Reproducible Environments

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

Package Version Control with renv

Introduction

Replicable code should produce the same results, regardless of when or where it's run. However, our analyses often leverage open-source R packages that are developed by other teams. These packages continue to be developed after research projects are completed, which may include changes to analysis functions that could impact how code runs for both other team members and external replicators.

For example, suppose we had used a function that took in one argument, such that our code contained `example_function(arg_a = "a")`. A few months after we publish our code, the package developers update the function to take in another mandatory argument `arg_b`. If someone runs our code, but has the most recent version of the package, they'll receive an error message that the argument `arg_b` is missing and will not be able to fully reproduce our results.

To ensure that the right functions are used in replication efforts, it is important for us to keep track of package versions used in each project.

`renv` can be to promote reproducible environments within R projects. `renv` creates individual package libraries for each project instead of having all projects, which may use different versions of the same package, share the same package library. However, for projects that use many packages, this process can be memory intensive and increase the time needed for a new users to start running code.

This chapter provides a quick tutorial for integrating `renv` into research workflows. For more detailed instructions, please refer to the `renv` package vignette.

Implementing renv in projects

Ideally, `renv` should be initiated at the start of projects and updated continuously when new packages are introduced in the codebase. However, this process can be initiated at any point in a project

To add `renv` to your workflow, follow these steps:

1. Install the `renv` package by running `install.packages("renv")`
2. Create an RProject file and ensure that your working directory is set to the correct folder
3. In the R console, run `renv::init()` to initialize renv in your R Project
4. This will create the following files: `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R`. Commit and push these files to GitHub so that they're accessible to other users.
5. As you write code, update the project's R library by running `renv::snapshot()` in the R console
6. Add `renv::restore()` to the head of your config file, to make sure that all users that run your code are on the same package versions.

Configuring renv settings

The `renv/settings.json` file created during initialization allows you to customize how `renv` behaves in your project. One useful setting is `snapshot.dev`, which controls whether development dependencies are included by default when calling `renv::snapshot()` or `renv::status()`.

Reducing startup messages

When working on projects, you may encounter startup messages indicating that `renv` is out of sync with the lockfile. To reduce these messages in most projects, add the following setting to `renv/settings.json`:

```
"snapshot.dev": true
```

This setting (available in `renv` version 1.1.6 and later) includes development dependencies in snapshots by default, which helps keep the lockfile aligned with your actual usage and eliminates many synchronization warnings.

If startup messages about being out of sync persist after enabling this setting, use `renv::restore()` to sync your local library with the lockfile, or `renv::snapshot()` to update the lockfile with your current package versions.

For more details on `renv` configuration options, see the official renv documentation.

Using projects with renv

If you're starting to work on an ongoing project that already has `renv` set up, follow these steps to ensure that you're using the same project versions.

1. Install the `renv` package by running `install.packages("renv")`
2. Pull the most updated version of the project from GitHub
3. Open the project's RProject file
4. Run `renv::restore()`. Some projects call `renv::restore()` at the top of a setup or config script; if yours does, running that script restores the environment for you.
5. This will pull up a list of the project's packages that need to be updated for you to be consistent with the project. The console will ask if you want to proceed with updating these packages - type "Y" to continue.
6. Wait for the correct versions of each package to install/update. This may take some time, depending on how many packages the project uses.
7. Your R environment should now be using the same package versions as specified in the `renv` lock file. You should now be able to replicate the code.
8. If you make edits to the code and introduce new/updated packages, see the section above for instructions on how to make updates.

Fast Package Installation on Ubuntu with r2u

For users working on Ubuntu Linux systems, r2u provides an alternative approach to package management that complements or replaces `renv` in certain workflows.

What is r2u?

r2u integrates R package installation with Ubuntu's `apt` package manager, providing pre-compiled binary packages for all CRAN packages. This integration offers several advantages over traditional R package installation:

- **Fast installation**: Binary packages install in seconds, not minutes
- **Automatic dependency resolution**: System dependencies are handled automatically
- **System-level integration**: Packages integrate with Ubuntu's package management
- **No compilation required**: Eliminates the need for build tools and compilers

Supported Ubuntu versions (as of early 2025):

- Ubuntu 24.04 "noble" (amd64 and arm64)
- Ubuntu 22.04 "jammy" (amd64)
- Ubuntu 20.04 "focal" (amd64, archived - no new updates)

When to use r2u vs. renv:

- **Use r2u when**: You work primarily on Ubuntu systems, want fast package installation, or need system-level package management
- **Use renv when**: You need cross-platform reproducibility, work on macOS/Windows, or require project-specific package libraries
- **Use both when**: You want fast installation on Ubuntu while maintaining reproducibility through renv lockfiles

Basic setup:

r2u requires a one-time system setup. See the r2u website for detailed installation instructions.

Once configured, R's `install.packages()` automatically uses binary packages from r2u, making package installation much faster.

Docker containers:

The Rocker Project provides `rocker/r2u` Docker containers with r2u pre-configured, ideal for reproducible containerized workflows.

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

Managing R Versions with rig

rig is an R installation manager that makes it easy to install, manage, and switch between multiple versions of R on macOS, Windows, and Linux.

When working across projects that require different R versions, or when you need to match a specific R version for reproducibility, `rig` provides a simple command-line interface to handle R version management.

Key features:

- **Multi-version support**: Install and switch between multiple R versions
- **Cross-platform**: Works on macOS, Windows, and Linux
- **Simple CLI**: Install a version with `rig add release`, list installed versions with `rig list`
- **RStudio / Positron integration**: Registered R versions are automatically available in RStudio and Positron
- **Handles permissions for you**: switches to the root/administrator user as needed when installing R versions

When to use rig:

- You need a specific older R version to reproduce results from a published analysis
- You want to test code against multiple R versions
- You are setting up a new machine and want a streamlined R installation process

Basic usage:

```
# Install the latest release of R
rig add release

# Install a specific version
rig add 4.3.3

# List installed versions
rig list

# Switch the default R version
rig default 4.3.3
```

For full installation instructions and documentation, see the rig GitHub repository.

Clearing Docker Cache

When Docker images and containers accumulate on a server, they can consume significant disk space. To reclaim disk space by removing unused Docker data:

1. SSH into the Docker server.
2. Run `docker system prune -a`.

This command removes all stopped containers, all unused networks, all unused images (not just dangling ones), and all build cache.

> ⚠ Warning
>
> `docker system prune -a` removes **all** unused images, not just dangling ones. Confirm that no important containers are currently stopped before running it, as their associated images will also be removed.