Paper: OS04

How CI/CD Enhances the Development of R Packages in the Pharmaverse

Ben Straub, GSK, USA
Dinakar Kulkarni, Roche, USA

ABSTRACT

Continuous integration (CI) and continuous delivery (CD) are playing a pivotal role in ensuring that R packages in Pharma meet the highest standards. Focus is placed on ensuring that packages are fit for purpose for both internal systems as well as the various requirements for CRAN/BioConductor. In this paper, we discuss a few best practices that were adopted into making developer-friendly and efficient CI/CD pipelines and the impact that these pipelines have had in the open-source Pharma community and at Roche/Genetech. Two case studies of package and pipelines will be discussed - one on a beginner level and one on an advanced level. The first will be CI/CD workflows for the admiral R package, used for building Analysis Data Models (ADaMs) and the second case will be regarding the NEST framework, a collection of R packages for creating TLGs.

INTRODUCTION

Both small and large software projects can greatly benefit from employing CI/CD into their workflows. A lone software developer maintaining a project over many years can benefit from continually checking that their software runs under current and new dependent packages and operating systems. A large open-source projects can benefit from continually checking that contributors code in Pull Requests/Merges meet standards set by the project. Both can leverage the growing CI/CD open-source pipelines being developed across the myriad and vibrant open-source projects.

Below we will discuss a few scenarios where CI/CD has greatly benefited the open-source R package {admiral} and {nest} framework.

{admiral}

{admiral}, ADaM in R Asset Library, is an open-source R package that seeks to build a modularized toolbox to develop ADaMs according to CDSIC standards. The package was initially started by a collaboration between GSK and Roche and has since expanded to more companies and created additional extension packages for specific disease areas. The expansion has come with growing pains, but these pains are greatly reduced by CI/CD workflows.

Below we will look at two examples of CI/CD workflows employed in {admiral}:

1. Continuously checking template code that build standard ADaMs
2. Managing multiple CI/CD workflows across multiple packages repositories

ADaM Template Codes

The community of developers building {admiral} have a strong desire to help users build ADaMs in R. This is accomplished by robust documentation for {admiral}'s functions as well as User Guides often referred to as Vignettes. However, documentation can only go so far in demonstrating the fitness of the package. Robust code examples that can build entire ADaM datasets like ADAE or ADLB are incredibly helpful to showcase the modularized approach to {admiral}. These ADaM code examples are given the name **templates** within {admiral}. A user after installing {admiral} can simply call:

```
admiral::use_ad_template("ADVS")
```

to have a fully formed R script for creating an ADVS dataset with most common variables derived using {admiral} functions.

Unfortunately, these templates are not checked when building the R package as they would violate CRAN policies around examples run-times(citation). CRAN has many checks on a package, one being that only allows examples to execute under a certain time limit. To address this limitation from CRAN, a custom CI workflow was developed to test that these templates are indeed working as intended. We discuss this workflow in the following scenario.

Scenario: Update to an {admiral} function

The following scenario is common on {admiral}'s GitHub Repository. A user has identified an issue with a function in {admiral} that does a derivation for a BDS-Finding dataset. A developer on {admiral} updates the function in a feature branch as well as updates the BDS-Findings template. The developer then initiates a Pull Request of their feature branch into the main code branch of {admiral}. Before the Pull Request of the feature branch is allowed to be merged into main, a series of CI workflows are ran and must all pass. One of these CI process is the **Check Templates Workflow**. The workflow in its simplest form, has a new instance of R installed on a GitHub server with appropriate operating system and dependent packages installed. All the template code, for each ADaM, is then unpacked from the package and run on that GitHub server. If each template runs free of errors on the server, then the Check Templates Workflow is shown to have passed.

We have provided the first 10 lines of the `yaml` file that GitHub uses to run this workflow on the {admiral} GitHub repository. The yaml file is the syntax of choice to define each workflow on the CI/CD platform on GitHub, which is called GitHub Actions.

```
---
name: Check Templates

on:
  workflow_dispatch:
  pull_request_review:
    types: [submitted]

jobs:
  templates:
    name: Check Templates
    uses: pharmaverse/admiralci/.github/workflows/check-templates.yml@main
    if: github.event.review.state == 'approved'
    with:
      r-version: "4.0"
```

A few parts to note in the file:

- `uses:` - The core workflow instructions are housed on the `admiralci` GitHub repository. Essentially, the workflow in {admiral} pulls from the repository {admiralci} to run. This is a common practice across GitHub. More will be discussed on this `admiralci` repository later in the **Common CI/CD framework** framework section and how to leverage it in our family of repositories.

