Paper OS06

R Shiny from Code to Cloud

Abu Zafar Mohammad Sami, mainanalytics GmbH, Sulzbach/Taunus, Germany
Yann Féat, mainanalytics GmbH, Sulzbach/Taunus, Germany

ABSTRACT

R® Shiny is a commonly used tool for transforming clinical trial analysis from static to dynamic, which is nowadays of high interest to stakeholders. However, the development and deployment of a complex R Shiny app can be challenging. Following good software development practices in a collaborative effort is essential. Additionally, cloud technologies are popular for managing the app lifecycle. The advantages include improved collaboration, excellent reproducibility, accessibility, and enhanced security. In the presentation, we will explore the design of a Shiny app with modules that can flexibly be added or updated and demonstrate how to create comprehensive documentation. Furthermore, we will present a collaborative process for developing an app in a cloud environment, using a CI/CD (Continuous Integration and Continuous Deployment) pipeline with Azure DevOps which facilitates automatic linting, testing, R Markdown knitting, and deploying.

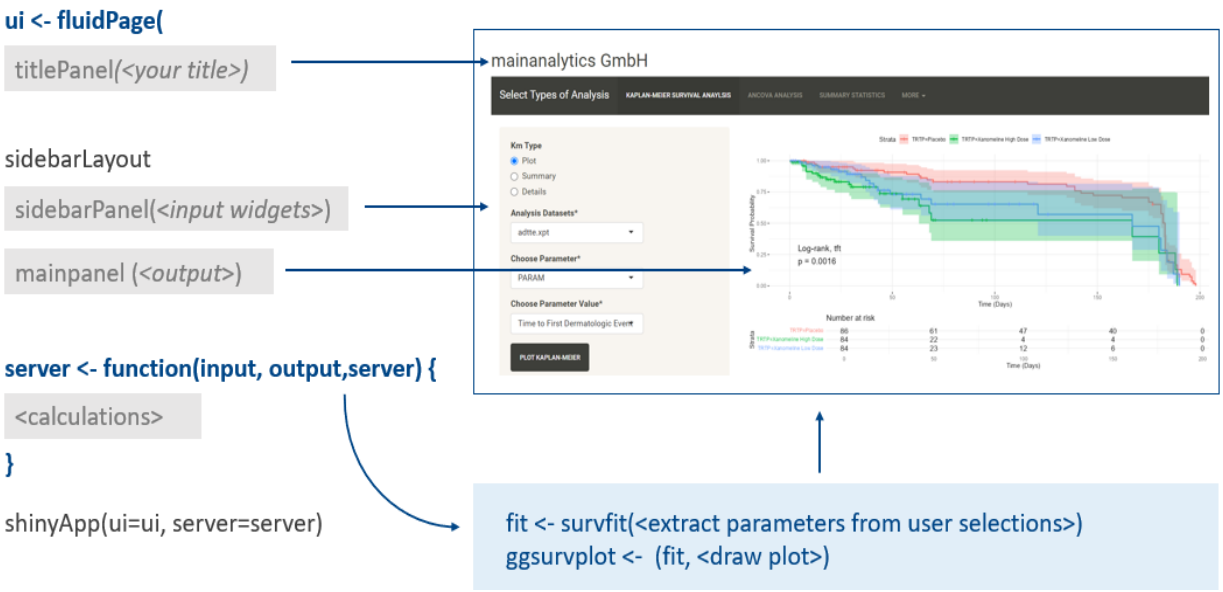
INTRODUCTION

Shiny is a web application framework used to make interactive data visualization apps, which has recently gained traction and popularity. In the pharmaceutical industry, the applications range from data exploration, safety analysis, clinical study reports, patient profiles, to (possibly) FDA submissions. The architecture of a prototypical Shiny app is quite straightforward, pairing a user interface (UI) and an application program stored on a local or remote server. In practice, as the product grows with the number of features, the design easily reaches the cliff of code complexity. It also becomes harder to test the application.

The scope of this paper is to demonstrate how we overcome the aforementioned challenges. First, each feature area was broken down into separate UI and server subcomponents, which were coupled together using Shiny modules reducing the complexity of adding/updating features to the app. In addition, it is recommended to follow a standard package development process, in order to better document and test the application. Another important consideration in the development of a Shiny app is the maintenance of good software engineering practices through all releases. Cloud technologies such as Azure DevOps add significant value in automating the integration and delivery/deployment processes. Indeed, this cloud approach not only dramatically reduces the time and cost to release and deploy a version of an application, but also ensures high-quality validation, through linting and testing. Finally, dynamic synchronization of the development process to/from the cloud environment provides greater flexibility in scaling development teams.

