the substitution reads as careful while drifting on the same schedule: a tree can shrink past "90K+" as easily as it grew past an exact figure.

The tell is that such a sentence has to describe its own restraint. A form needing a clause like "a count this sentence deliberately does not pin" pinned one, or it would have nothing to disclaim. The pointer form needs no such clause, because it names the query rather than an answer.

Hedging is the wrong answer whether or not the figure has a source. What the source decides is which right answer applies, and the discriminator is whether a *source can be named* rather than whether a command exists. A figure whose source is nameable — a directory, an API, a lockfile — gets the pointer. A count of the items in the block directly beneath it names nothing, so it gets dropped or re-derived instead.

- **Do:** replace a figure the filesystem or an API enumerates with the command that derives it.

- **Don't:** hedge such a figure and keep it — an approximation drifts on the same schedule the exact value did.

This is conditioned on the external source being **reliably available** — do not add a network fetch or a fragile dependency where a static value would do. A constant that has no external owner (a magic number intrinsic to the algorithm, a default chosen by this project) is not "hard-coded data" in this sense — it is just a value. The target is duplicated *ownership* of a fact: if updating the external source should have updated this value too, and did not, that is the bug this guidance prevents.

A count in the prose above a block is the same duplicate, one line away

The section above describes a list mirroring something *elsewhere* — a directory the sentence cannot see, drifting over weeks as files land. The tighter case is a count of the items in the block directly beneath it: "Three reads settle it", above three commands. Same defect, since the count is a hand-maintained copy of something the block already enumerates. What differs is who invalidates it and when. Nobody adding a file to some other directory breaks this one. **You** break it, in the same review round, by fixing the block the count describes.

Two things keep it out of view at exactly that moment:

1. The count was **correct when written**, so it was never a mistake to notice and carry forward — it became false only when the block gained a command. And a review finding points at the block, so correcting the block feels like the whole action. The sentence introducing it is not part of what the reviewer flagged, so nothing prompts a re-read.
2. Adjacency reads as safe. A count of items in a distant file is obviously fragile, and that visible fragility is what makes anyone check it. A count one line above the thing it counts feels like it cannot drift, since both are on the screen at once — which is precisely why nobody looks at the prose while editing the block.

Two remedies that work:

- **Drop the count.** The block is immediately below, so "These reads settle it" loses nothing a reader could not get by looking down. This is the better answer whenever the number carries no argument.
- **Re-derive it mechanically before pushing**, when the number is doing real work in the sentence. One command decides it exactly, which makes this a check to automate rather than something to settle by recollection.

Scope that counting command to the block, not to the whole file. A file-wide `grep -c` is right only while the pattern happens to match nothing outside the block, which is a property of the file today rather than of the command. An unrelated edit that adds one matching line anywhere else silently inflates the count, so the instrument acquires exactly the failure mode it was reached for to prevent.

- **Do:** re-read the sentence introducing a block whenever a review finding changes what is in that block.
- **Do:** delete a count the neighbouring block already states, and re-derive by command any count you keep.
- **Don't:** treat a fix to the block as complete because the finding named only the block.
- **Don't:** read adjacency as protection — the nearest duplicate is the one your own edit falsifies first.

A qualitative generalization above a block goes unchecked where a count would be re-derived

The section above governs a **count** stated above the block it counts, and its remedy is to drop the count or re-derive it by command. A qualitative claim over the same block is the same shape, and it slips past that remedy entirely, because it carries no number to re-derive.

A count invites verification because it is obviously a number someone must have measured, and a stale one reads as a typo waiting to be caught. A qualitative claim — "always", "independently of", "in every case", "the same in both" — carries no such tell. It reads as characterization rather than as a checkable fact, so it survives review exactly where a count would not.

The check is not "re-derive the number" — there is none — but "read the introducing sentence against the block", applied here to the sentence a table or block sits directly beneath.