- `if: github.event.review.state == 'approved'` - This workflow is only run after a reviewer of the code **Approves** the Pull Request. As these runs of the templates can be resources intensive it was decided to only run after a Pull Request is approved.
- `r-version:` The Version of R can be customized as needed in the package repository.

In summary, every time there is a need to add or update code to {admiral} a Pull Request must be initiated. For that code in the Pull Request to be successfully merged in all of the CI process must pass. One of those CI workflows is around the ADaM template code that can *NOT** be executed with the common package checks for CRAN. The Check Templates makes sure that this template code is robust and current by continuously checking the templates as {admiral} code is developed and ready for user use.

Common CI/CD framework

{admiral} has grown into multiple extension packages for specific disease areas as well as two dependency packages for testing and developer tools. Multiple companies with varying expterise of developers are actively involved on a daily basis with a shared goal of helping users to build ADaMs in R. To maintain integrity within the family of admiral a common framework of CI/CD workflows have been developed and housed in the GitHub repository admiralci. This CI/CD framework is not optional and is required to be used at a minimum. Additional CI/CD checks can be added if needed to the repository.

In each of the repositories, there is a `.github/workflows` folder and in it you will find a file called `common.yml`. This file deploys the same CI/CD workflows across all repositories regardless of the package name or purpose. The workflows are around code style, linting, spell check, link validation, creating a validation report, R-CMD checks, checking documentation is current and publishing a {pkgdown} website.

A partial glimpse is provided below that showcases four CI workflows: code style, spelling, validation and R-CMD checks. Please note, the R-CMD check are the common checks employed by CRAN to test your package. Failure of any of the CRAN tests mean rejection of your package. You can run these locally on your machine or in our case run them as CI checks!

```yaml
---
jobs:
  style:
    name: Code Style
    uses: pharmaverse/admiralci/.github/workflows/style.yml@main
    if: github.event_name == 'pull_request'
    with:
      r-version: "4.0"
  spellcheck:
    name: Spelling
    uses: pharmaverse/admiralci/.github/workflows/spellcheck.yml@main
    if: github.event_name == 'pull_request'
    with:
      r-version: "4.0"
  check:
    name: Check
```

```
    uses: pharmaverse/admiralci/.github/workflows/r-cmd-check.yml@main
    if: github.event_name == 'pull_request'
  validation:
    name: Validation
    uses: pharmaverse/admiralci/.github/workflows/r-pkg-validation.yml@main
    if: github.event_name == 'release'
    with:
      r-version: "4.0"
```

A few parts to note in this partial file glimpse of `common.yml`:

- `uses:` - Just like in the above **Check Templates** section this `common.yml` file references the workflows housed on `admiralci`. Readers are encouraged to go the admiralci repository and explore the yaml syntax for these checks.
- `if: github.event_name == 'pull_request'` - A slight deviation from **Check Templates**. This CI workflow is triggered when the Pull Request is started, which differs from when the Pull Request is Approved. This is helpful for developers as they will get feedback almost immediately if their code is meeting the admiral standards.
- `if: github.event_name == 'release'` - When a Release is initiated on a GitHub Repository this will trigger an event where a custom validation report is created for the release.
- `r-version: "4.0"` - The code style and spelling CI workflows both make use of R version 4.0, i.e. the GitHub Server that runs these checks only installs R version 4.0.

- Missing R version for Check Workflow. The check workflow does not have an `r-version` listed. This workflow installs the three latest version of R. At the time of this paper, that was 4.0, 4.1 and 4.2. A snapshot of the dependent packages closes to the release of those version of R is also installed. The package is then run on these three instances of R and if it installs properly and passes all CRAN checks, then the CI workflow is passed.