SHINY OVERVIEW

R Shiny is a web application framework for R, designed to build interactive web apps directly from R code. It allows users to create applications for data visualization, statistical analysis, and dashboards without requiring extensive web development skills with HTML, CSS, or JavaScript. The key features of Shiny are reactive programming, UI and server separation, customizability, scalability and the possibility to extend it..
Reactive programming allows any output to be automatically updated in response to changes in inputs. This makes it easier to create dynamic and responsive applications. Shiny separates the user interface (UI) from the server logic. The UI component defines the layout and appearance of the app, while supporting local interactivity using JavaScript. On the other hand, the server component contains the R code that processes input from the UI or internal resources and sends data back to the UI. Shiny allows a high degree of customization. Users can design UIs with various input widgets, such as sliders, text inputs, and drop-down menus, and combine them into components for interactive visualization. In addition, Shiny applications can scale from simple prototypes to complex, full-featured applications, sometimes in conjunction with other web applications. Users can either run them locally, host them on a remote server (e.g., using Shiny Server), or deploy them on a cloud platform such as Azure App Service. The following example shows a typical R Shiny application structure.



THE POSSIBLE USAGES OF SHINY APPLICATION IN CLINICAL RESEARCH

Possible use cases for Shiny apps include

- Data exploration
- Exploratory statistical analysis in studies (pre-clinical, clinical, …)
- Safety review of overall and patient-specific information from ongoing trials
- Clinical trial monitoring (safety analyses, data checks, …)
- Patient profiles (lab, AEs, …)
- FDA submissions

Many functions can benefit from them, such as biostatistics, patient safety, data management, medical writing and regulatory affairs. The provided interactivity allows these stakeholders to parameterize analyses without too much back-and-forth between them and the statistical programming teams.

THE CHALLENGES OF A SHINY APPLICATION

Developing and deploying R Shiny applications comes with its own set of challenges, especially as projects grow in complexity or need to be scaled up for wider deployment. Below are some of the most common challenges during R Shiny development and deployment.

CODE REACHING CLIFF OF COMPLEXITY

The following complexities are key to assume that the application development process is reaching a problematic level of complexity.

- Overloaded UI and server functions: if they grow too large, it can become hard to read, maintain, and debug them. The collaboration between team members, especially, can become challenging.
- Reactive overhead: reactive objects depend on each other in complex, often unintuitive ways. Changes to one input or reactive object can trigger a cascade of unexpected changes throughout the application. Debugging these reactivity chains becomes a nightmare because of hard-to-trace dependencies and unexpected reactive updates.
- Long loading times and unresponsive UI: as the complexity of the app increases, loading times may increase, and UI responsiveness may degrade, particularly if the app handles large datasets or complex computations.

QUALITY CONTROL (QC)

Ensuring a high-level of quality to the different stakeholders of a Shiny app is often paramount. Therefore, the behavior of the application should be as rigorously tested as possible. R Shiny lacks robust built-in testing frameworks and Shiny apps' interactivity and reactive behavior can be tricky. Organizations might rely on snapshot testing or on manual User Acceptance Testing (UAT), each coming with their own sets of problems.

These approaches are not in the scope of this paper. It is recommended to circumvent this challenge as much as possible by separating the business logic (data manipulations, statistical algorithms, …) from the Shiny logic (reading user input, updating the user interface, …) within server components. Moreover, the respect of good development practices should be controlled to mitigate risks and facilitate the maintenance of the app. Bad development practices can take the shape of functions that are too long and hard to test, bad separation between business and Shiny logics, poor documentation, “unclean” code syntax…

CONTINUOUS DEVELOPMENT, DELIVERY AND DEPLOYMENT

The process of maintaining, updating, and delivering Shiny apps in an efficient and robust manner is essential, but possibly complex to handle for a team without software engineering expertise.

Deploying Shiny apps often requires a dedicated server or a cloud service. Configuring and maintaining the server infrastructure can be time-consuming and complex, especially if scaling for a larger audience. In addition, every update/upgrade of a features in the app requires testing, integration, releasing, and deployment processes. Any mismanagement can lead to increases in delays and costs.

Furthermore, it should be ensured that the development, test, delivery and/or deployment environments are similar enough to ensure that the confidence provided by QC processes applies to all of them. This issure of reproducibility/robustness is especially important when delivering applications to another team, a client, a peer reviewer or a regulatory authority.

