Continuous Integration

Last modified: 2026-11-03 08:35:08 (PST)

Understanding GitHub Actions

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

Key benefits of GitHub Actions:

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

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

Action Versioning

When a workflow references an action, it must specify which version to use:

```
uses: r-lib/actions/setup-r@v2
```

The string after `@` is a Git ref—it can be a branch name, a commit SHA, or a tag. For most workflows from `Morrison-Lab/gha` you will see a short tag like `v2`, and it is worth understanding what that actually means.

Floating (mutable) tags

`v2` is a **floating tag** (sometimes called a "sliding tag" or "mutable tag"): a Git tag that the action's maintainers deliberately move forward each time they publish a new backward-compatible release. When your workflow runs, GitHub resolves `v2` to whatever commit that tag currently points to, so you automatically pick up bug fixes, new platform support, and tool upgrades without editing your workflow file.

This is the officially recommended pattern for GitHub Actions: maintainers commit to keeping all changes under a major-version floating tag backward-compatible, and bump to `v3` only for breaking changes.

> **Note**
>
> "Floating tag" is the standard term used across the ecosystem.

How a floating tag differs from a branch

Floating tags and branches are superficially similar—both are mutable named pointers to a commit. The key differences are:

- **Branches** advance automatically as commits are pushed. A floating tag only moves when the maintainer explicitly force-pushes it, so they retain deliberate control over when `v2` advances.
- **Semantics**: branches signal active development; a floating major-version tag signals a stable, maintained release line.
- In the GitHub Actions context the two are functionally equivalent as refs, but `@v2` communicates intent in a way that `@main` does not.

> **Note**
>
> Moving tags are generally considered bad practice in version control, because a tag that can change breaks the expectation that tagged commits are stable reference points—other collaborators who have already fetched the old target will silently diverge from the new one. See this GitHub Actions Toolkit discussion for a detailed discussion of the pitfalls. GitHub Actions major-version tags are a deliberate, ecosystem-wide exception: maintainers accept the complexity in exchange for giving users automatic backward-compatible updates, and the pattern is explicitly endorsed in the Actions versioning guide.

The reproducibility tradeoff

Using `@v2` is convenient but not fully reproducible: the same workflow file can silently resolve to a different commit on different days. For most R package development this is the right

tradeoff—you benefit from fixes automatically. If you need exact reproducibility of your CI environment (e.g., for a frozen simulation study), you can pin to a specific commit SHA instead:

```
# Pinned to a specific commit; immune to tag movement
uses: r-lib/actions/setup-r@a51a8012b0aab7c32ef9d19bf54da93f3254335e # v2
```

The inline comment preserving the human-readable tag name is a community convention supported by dependency-update tools like Dependabot and Renovate.

For `Morrison-Lab/gha` workflows, **the floating major tag (such as `@v2`) is the default recommendation**. Reach for SHA pinning only when exact CI reproducibility is a stated project requirement.

Setting Up GitHub Actions

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

Essential Workflows

1. R CMD check (most important):

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

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

2. Test coverage:

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

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

3. Package website:

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

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

Interactive Setup

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

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

Allowing Actions to Create Pull Requests

Some workflows open pull requests themselves — for example, a scheduled job that bumps a submodule pointer or updates generated files. By default, GitHub blocks workflows that authenticate with the built-in `GITHUB_TOKEN` from creating or approving pull requests, and such a workflow fails with a "GitHub Actions is not permitted to create or approve pull requests" error.

To allow it, go to **Settings > Actions > General > Workflow permissions** in the repository and check **Allow GitHub Actions to create and approve pull requests**.

A few related points:

- The workflow also needs write access: either select **Read and write permissions** on the same settings page, or grant it per workflow with a `permissions:` block (`contents: write` and `pull-requests: write`).
- For repositories in an organization, the same setting exists at the organization level, and the organization setting must allow it before the repository-level setting can take effect.
- This governs only the built-in `GITHUB_TOKEN`; workflows that authenticate with a personal access token are governed by that token's own scopes instead.