Maintaining the integrity of multiple dependent and extended packages across multiple repositories is a challenge. Introducing different levels of skills with your developers compounds potential issues. Developing a common framework of CI/CD housed in the admiralci repository has allowed us to continually provide feedback to both new and veteran developers while also ensuring that our code base is maintained to a high-level of integrity. At the same time, we have a common CI/CD framework housed in a central place has enabled us to quickly bring online additional pharmaverse packages.

NEST

The NEST framework is comprised of a suite of R packages that are used for clinical reporting purposes. `{teal}`, `{tern}`, and `{rtables}` are at the core, with several other supporting R packages that make up the framework.

In order to maintain the highest levels of quality, security, and compliance, a common set of CI/CD pipelines are applied across the board. There are some pipelines and workflows that cater to specific use cases in the development of NEST which are discussed below in brief.

Shiny App Testing

Shiny is a popular web application framework for R that allows developers to create interactive data visualizations and user interfaces. As with any web application, it is important to thoroughly test Shiny

apps to ensure that they function correctly and provide a good user experience. Automated testing using Continuous Integration/Continuous Deployment (CI/CD) can help to streamline the testing process and ensure that Shiny apps are thoroughly tested before they are deployed.

Automated Shiny app testing using CI/CD typically involves the following steps:

Test Automation: Automated tests are created to test various aspects of the Shiny app, such as the user interface, data input and output, and interactions between different components of the app. Tests are typically created using tools such as `{rSelenium}`, `{shinytest}`, `{shinytest2}`, and `{testthat}`.

Version Control: The Shiny app code is stored in a version control system, such as Git. This allows multiple developers to work on the code simultaneously and tracks changes over time.

Continuous Integration: The Shiny app code is automatically integrated with other code changes and tested to ensure that it works correctly with the rest of the software system.

Continuous Deployment: The Shiny app code is automatically deployed to a staging environment, where it can be further tested by developers and stakeholders before it is released to production.

Monitoring: Once the Shiny app is in production, it is important to monitor it for issues or bugs that may arise. Automated monitoring tools can be used to detect issues and alert developers if there are any problems.

Automated testing using CI/CD can provide several benefits for Shiny app development. For example, it can help to:

Catch errors and issues early in the development process, before they become a problem in production. Ensure that the Shiny app functions correctly across different browsers and devices. Provide a consistent and reliable testing process that can be easily replicated across different projects. Save time and resources by automating repetitive testing tasks. In summary, automated testing using CI/CD can help to ensure that Shiny apps are thoroughly tested and function correctly before they are deployed to production. By automating the testing process, developers can save time and resources and provide a better user experience for stakeholders and end users.

Shiny App Deployments

Shiny is a popular web application framework for R that allows developers to create interactive data visualizations and user interfaces. Once a Shiny app has been developed, it needs to be deployed to a production environment so that users can access it. Automated deployment using Continuous Integration/Continuous Deployment (CI/CD) can help to streamline the deployment process and ensure that the Shiny app is deployed consistently and reliably.

Automated Shiny app deployment using CI/CD typically involves the following steps:

Version Control: The Shiny app code is stored in a version control system, such as Git. This allows multiple developers to work on the code simultaneously and tracks changes over time.

Continuous Integration: The Shiny app code is automatically integrated with other code changes and tested to ensure that it works correctly with the rest of the software system.

Continuous Deployment: Once the code has been integrated and tested, it is automatically deployed to a staging environment, where it can be further tested by developers and stakeholders. If the app passes all tests, it is automatically deployed to production.

Configuration Management: Automated tools are used to manage the configuration of the Shiny app and its dependencies in the production environment. This helps to ensure that the app runs consistently and reliably across different servers and environments.

Monitoring: Once the Shiny app is in production, it is important to monitor it for issues or bugs that may arise. Automated monitoring tools can be used to detect issues and alert developers if there are any problems.

Automated deployment using CI/CD can provide several benefits for Shiny app development. For example, it can help to:

Ensure that the Shiny app is deployed consistently and reliably, reducing the risk of errors or downtime. Save time and resources by automating the deployment process, freeing up developers to focus on other tasks. Enable developers to easily roll back to a previous version of the app if issues arise. Provide a consistent and reliable deployment process that can be easily replicated across different projects. In summary, automated deployment using CI/CD can help to ensure that Shiny apps are deployed consistently and reliably, reducing the risk of errors or downtime. By automating the deployment process, developers can save time and resources and provide a better user experience for stakeholders and end users.

