Updating an R Package: A Complete Development Workflow

From code changes to CI/CD integration using Git, devtools, and GitHub Actions

Ronald 'Ryy' G. Thomas

2026-02-11

Table of contents

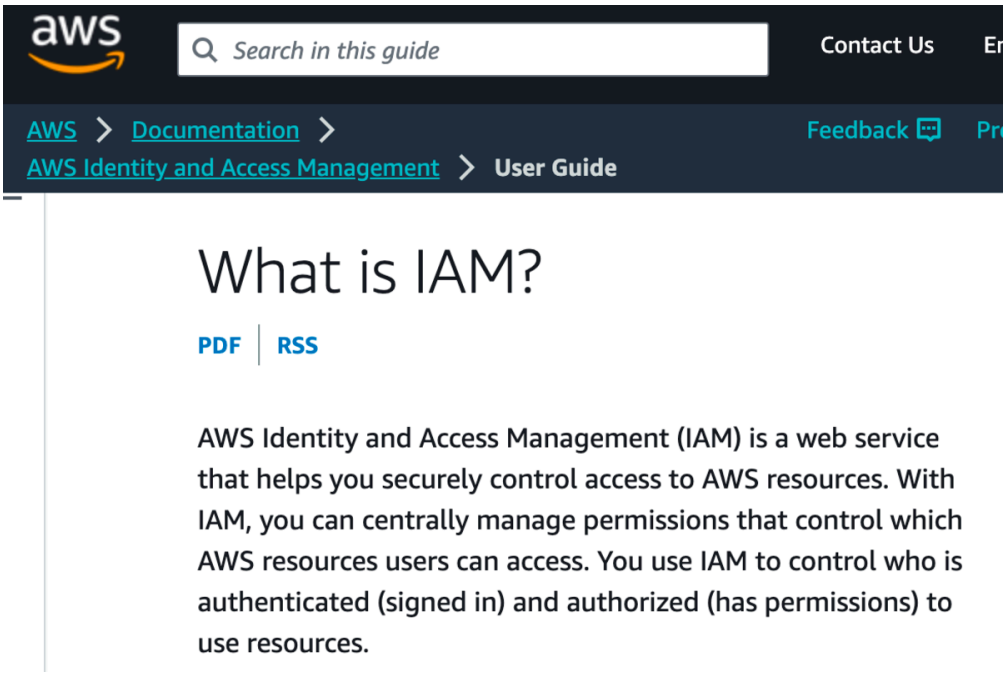

Figure 1: Terminal showing an R package build in progress

Setting up a reproducible R package development workflow.

1 Introduction

I did not really understand how much discipline goes into modifying an R package until I tried to push a change to one of my own projects and watched CI fail on three platforms simultaneously. It turns out that editing an R file is the easy part; the surrounding workflow of branching, documenting, testing, building, and integrating with GitHub Actions is what separates a quick fix from a reliable contribution.

The scenario is straightforward: you have an existing R package hosted on GitHub, and you need to add a feature, fix a bug, or improve error handling. The change itself might be small, but the process of getting it safely merged involves a surprising number of steps.

This post documents the complete workflow I learned while modifying my own package, `zz-dataframe2graphic`. Every step from creating a feature branch to cleaning up after a merge is covered, with the goal of making the process repeatable for future changes.

More formally, this post documents the R-packages layer (Layer 13) of the Workflow Construct described in post 52, specifically the maintenance loop that keeps an existing package healthy across a working biostatistician's many simultaneous projects. The construct's R-packages layer holds a small set of author-maintained `zz*` packages; this post is the how-to-update-one companion to post 35, which is the how-to-author-one entry point.

1.1 Motivations

- I kept making ad hoc changes to my R packages without a systematic process, and occasionally broke things on other platforms.
- I wanted a clear mental model for the branch-edit-test-merge cycle so I could contribute to open source R projects with confidence.
- I had GitHub Actions YAML files in my repositories but did not fully understand what they did or how to set them up from scratch.
- I needed a reference I could return to each time I make a package change, rather than re-learning the steps every time.
- I wanted to understand how `devtools`, `testthat`, and `roxygen2` fit together in a single coherent workflow.

1.2 Objectives

1. Create a feature branch, make code changes, and update roxygen2 documentation in an R package.
2. Write and run unit tests locally with `testthat` and build the package with `devtools`.
3. Set up three GitHub Actions workflows (R-CMD-check, test coverage, pkgdown) and understand what each one does.
4. Execute the full pull request lifecycle: push, review, address CI failures, merge, and clean up.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Figure 2: Settling in for a focused development session.