- **Do:** re-read a lead-in sentence against its block for what it generalizes or quantifies, not only for a stated count.
- **Do:** treat "always", "independently of", "in every case", "the same in both" as tells that a claim needs checking against the data beneath it.
- **Don't:** assume a qualitative lead-in is safe because it carries no number that could go stale.

Where the rule stops: text that records what was observed

Everything above pushes toward replacing a literal with whatever owns it. There is one boundary it must not cross, and a consistency sweep is precisely the operation that crosses it without noticing.

Text that **asserts what was observed** is not configuration. A command someone actually ran, the output it actually produced, and the conditions a measurement was actually taken under are claims about the past, and their literals are the evidence for those claims. Parameterizing them does not generalize the record. It falsifies it, in the name of consistency, and leaves no trace that anything was changed.

Three forms, each of which looks like the hard-coding this section bans:

- **A command that was executed.** `git worktree add /tmp/wt-ums main` reports a run. Rewriting it to `<default-branch>` asserts a run that never happened.
- **A verbatim error string.** `fatal: invalid reference: origin/main` is what the tool printed. A reader matches it against their own terminal, so a parameterized version matches nothing and stops being findable.
- **The conditions of a measurement.** A sentence saying the runs used a repo whose default branch is literally `main` states the scope of the result. It is usually the sentence that explains why the measurement did not surface the bug.

The tell is tense and mood rather than syntax. Prescriptive text tells a reader what to do next, and should name the parameter. Evidentiary text says what happened, and should keep the literal. One file routinely carries both, so decide occurrence by occurrence.

- **Do:** parameterize the occurrences that instruct, and leave the ones that record.
- **Do:** decide per occurrence in a file that carries both, reading each one's surrounding sentence.
- **Don't:** run a whole-file replace over a literal that also appears inside quoted commands, quoted output, or a statement of measurement conditions.
- **Don't:** treat an unparameterized literal inside a case record as a defect left behind — there it is the evidence.

Construct Complex Inputs Before the Call

Build a complex argument as a named intermediate first, then pass that name to the function. Naming the intermediate keeps the call short, makes the data flow read top to bottom, and lets you inspect the value in a debugger.

```
# Good: name the intermediate, then pass it
model_vars <- c("age", "sex", "titer")
model_formula <- reformulate(model_vars, response = "outcome")
fit <- lm(model_formula, data = study_data)

# Avoid: complex input constructed inline in the call
fit <- lm(reformulate(c("age", "sex", "titer"), response = "outcome"), data = study_data)
```

A pipe is the other idiomatic way to avoid an inline-constructed argument, when the input is the result of a short sequence of transformations:

```
# Good: build the input with a pipe, then pass it
recent_cases <-
  case_data |>
  filter(year >= 2017) |>
```

```
  arrange(onset_date)

ggplot(recent_cases, aes(x = onset_date)) +
  geom_histogram()
```

This is the same readability goal as Section : prefer named steps you can read and check over one dense expression.

Name Function Arguments

Name the arguments in every function call, so a reader sees what each value is for without looking up the function's argument order.

```
# Good: each value says what it is for
df |> ggplot() + aes(x = day, y = Volume) + geom_line()
read_csv(file = path, show_col_types = FALSE)

# Avoid: the reader must know the argument order
df |> ggplot() + aes(day, Volume) + geom_line()
read_csv(path, show_col_types = FALSE)
```

There are two exceptions:

- A call with a single argument, such as `nrow(df)` or `log(x)`.
- A call whose positional arguments are extremely obvious, such as `head(data, 1)`.

Always name the arguments of `ggplot2::aes()`, even a single one: write `aes(x = Lag1)`, not `aes(Lag1)`. Positional `aes()` arguments are easy to misread, because the aesthetic each variable maps to is not visible at the call. `lms::aes_args_named_linter()` flags every unnamed `aes()` argument.