R Package Deployments

R is a popular programming language used for data analysis and statistical computing. R packages are collections of code, data, and documentation that can be easily shared and reused by others. Once an R package has been developed, it needs to be deployed to a repository where others can access it. Automated deployment using Continuous Integration/Continuous Deployment (CI/CD) can help to streamline the deployment process and ensure that the R package is deployed consistently and reliably.

Automated R package deployment using CI/CD typically involves the following steps:

Version Control: The R package code is stored in a version control system, such as Git. This allows multiple developers to work on the code simultaneously and tracks changes over time.

Continuous Integration: The R package code is automatically integrated with other code changes and tested to ensure that it works correctly with the rest of the software system. Tests can be run using tools such as Testthat, which is a popular testing framework for R.

Continuous Deployment: Once the code has been integrated and tested, it is automatically deployed to a package repository, such as CRAN or GitHub. This can be done using tools such as devtools or packrat, which automate the package building and deployment process.

Configuration Management: Automated tools are used to manage the configuration of the R package and its dependencies in the production environment. This helps to ensure that the package runs consistently and reliably across different systems and environments.

Monitoring: Once the R package is in production, it is important to monitor it for issues or bugs that may arise. Automated monitoring tools can be used to detect issues and alert developers if there are any problems.

Automated deployment using CI/CD can provide several benefits for R package development. For example, it can help to:

Ensure that the R package is deployed consistently and reliably, reducing the risk of errors or issues. Save time and resources by automating the deployment process, freeing up developers to focus on other tasks. Enable developers to easily roll back to a previous version of the package if issues arise. Provide a consistent and reliable deployment process that can be easily replicated across different projects. In summary, automated deployment using CI/CD can help to ensure that R packages are deployed consistently and reliably, reducing the risk of errors or issues. By automating the deployment process, developers can save time and resources and provide a better user experience for stakeholders and end users.

Continuous Compliance And Security

Continuous compliance and security for R packages involves ensuring that R packages meet compliance and security requirements throughout their development and deployment lifecycles. This can include ensuring that R packages meet regulatory requirements, follow best practices for security, and are up-to-date with the latest security patches and updates.

There are several steps that can be taken to ensure continuous compliance and security for R packages:

Establish Compliance and Security Requirements: The first step in ensuring continuous compliance and security for R packages is to establish the specific compliance and security requirements that the package must meet. This may include regulatory requirements, industry standards, or internal company policies.

Implement Security Best Practices: The R package code should be developed using best practices for security, such as following the principle of least privilege, using secure coding practices, and implementing secure authentication and authorization mechanisms.

Conduct Regular Security Scans: Regular security scans should be conducted to identify vulnerabilities in the R package code and dependencies. This can be done using tools such as R package security scanners, static code analysis tools, and vulnerability scanners.

Use Secure Dependencies: R packages should only use dependencies that are known to be secure and have been updated with the latest security patches and updates.

Perform Continuous Integration/Continuous Deployment (CI/CD): Continuous Integration/Continuous Deployment (CI/CD) can help to ensure that R packages are continuously tested and deployed using a consistent and reliable process. This can help to identify and address compliance and security issues early in the development process.

Conduct Regular Audits: Regular audits should be conducted to ensure that R packages meet compliance and security requirements. This can include internal audits as well as third-party audits.

Implement Access Controls: Access controls should be implemented to ensure that only authorized users have access to the R package code and related data.

Continuous compliance and security for R packages is important to ensure that packages are secure and meet regulatory requirements. By following best practices for security and implementing regular security scans, developers can identify and address vulnerabilities early in the development process. By using secure dependencies and implementing access controls, developers can help to ensure that R packages are secure throughout their development and deployment lifecycles.

TLG Catalog Rendering And Generation

In clinical reporting, tables, listings, and graphs are often used to present the results of clinical trials and other medical research studies. These types of data visualizations can help to convey complex information in a clear and concise way, making it easier for healthcare professionals and researchers to understand and interpret the results of a study.