Settling in for a focused development session.

2 Prerequisites and Setup

This workflow assumes familiarity with R, basic Git commands, and a GitHub account. The following tools should be installed:

```
install.packages(
  c("devtools", "testthat", "roxygen2",
    "rcmdcheck", "covr", "pkgdown")
)
```

You also need Git configured on your system and SSH or HTTPS access to your GitHub repositories. The examples below use a package called `zzdataframe2graphic`, but the process applies to any R package hosted on GitHub.

3 What is an R Package Development Workflow?

An R package development workflow is a structured sequence of steps that takes a code change from an idea to a tested, documented, and merged contribution. Think of it as a checklist for responsible software modification: rather than editing a file and hoping for the best, you create an isolated branch, update documentation automatically, run tests on multiple platforms, and only merge when everything passes.

The key tools in this workflow are `devtools` (which orchestrates building, testing, and checking), `roxygen2` (which generates documentation from inline comments), `testthat` (which provides a structured testing framework), and GitHub Actions (which runs automated checks on every push). Together, they form a safety net that catches problems before they reach production.

4 Getting Started: Creating a Feature Branch

Every change begins with a new branch. This keeps the main branch stable while you work on your modification.

```
git checkout main
git pull origin main
git checkout -b feature-name
```

The first two commands ensure you are starting from the latest version of the codebase. The third creates a new branch and switches to it. All subsequent changes happen on this branch, isolated from `main` until you are ready to merge.

4.1 Editing the R Code

With the branch created, navigate to the relevant source file and make your changes. For an R package, the source files live in the `R/` directory:

1. Open the target file (e.g., `R/zzdataframe2graphic.R`).
2. Make the code changes.
3. Update or add roxygen2 comments above any modified or new functions.
4. Save the file.

After editing, regenerate the documentation:

```
devtools::document()
```

This command reads your roxygen2 comments and updates the `NAMESPACE` file and the `man/` directory. It is important to run this every time you change function signatures, add parameters, or modify exported functions.