Comments

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

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

# Bad: States the obvious
```

```r
# Create a histogram
ggplot(data, aes(x = income)) + geom_histogram()
```

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

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

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

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

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

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

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

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

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

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

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

Line Breaks and Formatting

Blank Lines Before Lists

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

Correct:

```
Here are the requirements:

- First item
- Second item
```

Incorrect:

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

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

Here are the requirements: - First item - Second item

Semantic Line Breaks in Plain Text

Add a newline at the end of every phrase or logical unit of text in plain-text source files. A phrase is typically a complete thought, clause, or sentence. This applies to:

- Plain-text paragraphs in `.qmd` files
- Source code text: comments, documentation strings, and error messages

Correct (prose in .qmd):

```
When talking about code in prose sections,
use backticks to apply code formatting.
This helps maintain readability in source files
and makes diffs easier to review.
```

Incorrect (prose in .qmd):

```
When talking about code in prose sections, use backticks to apply code formatting. This helps
```

Correct (R code comment):

```
# First, check if the input is valid.
# Then, process the data.
# Finally, return the result.
```

Incorrect (R code comment):

```
# First, check if the input is valid. Then, process the data. Finally, return the result.
```

This practice is also known as semantic line breaks.

Guidelines:

- Break after complete sentences (at periods)
- Break after long phrases or clauses (at commas or conjunctions)
- Aim for lines under 80 characters
- Keep related short phrases together on one line
- Do not break in the middle of inline code, links, or formatting

Line Breaks in Code

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

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

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

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

And of a complex `ggplot` call:

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

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

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

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

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

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

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

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

```
        axis.text.x = element_text(size = 12)) +
  ggtitle(title)
```

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

Markdown and Quarto Formatting

Writing about code in Quarto documents

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

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

Messaging and User Communication

Use {cli} package functions for all user-facing messages in package functions. This is enforced by the example `.lintr.R` configuration via `undesirable_function_linter()`.

Required messaging functions:

- Use `cli::cli_inform()` instead of `message()`, `inform()`, or older `cli_alert_*()` functions
- Use `cli::cli_warn()` instead of `warning()` or `warn()`
- Use `cli::cli_abort()` instead of `stop()` or `abort()`

This provides better formatting, color support, and consistent messaging across packages.

Examples:

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

# Bad - don't use these in package code
message("Analysis complete")
```

```r
warning("Missing data detected")
stop("Invalid input")
```

CLI Styling: Progress Notes vs. Outcomes

When writing command-line tools, pipeline scripts, or package messaging, maintain a clear visual distinction between intermediate progress updates and terminal outcomes:

- **Progress updates should use neutral, non-emphatic styling**: Routine status notes describing in-flight actions (such as "Starting to compute…", "Loading input data…", or "Querying remote endpoint…") should remain quiet and plain. Avoid bold fonts, bright highlight colors, or heavy icons for intermediate steps. Applying emphatic styling to routine progress causes visual fatigue and desensitizes users to important messages.
- **Outcomes should receive emphatic styling**: Definitive conclusions and status transitions (such as "Succeeded", "Failed", or "Warning") signal results that require user awareness or intervention. Emphasize these outcomes with bold text, standard semantic colors (such as green for success, yellow for warning, and red for failure), or recognizable status indicators (such as check marks or crosses).

This separation creates an effective visual hierarchy: users can scan terminal output quickly without getting bogged down by intermediate noise, while immediately noticing whether a process succeeded or encountered issues.