To ensure that tables, listings, and graphs are rendered correctly in clinical reports, it is important to use a consistent and reliable process for generating these visualizations. Continuous Integration/Continuous Deployment (CI/CD) is an approach to software development that can be used to automate the process of generating these visualizations, making it faster and more reliable.

The process of rendering tables, listings, and graphs in clinical reporting using CI/CD typically involves the following steps:

1. Data Extraction: Data is extracted from various sources, such as electronic health records or clinical trial databases, and then transformed into a format that can be used to generate tables, listings, and graphs.
2. Code Development: Code is developed to generate the tables, listings, and graphs from the extracted data. This code is typically written using programming languages such as R or Python.
3. Version Control: The code is stored in a version control system such as Git, which allows multiple developers to work on the code simultaneously and tracks changes over time.
4. Automated Testing: Automated tests are developed to ensure that the code generates the correct tables, listings, and graphs for a variety of input data. This helps to catch errors early in the development process, before they become a problem in the final clinical report.
5. Continuous Integration: The code is automatically integrated with other code changes and tested to ensure that it works correctly with the rest of the software system.
6. Continuous Deployment: The code is automatically deployed to a staging environment, where it can be further tested by clinical researchers and healthcare professionals before it is finalized for publication in a clinical report.

By using CI/CD to automate the process of generating tables, listings, and graphs in clinical reporting, healthcare organizations and research institutions can reduce the risk of errors and ensure that these visualizations are consistent, accurate, and easy to interpret. This can ultimately improve the quality of clinical research studies and help healthcare professionals make more informed decisions about patient care.

Integration Testing

Integration testing is a type of software testing that ensures that different components of a software system work together seamlessly. In the context of R packages, integration testing is particularly important because packages typically consist of multiple functions that need to work together coherently to produce accurate and reliable results.

Integration testing for R packages typically involves running a suite of tests that exercise the package's functions in various combinations and scenarios. The tests may be designed to check the following aspects of the package:

1. Functionality: Does each function in the package work as intended? Do the functions produce the correct output for a range of input values?
2. Compatibility: Do the functions in the package work well together? Are there any conflicts or inconsistencies between the functions that need to be resolved?
3. Performance: How well does the package perform under a range of conditions? Does it handle large data sets efficiently? Are there any bottlenecks or performance issues that need to be addressed?
4. Usability: Is the package easy to use? Are the functions well-documented and easy to understand? Are there any usability issues that need to be addressed?

To perform integration testing for an R package, you can use a testing framework such as `{testthat}`. This framework provides a set of tools for defining and running tests, including functions for checking expected outputs, comparing data frames, and evaluating error messages.

To write integration tests for an R package, you should first identify the different scenarios in which the package will be used, and then create tests that exercise the functions in those scenarios. For example, you might create tests that:

Use multiple functions in the package to solve a complex problem Test the package's ability to handle missing or invalid input data Check the performance of the package on large datasets Test the package's compatibility with other R packages that it is likely to be used with As you write your tests, it's important to consider the different types of input data that the functions in the package may be used with. You should create test cases that cover a range of input values, including edge cases and extreme values that could cause the functions to fail.

Overall, integration testing is an important step in ensuring that an R package is reliable, accurate, and easy to use. By testing the package's functionality, compatibility, performance, and usability, you can ensure that it works seamlessly with other components of the R ecosystem and delivers accurate and reliable results to users.

CONCLUSION

We encourage readers new to CI/CD to also check out a workshop conducted by the authors called: An intro to CI/CD for R packages, which you can find linked in the Reference section below.

REFERENCES

- admiral
- pharmaverse
- nest
- CDISC

ACKNOWLEDGMENTS

- GSK and Roche
- Huge shout out to the developers of the pharmaverse building R packages.

RECOMMENDED READING

- Further Reading
  - GitHub Actions
  - GitLab CI
- Advanced Examples
  - r-lib/actions
  - `{admiralci}`
  - Docker
- Presentation built with Quarto
- This paper
- Presentation related to this paper
- Workshop at R/Pharma: An intro to CI/CD for R packages

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Ben Straub

Company: GSK

Address: 1000 Black Rock Rd

City / Postcode: Collegeville, PA 19426

Work Phone: (610) 917-3493

Email: ben.x.straub@gsk.com

Web: www.gsk.com