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 set up CI/CD workflows that continually check that their software runs under current and new dependent packages and operating systems. It is even possible to tailor these workflows to send an email if one of the workflows from an updated dependent package! Large open-source projects maintained by many developers can benefit from continually checking that developers code in Pull Requests/Merges meet standards set by the project. Letting the machines be the arbitrator for such things as code style, spell check and grammar can allow reviewers to give more focus to the actual content of the proposed changes. In the next sections, we will focus in on scenarios where CI/CD has greatly benefited the open-source R package {admiral} and NEST framework. We will look at specific use-cases on why the CI/CD workflows were implemented, beifely touch on how they were implemented and the impact on the particular project.

{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. **ADaM Template Codes** - Here we will look at a specifc CI workflow for continuously checking template code that build standard ADaMs

2. **Common CI/CD framework** - Here will look at how {admiral} manages 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 automated CI workflows are ran and must all pass as well as a manual review by another developer. 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 regardless if the change impacts the template, 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`. Please note the absence of curly brakets {} to denote the R package as `admiralci` is just a GitHub repository and not a R pacakge. 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 exists in all of the family of admiral packages and 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!

```
---
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 Initiated, 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'` - The Validation CI workflow is only triggered when a Release is triggered, where a custom validation report is created for the release. Please note how this is different from the other two GitHub events: `pull_request` and `approved`.
- `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.
- Please note the 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. We have provided a partial snapshot of the file, with modified repos to be fit on the page, that runs this workflow.