A workflow that creates or comments on pull requests depends on this setting and grants itself the needed scopes with a job-level `permissions:` block (the block can sit either at the top of a workflow file or under an individual job, as here):

```
jobs:
  deploy:
    permissions:
      contents: write
      pull-requests: write
```

Reusable Workflow Collections

Several repositories collect reusable GitHub Actions and workflows for R projects, so you can call shared, community-maintained CI steps instead of writing your own:

- Morrison-Lab/gha: the Morrison Lab's collection of composite actions and reusable workflows for R-package and Quarto repositories, covering previews, publishing, test coverage, spell/link/character checks, bibliography DOI checks, and the Claude review bots used across its repos.
- r-lib/actions: the tidyverse team's actions and example workflows (`R CMD check`, test coverage, `{pkgdown}`, `{lintr}`), installable with `usethis::use_github_action()` as described in Section .
- eddelbuettel/r-ci: a portable, shell-script-based CI setup for R that works across GitHub Actions, Azure DevOps, Docker, and other providers from one configuration; the successor to the r-travis project.
- easystats/workflows: reusable workflows the easystats collective runs across its R packages, useful as worked examples of centralizing many repos' automation in one place.
- RMI-PACTA/actions: R and Docker actions plus example caller workflows; now archived ("no new work is expected"), but still readable as a reference for structuring an actions repo.

When several of your repositories need the same workflow, prefer calling a shared collection (or adding to ours) over copying workflow files between repos, for the same reason we avoid copy-pasted functions: fixes and improvements then land everywhere at once.

How GitHub Actions Workflows Work

When you set up a workflow, `usethis` creates a YAML configuration file in `.github/workflows/`. For example, `usethis::use_github_action("check-standard")` copies the `check-standard` example workflow from r-lib/actions to `.github/workflows/R-CMD-check.yaml`.

This workflow automatically runs when you:

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

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

Workflow Files and Security

> ⚠ Warning
>
> **Important Security Consideration**
> Workflow files (`.github/workflows/*.yaml`) have access to repository secrets and can execute code. Always review workflow files carefully before committing them, especially if copied from external sources.
> See the wai site's "Best Practices for Safe and Successful Use" section for guidance on working with workflow files using AI tools.

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

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

Pipeline Design Patterns

CI/CD pipelines are infrastructure code, and they face engineering challenges that application code rarely does: several pipeline runs can execute at once, any step can fail partway through, and a re-run must not repeat side effects the first run already performed. The patterns below apply to any CI system; the examples name the GitHub Actions and GitLab CI features that implement them.

Concurrency Control

Two pipeline runs can execute at the same time whenever two triggering events land close together — two pushes, a push plus a scheduled run, or two merged pull requests. Runs that only *read* the repository (tests, lint, spell check) are safe to run in parallel. Runs that *write* to a shared resource — a deployment environment, a `gh-pages` branch, a package registry, a bot comment — must be serialized, or the runs will interleave and corrupt the resource.

Prefer the CI system's built-in serialization over hand-rolled lock files:

- GitHub Actions: a `concurrency:` block assigns runs to a named group; at most one run per group executes at a time, and `cancel-in-progress: true` additionally cancels a superseded run instead of queueing behind it. Use cancellation for runs whose results only matter for the latest commit (reviews, previews), and plain queueing for runs that must all complete (deployments).
- GitLab CI: a `resource_group` serializes jobs that name the same group, across pipelines.

When the platform primitive doesn't fit — for example, a lock that must span two different workflows, or one that a human must be able to hold — an *advisory lock* (a label, a file committed to a branch, an issue assignment) can work, but it inherits the race condition described in the next section (Section): acquiring it is a check followed by an act, so two runs can both "acquire" it unless the acquisition step is atomic.