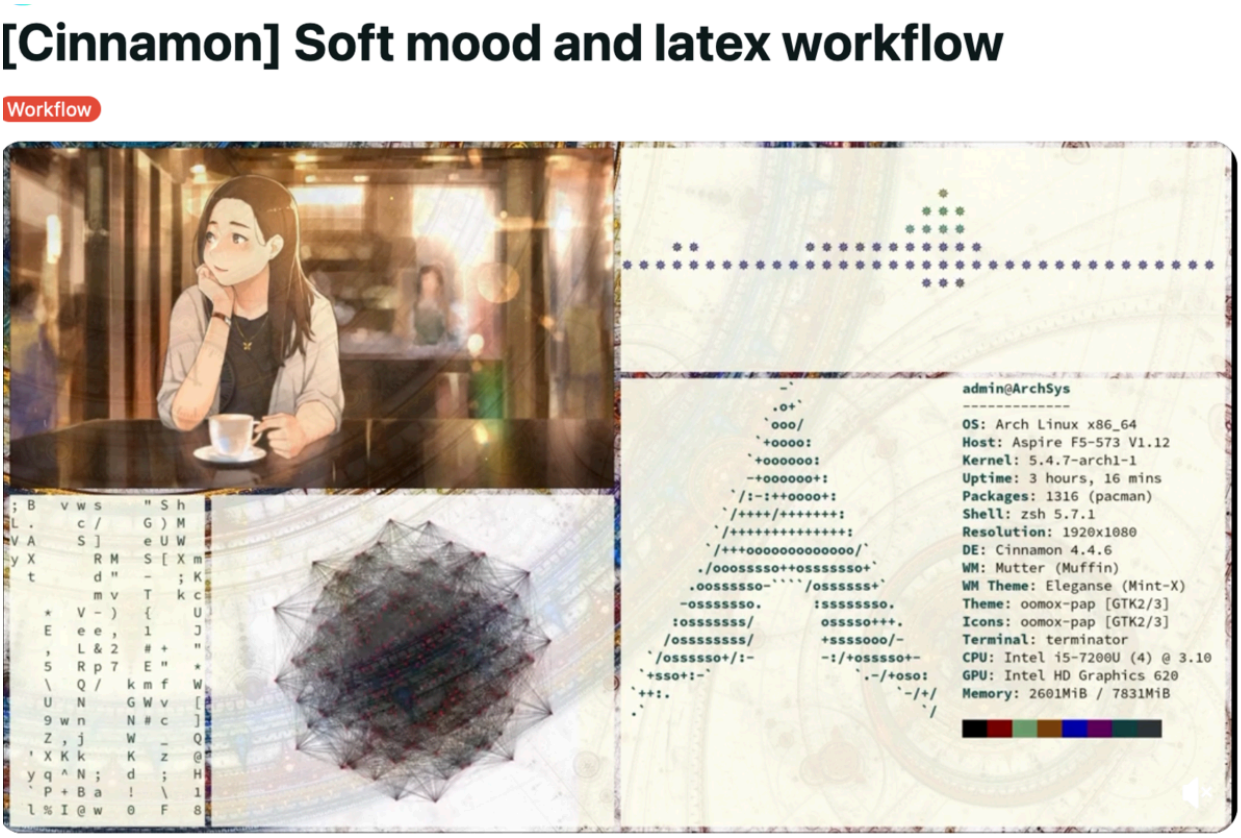

Figure 3: Code review on a laptop screen

Taking a step back to review the changes before testing.

4.2 Writing and Running Tests

Every code change should be accompanied by tests. The `testthat` framework provides a clean structure for this:

```
test_that("new feature works as expected", {
  result <- your_function(test_input)
  expect_equal(result, expected_output)
})
```

Add your test cases to the appropriate file in `tests/testthat/`. Then run the full test suite locally:

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

Fix any failures before proceeding. If the change is substantial, consider adding multiple test cases covering edge cases and expected error conditions.

4.3 Building and Checking the Package

Once tests pass, build and check the package:

```
devtools::build()
devtools::check()
```

The `check()` function runs `R CMD check`, which is the standard quality gate for R packages. It verifies that documentation is complete, examples run without error, tests pass, and the package can be installed cleanly.

For more rigorous checking, run against additional platforms:

```
devtools::check_win_devel()
devtools::check_mac_release()
rcmdcheck::rcmdcheck(args = c("--as-cran"))
```

The `--as-cran` flag applies the stricter checks that CRAN uses when evaluating package submissions.

4.4 Updating the DESCRIPTION File

Before committing, update the package metadata:

1. Increment the version number (e.g., from 0.2.0 to 0.2.1).
2. Update the `Date` field.
3. Add any new dependencies to `Imports` or `Suggests`.

5 The Git and GitHub Workflow

With all local checks passing, it is time to commit, push, and open a pull request.

5.1 Staging and Committing

```
git status
git add R/zzdataframe2graphic.R
git add tests/testthat/test-zzdataframe2graphic.R
git add DESCRIPTION
git add man/*
git add NAMESPACE
```

Review the staged changes with `git status`, then commit with a descriptive message:

```
git commit -m "feat: add error handling to
  zzdataframe2graphic
```

```
- Added input validation for data frame columns
- Updated roxygen2 documentation
- Added unit tests for edge cases"
```

5.2 Pushing and Creating a Pull Request

```
git push origin feature-name
```

Then create a pull request on GitHub:

1. Navigate to the repository on GitHub.
2. Click the "Pull requests" tab.
3. Click "New pull request."
4. Set base to `main` and compare to `feature-name`.
5. Fill in the PR description: what changed, why, how to test, and any related issues.

5.3 GitHub Actions: Automated CI/CD

GitHub Actions automate the testing process every time you push or open a pull request. A typical R package uses three workflows:

R-CMD-check runs `R CMD check` across multiple operating systems:

- Windows (latest)
- macOS (latest)
- Ubuntu (latest, with R-release, R-devel, and R-oldrel)

This workflow triggers on pushes and pull requests to the main branch.

Test coverage runs all package tests, generates coverage reports, and identifies which parts of the code need additional testing.

pkgdown builds the package documentation website and deploys it to GitHub Pages whenever changes are pushed to the main branch.

5.4 Setting Up GitHub Actions

To add these workflows to a package:

```
mkdir -p .github/workflows
```

Then add three YAML configuration files:

- `.github/workflows/R-CMD-check.yaml`
- `.github/workflows/test-coverage.yaml`
- `.github/workflows/pkgdown.yaml`

Configure repository permissions:

1. Go to Settings, then Actions, then General.
2. Set "Workflow permissions" to "Read and write."
3. Enable GitHub Pages deployment.