Examples:

```r
# Good: quiet progress note, emphatic outcome
cli::cli_inform("Computing bootstrap estimates...")
cli::cli_inform(c("v" = "{.strong Succeeded:} computed {.val {n_reps}} replicates."))

# Good: emphatic warning status in pipeline reporting (does not signal an R warning condition
# (use cli_warn() in package code to signal an R warning condition)
cli::cli_inform(c("!" = "{.strong Warning:} {.val {n_dropped}} missing observations dropped.

# Good: emphatic failure status in batch scripts (non-aborting -- processing continues)
cli::cli_inform(c("x" = "{.strong Failed:} input file {.file data.csv} not found."))

# Good: emphatic failure condition that halts execution in package functions
cli::cli_abort("Failed to converge after {.val {max_iter}} iterations.")

# Bad: over-emphasized progress note (distracting and noisy)
cli::cli_inform(c("*" = "{.strong STARTING TO COMPUTE ESTIMATES...}"))
```

```r
# Bad: plain unstyled failure (and wrong function if execution should halt; use cli_abort())
cli::cli_inform("failed: check inputs")
```

Package Code Practices

- **No `library()` in package code**: Use `::` notation or declare in DESCRIPTION Imports. This is enforced by the example `.lintr.R` configuration via `undesirable_function_linter()`. Instead, use:
  - `::` for explicit namespace references (e.g., `dplyr::mutate()`)
  - `usethis::use_import_from()` to declare imports in `NAMESPACE`
  - `withr::local_package()` for temporary package loading in tests
- This keeps the global search path clean and makes dependencies explicit. See R Packages - The R landscape for more details.
- **Document all exports**: Use roxygen2 (`@title`, `@description`, `@param`, `@returns`, `@examples`)
- **Avoid code duplication**: Extract repeated logic into helper functions

Tidyverse Replacements

Use modern tidyverse/alternatives for base R functions:

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

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

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

# String operations
stringr::str_which()       # instead of grep()
stringr::str_subset()      # instead of grep(value = TRUE)
stringr::str_replace_all() # instead of gsub()
```

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

# Variable labels
labelled::set_variable_labels() # instead of attr(x, "label") <- ...

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

Subsetting data frames

Subset a data frame or tibble with `dplyr::filter()` for rows and `dplyr::select()` for columns. Extract one column as a vector with `dplyr::pull()`. Do not index a data frame with base `[]`.

```
# Rows and columns
smarket |>
  dplyr::filter(Year < 2005) |>
  dplyr::select(Lag1, Lag2)   # not smarket[smarket$Year < 2005, c("Lag1", "Lag2")]

# One column, as a vector
smarket |>
  dplyr::filter(Year < 2005) |>
  dplyr::pull(Direction)      # not smarket$Direction[smarket$Year < 2005]
```

Base `[]` stays for vectors, lists and matrices, where `filter()` and `select()` do not apply. When a matrix needs `x[i, ]`, mark the line `# nolint`.

See also Tidyverse.

The here Package

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

```
library(here)

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

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

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

Pass subpath components directly to `here()`

`here::here()` already accepts path components as arguments and joins them with the project root. Do not wrap `here::here()` in `file.path()` or `fs::path()`:

```
# Good: pass subpath components directly
rds_path <- here::here("inst", "extdata", "mapping.rds")

# Bad: redundant wrapping adds noise and defeats the purpose
rds_path <- file.path(here::here(), "inst", "extdata", "mapping.rds")
rds_path <- fs::path(here::here(), "inst", "extdata", "mapping.rds")
```

The wrapped form is functionally equivalent but harder to read and obscures the intent. `here()` was designed to replace `file.path()` for project-relative paths, so let it do its job.

See also The here package for detailed explanation of the here package.

Object Naming

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

```
# Good
vaccination_coverage_2017_18
absentee_flu_residuals

# Less good
vaxcov_1718
flu_res
```

Prefer nouns for objects and verbs for functions:

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

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

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

Use consistent prefixes to signal what a function returns:

- `add_...()` for functions that add one or more columns to a `data.frame` or `tibble` and return the modified table.
- `compute_...()` for functions that compute and return a single vector.

```r
# Good
add_age_group <- function(data) {
  data |> dplyr::mutate(age_group = cut(age, breaks = c(0, 18, 65, Inf)))
}

compute_age_group <- function(age) {
  cut(age, breaks = c(0, 18, 65, Inf))
}

# Less clear
make_age_group <- function(data) { ... }
get_age_group  <- function(age)  { ... }
```

This distinction makes the return type visible from the call site without reading the function body.