Fail-Open vs. Fail-Closed

When a guard mechanism itself fails — the lock service is down, the label query errors, a duplicate check times out — the pipeline must choose between two failure modes:

- **Fail-open**: proceed as if the guard had allowed it. Risks duplicate work (two deploys, two bot comments), but never stalls the pipeline.
- **Fail-closed**: abort the run. Never duplicates work, but a broken guard now blocks all work behind it until someone intervenes.

Choose based on the cost of each failure: fail-open when duplicates are cheap and annoying (a second copy of a bot comment), fail-closed when duplicates are expensive or irreversible (a production deploy, a package release, an email to a mailing list). Either way, make the choice explicit in the pipeline code and its comments, and log loudly when the guard misbehaves so the degraded mode is visible rather than silent.

Avoiding Time-of-Check to Time-of-Use Races

A *TOCTOU* (time-of-check to time-of-use) race is the gap between checking a condition and acting on it:

```
check: is there already a "claimed" label?  # no
                                            # <- another run claims here
act:   add the label, start working         # both runs now think they own it
```

Under concurrency, `check -> act` sequences are unsafe no matter how small the gap, because another run can change the condition between the two steps. Closing the gap requires an *atomic* operation — one the platform guarantees to check and act in a single step, such as a git push that fails if the remote ref moved (the push atomically verifies "my branch is current" and updates it), or creating a file with an exclusive-create flag. When no atomic primitive is available, fall back to the platform's serialization (Section) so that only one run executes the `check -> act` sequence at a time, and design the action to be idempotent (Section) so that losing the race is harmless rather than corrupting.

Idempotency

A pipeline step is *idempotent* when running it twice has the same effect as running it once. Idempotent steps make re-runs safe: after a transient failure, a canceled run, or a manual retry, nobody has to reason about what the first attempt already did.

Common CI applications:

- **Bot comments**: update one "sticky" comment in place instead of appending a new comment per run (see 1).
- **Deploys**: overwrite the target with the built artifact (`rsync --delete`, a force-push to `gh-pages`) rather than applying increments to it.
- **Releases and tags**: check whether the tag already exists and skip cleanly, rather than failing on the collision.

The test for idempotency is concrete: re-run the job from the CI interface and inspect the result — one comment or two, one release or an error?

Retries and Timeouts

Pipelines call networks constantly (package installs, API queries, link checks), so transient failures are routine, not exceptional. Retry transient operations with **exponential backoff** (wait 2s, 4s, 8s, … between attempts) and a **bounded number of attempts**, and do not retry *permanent* failures — a `404 Not Found` will not succeed on attempt three, and retrying it just slows the pipeline and hides the real error. This repository's own DOI checker is an example: `.github/scripts/check-bibliography-dois.R` uses `{httr}`'s `RETRY()` with three attempts, exponential pauses, and `terminate_on = c(404, 410)` so that genuinely-missing DOIs fail immediately while rate limits and timeouts are retried.

Give every job an explicit timeout (`timeout-minutes:` in GitHub Actions, `timeout:` in GitLab CI) sized to a small multiple of its normal runtime. The platform defaults are generous (GitHub Actions allows a job six hours), so a hung job otherwise burns runner minutes for hours before anyone notices.

Pipeline Decomposition

The function-decomposition principles from R Coding Practices apply to pipelines: split a pipeline into jobs along the lines of **independent failure and re-run** — if one part can fail while another succeeds, and re-running only the failed part would save meaningful time, they belong in separate jobs. Separate jobs also parallelize across runners and show up as separate checks on a pull request, which makes failures legible at a glance.

Keep a pipeline monolithic when its steps share expensive setup (one job that restores an `renv` library and then lints, tests, and builds beats three jobs that each restore it) or when the steps are meaningless in isolation.

Pass data between jobs through the platform's declared channels — job `outputs` and `needs:` in GitHub Actions, artifacts in both systems — not through side effects like pushing intermediate state to the repository.