The `r-lib/actions` repository on GitHub provides standard workflow templates for all three.

5.5 Addressing CI Failures

If any workflow fails after pushing:

1. Check the "Actions" tab on GitHub.
2. Review the failure logs for each workflow.
3. Fix the issues locally.
4. Push the fixes:

```
git add .
git commit -m "fix: address CI failures"
git push origin feature-name
```

The pull request will automatically re-run the workflows.

5.6 Merging and Cleaning Up

Once all checks pass and the PR is approved:

1. Choose a merge strategy (usually "Squash and merge").
2. Update the PR title if needed.
3. Click "Squash and merge."

Then clean up locally:

```
git checkout main
git pull origin main
git branch -d feature-name
```

5.7 Creating a Release (Optional)

For significant changes, create a GitHub release:

1. Navigate to the "Releases" section.
2. Click "Create new release."
3. Tag the version (e.g., `v0.2.1`).
4. Add release notes summarizing the changes.
5. Publish the release.

6 Verification

After completing the branch-edit-test cycle, confirm that each stage of the pipeline passes.

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

devtools::check(args = c("--as-cran"))

git log --oneline -5
```

If `document()` runs without warnings, `test()` reports no failures, and `check()` returns 0 errors, 0 warnings, 0 notes, the package is ready to push.

7 Daily Workflow

Task	Command
Create feature branch	`git checkout -b feature-name`
Regenerate docs	`devtools::document()`
Run tests	`devtools::test()`
Build package	`devtools::build()`
Full check	`devtools::check(args = c("--as-cran"))`
Push branch	`git push origin feature-name`
Clean up after merge	`git checkout main && git pull && git branch -d feature-name`

7.1 Things to Watch Out For

1. **Run `devtools::document()` before committing.** I have forgotten this step more than once, leading to CI failures because the NAMESPACE file was out of date.
2. **Stage files explicitly rather than using `git add .`** The R build process creates temporary files that should not be committed. Staging specific files avoids accidental inclusion of build artifacts.
3. **Check on multiple platforms.** A package that passes on macOS may fail on Windows due to path separators or system dependency differences. The multi-platform R-CMD-check workflow catches these issues.
4. **Keep commits focused.** A PR that changes one function, updates its documentation, and adds its tests is easy to review. A PR that touches fifteen files across unrelated features is difficult to evaluate.
5. **Address CI failures immediately.** Letting failures accumulate makes debugging harder. Fix each failure as it appears and push the fix before moving on.

8 Uninstall / Rollback

To revert a package change that has not yet been merged, delete the feature branch locally and on the remote.

```
git checkout main
git branch -D feature-name
git push origin --delete feature-name
```

If the change has already been merged, revert the merge commit on `main`:

```
git revert <merge-commit-hash>
git push origin main
```

To remove the development toolchain entirely:

```
remove.packages(c(
  "devtools", "testthat", "roxygen2",
  "rcmdcheck", "covr", "pkgdown"
))
```

Figure 4: Books and notes on a research desk

Reflecting on the lessons from a full development cycle.

9 What Did We Learn?

9.1 Lessons Learnt

Conceptual Understanding:

- The R package structure is not just an organizational convenience; it is a contract that `R CMD check` enforces across documentation, dependencies, and test coverage.
- Feature branches isolate risk. Working directly on `main` means every mistake is immediately visible to collaborators and CI systems.
- Automated CI/CD on three operating systems catches platform-specific issues that local testing on a single machine cannot detect.
- The pull request is not just a merge mechanism; it is a documentation artifact that records what changed, why, and what tests confirmed the change works.

Technical Skills:

- `devtools::document()` regenerates `NAMESPACE` and `man/` from roxygen2 comments, eliminating manual documentation maintenance.
- `devtools::check()` with `--as-cran` applies the strictest quality standards and is worth running before every push.
- The `r-lib/actions` repository provides ready-to-use GitHub Actions YAML templates for R-CMD-check, test coverage, and pkgdown.
- `rcmdcheck::rcmdcheck()` provides more detailed output than `devtools::check()` and is useful for diagnosing obscure failures.

Gotchas and Pitfalls:

- Forgetting to run `devtools::document()` after changing roxygen2 comments causes NAMESPACE mismatches that fail CI silently.
- Using `git add .` can accidentally stage build artifacts, `.Rhistory`, or `.DS_Store` files that do not belong in the repository.
- GitHub Actions workflows require "Read and write" permissions under repository settings; the default "Read" permission causes pkgdown deployment to fail without a clear error message.
- Test coverage workflows may require LaTeX dependencies for vignette building; missing system dependencies produce cryptic errors that do not mention LaTeX directly.

9.2 Limitations

- This workflow assumes a single-developer context. Multi-contributor projects require additional conventions around code review, branch protection rules, and merge conflict resolution.
- The GitHub Actions templates from `r-lib/actions` target CRAN-style packages. Packages with non-standard system dependencies may need custom workflow modifications.
- Test coverage reports measure which lines of code are executed during tests, not whether the tests are meaningful. High coverage does not guarantee high quality.

- The `--as-cran` check is conservative and may flag issues that are acceptable for internal or non-CRAN packages.
- This post does not cover continuous deployment to package repositories (e.g., r-universe or drat) or automated version bumping.

9.3 Opportunities for Improvement

1. Add pre-commit hooks that run `devtools::document()` and `devtools::test()` automatically before each commit.
2. Integrate `lintr` into the CI pipeline to enforce consistent code style across all contributions.
3. Set up branch protection rules on `main` to require passing CI checks before merging.
4. Add a `NEWS.md` file to track user-facing changes with each version increment.
5. Explore `usethis::use_github_action()` to generate workflow files directly from R rather than copying YAML templates manually.
6. Consider adding a code coverage badge to the README to make test coverage visible at a glance.

10 Wrapping Up

The R package development workflow is more involved than simply editing code and pushing to GitHub, but each step exists for a reason. The branching strategy protects the main branch, the documentation tools keep help files synchronized with code, the testing framework catches regressions, and GitHub Actions verify everything works across platforms.

What I learned most from going through this process is that the overhead of a proper workflow pays for itself quickly. The first time CI catches a platform-specific bug that I would never have found on my own machine, the entire setup justified its existence.

For anyone starting out with R package development, my advice is to set up the full workflow once, even for a small package, and then follow it consistently. The steps become automatic after a few iterations.

Main takeaways:

- Always branch before making changes, and keep each branch focused on a single feature or fix.
- Run `devtools::document()`, `devtools::test()`, and `devtools::check()` locally before pushing.
- Set up R-CMD-check, test coverage, and pkgdown GitHub Actions workflows using the templates from `r-lib/actions`.
- Treat the pull request as both a merge mechanism and a documentation record.

11 See Also

11.0.1 Related Posts

- Configure the Command Line for Data Science Development

- Setting Up Git for Data Science Projects

11.0.2 Key Resources

- R Packages (2e) by Hadley Wickham and Jenny Bryan – the definitive guide to R package development
- Pro Git Book – comprehensive guide to Git fundamentals
- GitHub Actions for R – r-lib's collection of workflow templates
- testthat documentation – the standard testing framework for R
- devtools documentation – comprehensive guide to the devtools package
- Writing R Extensions – the official R manual for package development
- roxygen2 documentation – inline documentation system for R
- pkgdown documentation – building package documentation websites
- covr documentation – understanding test coverage in R
- GitHub Actions Documentation – official reference for CI/CD workflows
- GitHub Skills – interactive courses for learning GitHub
- Oh Shit, Git!?! – practical solutions for common Git mistakes

12 Reproducibility

This post describes a workflow rather than an analysis pipeline. To follow along, you need:

- R 4.4 or later
- Git 2.30 or later
- A GitHub account with repository access
- The packages listed in the Prerequisites section

The key commands in sequence:

```
git checkout -b feature-name
# Edit R source files and roxygen2 comments

devtools::document()
devtools::test()
devtools::build()
devtools::check()

git add <specific-files>
git commit -m "feat: description of change"
git push origin feature-name
# Open PR on GitHub, wait for Actions, merge
git checkout main && git pull origin main
git branch -d feature-name

sessionInfo()
```

```
R version 4.5.3 (2026-03-11)
Platform: aarch64-apple-darwin20
Running under: macOS Tahoe 26.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib;  LAPACK version

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.5.3  fastmap_1.2.0   cli_3.6.5       tools_4.5.3
 [5] htmltools_0.5.9 parallel_4.5.3  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.57
[13] digest_0.6.39   rlang_1.1.7     evaluate_1.0.5
```

13 Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.