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

Uppercase acronyms are allowed in `snake_case` names (e.g., `prep_IDs_data`, `calculate_BMI_score`). This is enforced via a custom `object_name_linter` regex pattern in the example `.lintr.R` configuration.

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

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

Give each object one lasting name. Use an alias to shorten a name inside one piece of code, but avoid creating a synonym. GeeksforGeeks compares aliases and synonyms in SQL:

- An **alias** is a second name that lasts only within one scope. In SQL, `SELECT pat.id AS patient_id FROM patients pat` names the table `pat` and the column `patient_id` for that query only. `import pandas as pd` in Python names the module `pd` for one file.
- A **synonym** is a second name that is stored, so it lasts until someone removes it. Some databases, such as Oracle and SQL Server, provide `CREATE SYNONYM`, which stores a name that points to another object (Oracle documentation). PostgreSQL and MySQL do not have this command.

Keep an alias recognizable, such as `pat` rather than `p`, since the descriptive-name advice at the top of this section still applies. A synonym in your own code makes readers look for a difference between the two names.

An R package has one common exception. When you rename an exported function, keep the old name for a while as a wrapper so that existing code still runs, and mark it as deprecated with {lifecycle}:

```
# Keep the old name only as a deprecated wrapper
prep_data <- function(raw_data) {
  lifecycle::deprecate_warn("1.2.0", "prep_data()", "prep_study_data()")
  prep_study_data(raw_data)
}
```

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

Automated Tools for Style and Project Workflow

Styling

RStudio shortcuts

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

```
# Before
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Carl Munck Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Munck")
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, patte
  "East Oakland PRIDE Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "EnCompass Academy" = str_subset(string = OUSD_school_shapes$schnam, pattern = "EnCompass")
  "Global Family School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Global")
  "International Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern =
```

```
  "Madison Park Lower Campus" = "Madison Park Academy TK-5",
  "Manzanita Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Ma
  "Martin Luther King Jr Elementary" = str_subset(string = OUSD_school_shapes$schnam, patter
  "PLACE @ Prescott" = "Preparatory Literary Academy of Cultural Excellence",
  "RISE Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Rise Co
)
```

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

{styler}

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

> **Note**
>
> The default Tidyverse styler is subtly different from some of the things we've advocated for in this document. Most notably we differ with regards to the assignment operator (`<-` vs `=`) and number of spaces before/after "tokens" (i.e. Assignment Aligner add spaces before `=` signs to align them properly). For this reason, we'd recommend the following: `style_dir(path = ..., scope = "line_breaks", strict = FALSE)`. You can also customize {styler} even more if you're really hardcore.

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

> **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}`

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

> **Note**
>
> You can customize your settings to set defaults or to exclude files. More details can be found here.

Note

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

Using Lintr

`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 (see note below)
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

For package code, run `pkgload::load_all()` (or `devtools::load_all()`) **before** `lintr::lint_package()`. The example configuration includes `object_usage_linter()`, which resolves symbols using the loaded package; without loading first, it checks against a stale installed copy (or none), producing spurious `no visible binding for global variable` warnings. Loading also runs any `.onLoad()` side effects: for example, the `{lms}` linter package registers its `rex` shortcuts in `.onLoad()`, and the linter only sees them once the package is loaded.
We recommend `.lintr.R` files for configuration (lintr also supports the `.lintr` format, but `.lintr.R` gets full R syntax support).

A Recommended Lintr Configuration

We recommend a custom `.lintr.R` configuration file in each repository to enforce these style standards. You can view an example configuration, from the UCD-SERG lab manual, at https://github.com/UCD-SERG/lab-manual/blob/main/.lintr.R.