Secret Management

- Store credentials in the CI system's secret store (repository or organization secrets in GitHub Actions, masked CI/CD variables in GitLab), never in the repository — including its history.
- Grant the least privilege that works: prefer the ephemeral, automatically-scoped token (`GITHUB_TOKEN` with an explicit `permissions:` block) over a personal access token, and scope any long-lived token to the one repository and permission it needs.
- Never print secrets. Masking hides known secret *values* from logs, but not derived values (a URL that embeds the token, a `base64` encoding), so don't echo variables that might contain them.
- Rotate long-lived tokens on a schedule, and immediately when a person with access leaves the project.

Observability

Debugging a failed pipeline run is archaeology: all evidence must have been captured at run time.

- **Log decisions, not just actions.** When a guard skips a step, a retry fires, or a fail-open path activates, print *why* — the condition checked and the value found.
- **Structure long logs.** Group related output (`::group::` / `::endgroup::` in GitHub Actions, collapsible sections in GitLab) so a reader can scan to the failing step.
- **Upload artifacts for anything a re-run can't reproduce** — rendered output, test snapshots, the exact `renv.lock` used — with a retention period matched to how long debugging normally lags (the platform defaults, 90 days on GitHub, are usually more than enough).
- **Summarize outcomes** where reviewers look: a step summary (`$GITHUB_STEP_SUMMARY`) or a sticky PR comment

  (1) beats a verdict buried in a thousand-line log.

Troubleshooting Failed Workflows

Before spending time debugging, check GitHub Status to confirm that GitHub's services are fully operational. Outages can affect any GitHub service, not just CI— for example, as of May 2026, recent incidents have included:

- **Actions**: Hosted runners experiencing high queue times or failures, causing workflows to stall or fail with no change to your code
- **Copilot**: Inability to start Copilot Cloud Agent sessions or view running sessions, making the coding agent appear unresponsive
- **Pull Requests**: New PR comment threads and line comments failing to be created, making it seem as though review comments have disappeared or are not saving

If a GitHub-wide issue is reported, wait for the outage to resolve before investigating further—the problem is not in your code.

If GitHub Status shows all services operational, check the "Actions" tab in your GitHub repository for detailed logs. Common causes of CI failures include:

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

For help addressing workflow failures, see the wai site's "Addressing Failing GitHub Actions Workflows" section.

Pull Request Comment Automation

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

Common Use Cases

PR comment automation is particularly useful for:

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

Comparison of Popular Actions

marocchino/sticky-pull-request-comment

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

Key features:

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

Typical usage:

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

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

hasura/comment-progress

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

Key features:

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

Typical usage:

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

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

thollander/actions-comment-pull-request

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

Key features:

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

Typical usage:

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

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

rossjrw/pr-preview-action

Specialized action for deploying pull request previews to GitHub Pages. Unlike the general-purpose comment actions above, this action handles the complete preview deployment lifecycle: building previews, deploying them to a GitHub Pages branch, and posting a comment with the preview link.

Key features:

- **Automated preview deployment**: Creates and deploys PR previews to a GitHub Pages branch (e.g., `gh-pages`)
- **Preview URLs**: Generates predictable preview URLs like `https://[owner].github.io/[repo]/pr-preview/pr-[number]/`
- **Sticky comments with links**: Posts and updates a comment with the preview URL and optional QR code for mobile access
- **Automatic cleanup**: Removes preview deployments when PRs are closed
- **Selective cleanup**: Can be configured to only remove previews for unmerged PRs, preserving merged PR previews for historical reference
- **Compatibility safeguards**: Designed to coexist with main branch deployments without conflicts

Typical usage:

```yaml
- uses: rossjrw/pr-preview-action@v1
  with:
    source-dir: ./docs/
    preview-branch: gh-pages
    umbrella-dir: pr-preview
```

Advanced usage - only remove unmerged PR previews:

```yaml
# Deploy preview when PR is opened/updated
- uses: rossjrw/pr-preview-action@v1
  if: contains(fromJSON('["opened", "reopened", "synchronize"]'), github.event.action)
  with:
    source-dir: ./docs/
    action: deploy

# Remove preview only for unmerged PRs
- uses: rossjrw/pr-preview-action@v1
  if: github.event.action == "closed" && !github.event.pull_request.merged
  with:
    source-dir: ./docs/
    action: remove
```