```
config:
 - {os: ubuntu-20.04, r: '4.0', repos: 'cran/snapshot/2021-03-31/'}
 - {os: ubuntu-20.04, r: '4.1', repos: 'cran/snapshot/2022-03-10/'}
 - {os: ubuntu-20.04, r: 'release', repos:'/cran/__linux__/focal/latest'}
```

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 which provide all the tools necessary for generating static reports as well as interactive applications for regulatory and exploratory submissions respectively.

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.

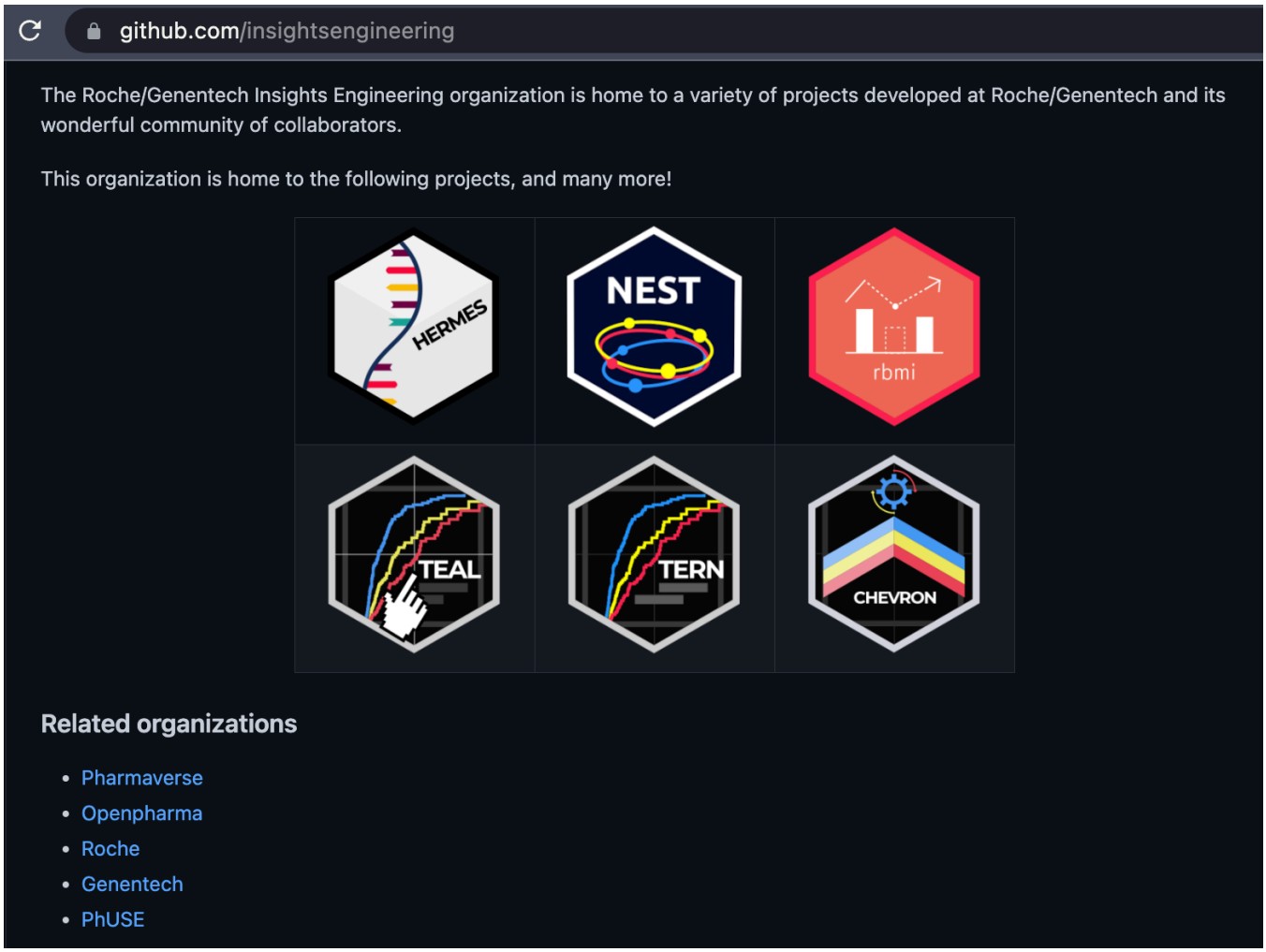


Figure 1: The Insights Engineering GitHub Organization, home to the NEST framework

In addition to several of the workflows that `{admiral}` uses, there are additional use cases for using CI/CD for NEST, some of which are discussed below.

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 in

testing whether a suite of inter-dependent and inter-related packages work together coherently to produce accurate and reliable results.

For NEST, the following factors are tested as part of the integration test suite:

1. *Functionality*: Do all packages work well together in a cohort? Do the functions produce the correct output for a range of input values?
2. *Compatibility*: Are there any conflicts or inconsistencies between the functions that need to be resolved?
3. *Performance*: How well do packages perform under a range of conditions? Do they handle large data sets efficiently? Are there any bottlenecks or performance issues that need to be addressed?
4. *Usability*: Are the packages easy to use? Are the functions well-documented and easy to understand? Are there any usability issues that need to be addressed?

Integration tests are orchestrated by a CI pipeline that:

1. Builds all packages from source.
2. Installs the builds in a library.
3. Executes `R CMD check` on packages.

Results from each of the above steps are reported in a report that package maintainers can review and make the necessary updates (bugfixes, performance improvements etc) to their packages.

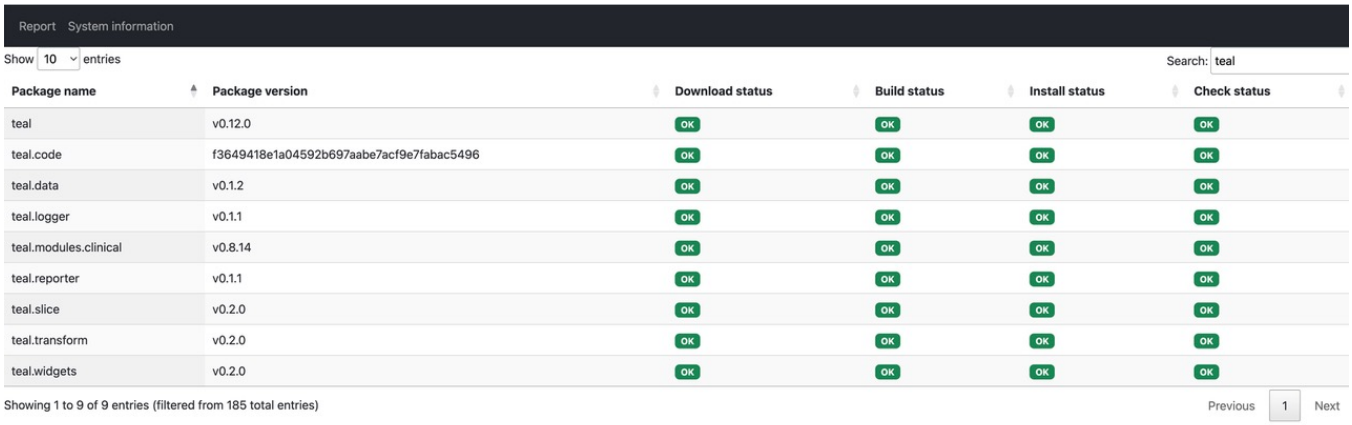


Figure 2: Integration test report

Web Application Testing

`{shiny}` is a popular web application framework for R that allows developers to create interactive data visualizations and user interfaces.