Key linters to enable:

- `pipe_consistency_linter(pipe = "|>")`: Enforces use of native pipe `|>` instead of `%>%`
- `object_name_linter()`: Enforces `snake_case` with custom regex allowing uppercase acronyms
- `undesirable_function_linter()`: Prohibits base messaging functions and `library()` in package code
- `redundant_equals_linter()`: Catches redundant `= TRUE` when `TRUE` is the default

Linters to disable:

- `return_linter(return_style = "explicit")`: Every function should end with `return(return_value)` rather than just `return_value`. You may disable this linter in older projects until you are ready to clean them up, but we recommend explicit returns for all new code, following the Google R Style Guide.
- `trailing_whitespace_linter`: Disabled (handled by `styler` instead)

Exceptions:

The example configuration allows relaxed rules for certain directories:

- `data-raw/`: Pipe consistency and undesirable function rules relaxed (exploratory scripts)
- `vignettes/`: Undesirable function and object naming rules relaxed (tutorial code may need `library()`)
- `inst/examples/`: Undesirable function rules relaxed
- `tests/testthat.R`: Undesirable function rules relaxed

{jarl}

{jarl} (Just Another R Linter) is a fast, standalone R linter written in Rust by Etienne Bacher. It is designed to provide high-speed static analysis for R code with minimal overhead.

Because {jarl} is implemented in Rust, it performs static analysis significantly faster than standard R-based linters.

Key Features

- **Fast execution:** Written in Rust to deliver near-instant linting feedback.
- **`{lintr}` compatibility:** Implements many common `{lintr}` rules (such as unused variables, improper whitespace, indentation, object naming, and syntax checks).
- **Flexible invocation:** Can be run as a standalone CLI tool, via pre-commit hooks, or integrated into editors via its editor integrations.
- **Auto-fixing:** Can automatically fix supported rule violations via `--fix`.
- **Project configuration:** Configured using a `jarl.toml` file in the project root.

Basic Usage

Run the standalone CLI (see the getting started guide):

```
# Lint all R files in the current project
jarl check .

# Automatically fix supported lint violations
jarl check . --fix
```

Comparison with `{lintr}`

Feature	`{lintr}`	`{jarl}`
Implementation	R	Rust
Execution speed	Standard	Extremely fast (Rust binary)
Ecosystem maturity	Industry standard, wide rule coverage	Emerging, expanding rule set
Custom R linters	Yes (e.g., `{lms}`)	No (Rust-only rule definitions)
Primary use case	CI verification, custom project rules	Fast pre-commit checks, local interactive editing

Recommendation

We recommend `{lintr}` with a custom `.lintr.R` configuration and a shared linter package (for example, the UCD-SERG lab manual's `.lintr.R` and this guide's `{lms}`) as the primary standard for CI pipelines and package validation. However, `{jarl}` is an excellent choice for local pre-commit hooks and interactive developer workflows where instant lint feedback saves time on large repositories.

Linting Changed Files vs. the Whole Project in CI

A project can fall out of lint compliance without any change to its own source code or `.lintr` configuration: `{lintr}` releases sometimes add default linters or tighten existing ones, so code that passed yesterday can fail after a routine package update.

When a continuous-integration job lints the whole project (for example, `lintr::lint_dir()` on every pull request), those newly introduced findings surface on whatever pull request happens to run CI next. That pull request then has to expand into unrelated files just to turn the lint check green, which conflicts with the expectation that a pull request stays scoped to one concern.

In most cases, a better default is to lint **changed files only**, so pre-existing findings elsewhere in the project don't block unrelated work. r-lib/actions provides a ready-made `lint-changed-files` example workflow; install it with `usethis::use_github_action("lint-changed-files")`.

Whole-project linting still has a place: run it on a schedule or as a manually triggered workflow, so that project-wide drift is still detected and cleaned up deliberately in its own dedicated pull request, rather than as a side effect of someone else's change.

Additional Resources

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

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Martin, Robert C. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Wickham, Hadley. 2023. *The Tidyverse Style Guide*. https://style.tidyverse.org/.