This selective cleanup pattern (from the pr-preview-action documentation) is particularly useful when you want to:

- Keep a historical record of what each merged PR changed
- Allow stakeholders to review merged changes after the fact

- Maintain preview URLs referenced in issue discussions or documentation

Best for: Projects using GitHub Pages for documentation, web apps, or other static sites where stakeholders benefit from previewing changes before merging. Essential when you want reviewers to see the rendered output (e.g., documentation sites, UI changes) rather than just the source code.

Feature Comparison

Table 1: Comparison of PR comment action features

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

Feature	marocchino/sticky	hasura/comment-progress	thollander/actions-comment	rossjrw/pr-preview
Works with push events		(requires PR number)	(requires PR number)	(requires PR number)
Progress tracking focus	(flexible)			
Deploy to GitHub Pages				
QR code generation				(optional)
Selective cleanup				(merged vs unmerged)

Choosing the Right Action

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

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

Use hasura/comment-progress when:

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

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

- You need simple comment posting without complex update logic
- You want to add emoji reactions to comments

- You're comfortable with the action's simpler update mechanism
- Your use case doesn't require the advanced features of the other options

Use rossjrw/pr-preview-action when:

- You need to deploy preview versions of your site/app to GitHub Pages for each PR
- You want stakeholders to review rendered output (documentation, UI) rather than just code
- You need automatic cleanup of preview deployments when PRs close
- You want to preserve previews of merged PRs for historical reference
- Your project uses GitHub Pages and benefits from preview environments

Security Considerations

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

Fast Package Installation with r2u

For GitHub Actions workflows running on Ubuntu, r2u provides a faster alternative to installing R packages from source. r2u offers pre-compiled binary packages for all CRAN packages on Ubuntu, which can dramatically reduce CI build times.

Key benefits for CI workflows:

- **Faster installation**: Binary packages install in seconds rather than minutes
- **Automatic dependency resolution**: All system dependencies are handled by `apt`
- **Complete CRAN coverage**: Over 30,000 CRAN packages available as binaries (as of early 2025)
- **GitHub Actions support**: Easy integration with Ubuntu runners

Quick example:

Instead of waiting several minutes for `install.packages("tidyverse")` to compile from source, r2u can install it as binaries in under 20 seconds.

How to use r2u in GitHub Actions:

The r2u project provides Docker containers and setup actions. For most workflows, you can use the r2u setup action:

```
- name: Setup r2u
  uses: eddelbuettel/github-actions/r2u-setup@master
```

Alternatively, use the rocker/r2u Docker containers directly in your workflow.

When to consider r2u:

- Your workflow installs many R packages
- Build times are a bottleneck in your CI pipeline
- You're running on Ubuntu (focal, jammy, or noble)
- You want to reduce GitHub Actions minutes usage

For complete documentation and setup instructions, see the r2u website.

Additional Resources

- GitHub Actions features overview
- r-lib/actions repository - R-specific actions and example workflows
- R Packages book: Continuous Integration
- GitHub Actions documentation
- Where to find help with r-lib/actions
- r-ci: Easy and portable CI for R packages - a portable shell script by Dirk Eddelbuettel for running R package CI across GitHub Actions, Azure Pipelines, CircleCI, and Docker
- r-codeberg - community organization supporting R package development and hosting on Codeberg
- r-codeberg migration guide - step-by-step tutorial on porting R package development, documentation, and CI workflows from GitHub to Codeberg
- Woodpecker CI - community-driven, container-based continuous integration system that integrates with Codeberg, Forgejo, and Gitea for R package checks
- shinycoreci: CI Testing for shiny - R package developed by the Posit Shiny team for testing the Shiny ecosystem across GitHub Actions runner environments
- GitHub Status - real-time status of GitHub services