The `{teal}` framework is built on top of the `{shiny}` framework. As with any web application framework, it is important to thoroughly test the framwork as well as applications built using the framework to ensure they function correctly, provide a good user experience, and behave consistently and predictably.

Automated tests are created to test various aspects of `{teal}` app, such as the user interface, data input and output, and interactions between different components of the app. Tests are created using unit test frameworks such as `{shinytest}`, `{shinytest2}`, and `{testthat}`, and orchestrated using GitHub Actions and other similar CI/CD products.

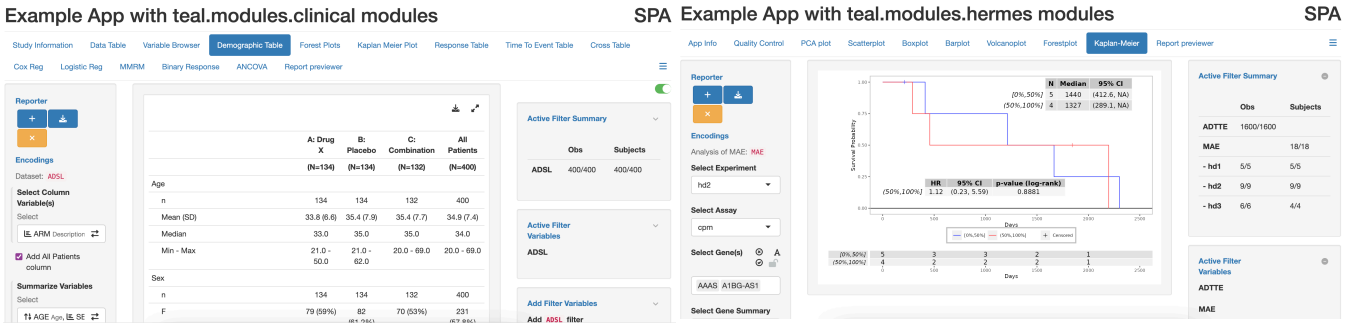
Benefits of running tests for `{teal}` and its applications via CI include:

1. Catching bugs, errors and other issues early in the development process, before they become a problem in production.
2. Ensure that `{teal}` components work correctly across different browsers and operating systems.
3. Save time and resources by automating repetitive testing tasks.

Web Application Deployments

Once `{teal}` apps have been developed, they need to be deployed to production environments so that users can access and use them. Automated deployment using CI/CD tools can help to streamline the deployment process and ensure that the `{teal}` app is deployed consistently and reliably.

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 a production environment.



(a) Efficacy `{teal}` sample application

(b) RNA-Seq `{teal}` sample application

Figure 3: Automatically deployed sample `{teal}` applications

Benefits of automatically deploying `{teal}` applications via CD include:

1. Ensure that `{teal}` applications are deployed consistently and reliably, thereby reducing the risk of errors or downtime. 1, Save time and resources by automating the deployment process, freeing up developers and operators alike to focus on other tasks.
2. Enable operators to easily roll back to a previous version of the app if issues arise.
3. Provide a consistent and reliable deployment process that can be easily replicated across different projects.

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:

1. *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.

2. *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.

3. *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.
4. *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.
5. *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.
6. *Implement Access Controls*: Access controls should be implemented to ensure that only authorized users have access to the R package code and related data.

	package	version	vulnerabilities	no_of_vulnerabilities
1	abind	1.4-5	NULL	0
2	backports	1.4.1	NULL	0
3	base64enc	0.1-3	NULL	0
4	brio	1.1.3	NULL	0
5	broom	1.0.3	NULL	0
6	bslib	0.4.2	NULL	0
7	cachem	1.0.6	NULL	0
8	callr	3.7.3	NULL	0
9	car	3.1-1	NULL	0
10	carData	3.0-5	NULL	0
11	checkmate	2.1.0	NULL	0
12	cli	3.6.0	NULL	0

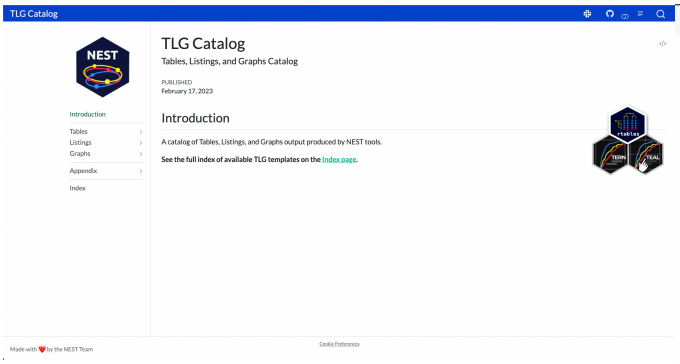
Figure 4: Dependency audit log

Rendering And Publishing of Documentation

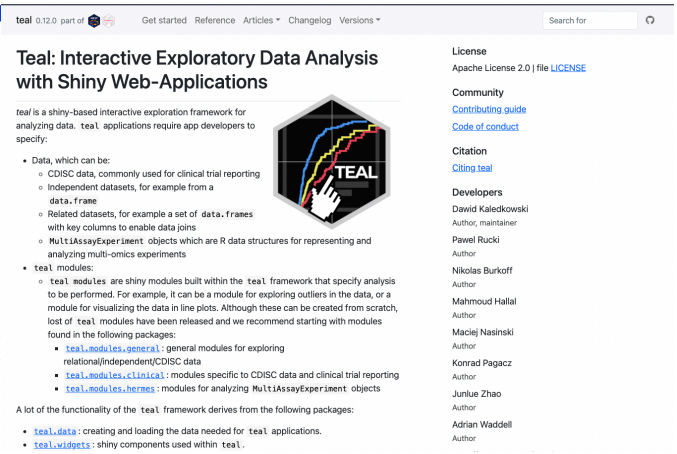
In clinical reporting, tables, listings, and graphs (TLGs) are often used to present the results of clinical trials and other pharmaceutical research studies. These types of data visualizations can help to convey complex information in a clear and concise way, making it easier for scientists, researchers,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

To ensure that TLGs are rendered correctly in clinical reports, it is important to use a consistent and reliable process for generating these visualizations. All NEST products leverage a variety of tools for documentation generation and publishing. For a catalog of TLGs as well as for creation of user guides, Quarto is used to render the documentation, while `{pkgdown}` is used to generate R package documentation.

CI is used for building the documentation and also for testing structural integrity of the documention (i.e. are URLs and hyperlinks correct, can the documentation be rendered, etc). CD is used to publish the documentation onto a web server.



(a) TLG Catalog



(b) `{teal}` package documentation

Figure 5: Auto-generated and auto-published documentation

R Package Deployments

After the NEST packages have been automatically tested with CI, they are deployed into staging and non-production environments where they are used for development and user-acceptance testing purposes.

Furthermore, when the NEST packages are ready for a production release, they are automatically built and deployed into R package repository for general consumption.

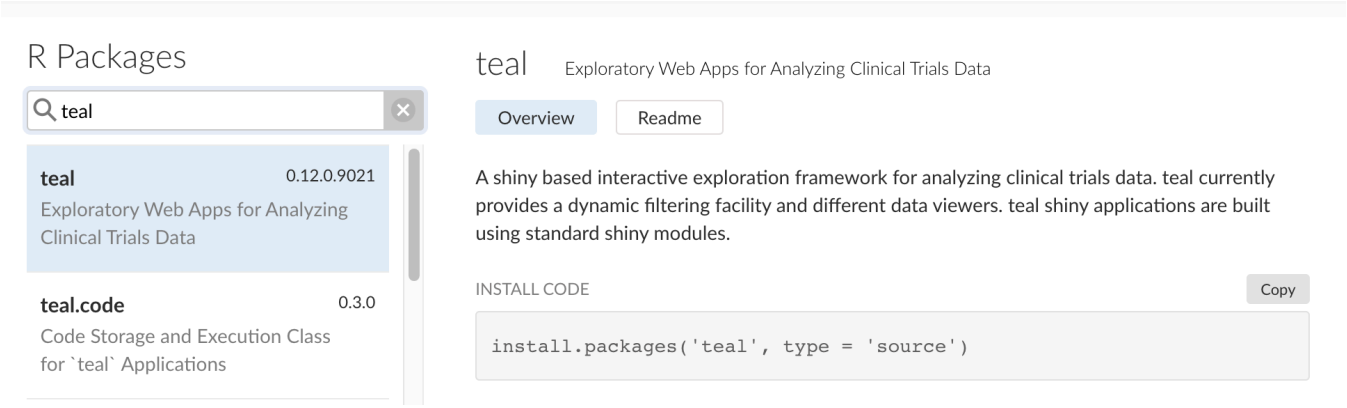


Figure 6: NEST R package repository

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: **+1 (610) 917-3493**
Email: **ben.x.straub@gsk.com**
Web: **www.gsk.com**