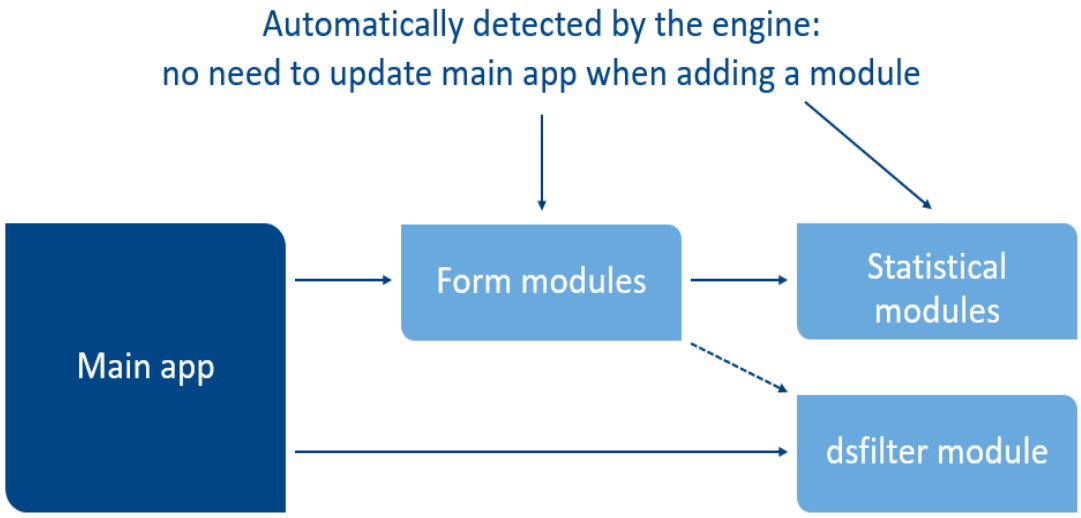
SHINY APPLICATION DEVELOPMENT WORKFLOW

SHINY MODULES

Our app is divided into Shiny modules for individual features. Each of them encapsulates a piece of the app’s UI and associated server logic, making the app easier to manage, and has a unique pair name They can only be run as part of an app, and they are composable: one module can contain another.

Input and output IDs in Shiny apps share a global namespace, meaning, each ID must be unique across the entire app. The modularization approach resolves the namespace problem present in large apps, by making it harder for two UI functions to be given the same ID, as they would usually be part of different modules.

Another motivation is to break up a complicated Shiny app into separate modules, each of which can be reasoned about independently; such modules have no direct potential for reuse, but they serve an important purpose within an app. Each module runs independently within the app, meaning that problems or changes within one module are less likely to affect other modules. This is particularly useful when handling different sets of data or logic within an app. Our app follows this modularization principle, if there is a need to scale or add a functionality, it can be easily extended with new modules without affecting the rest of the application. This improves the overall scalability, ease of collaboration, and flexibility. The below figure depicts the design of our application with its modular approach.

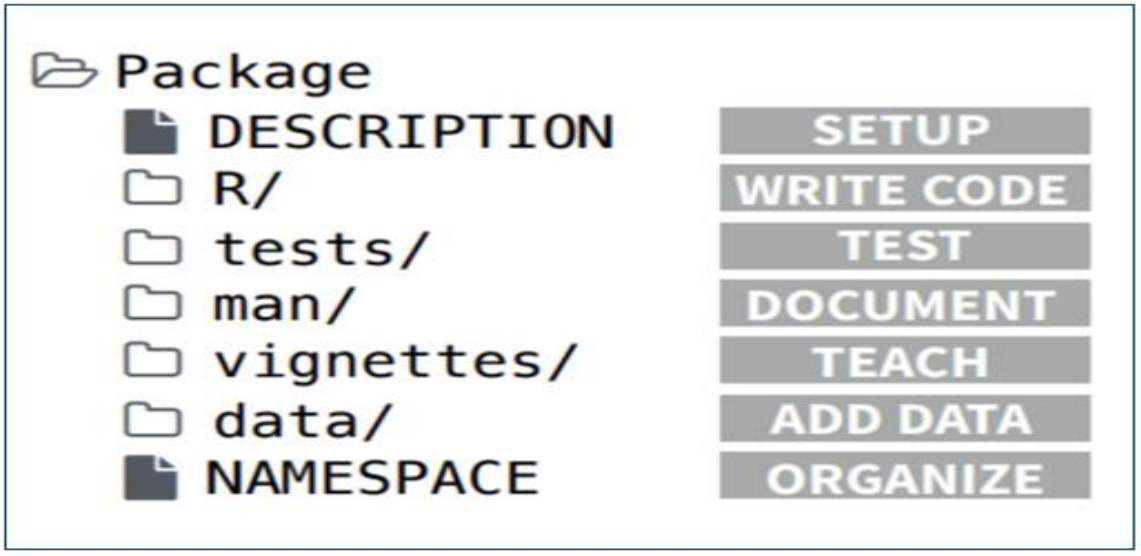


In addition, the modularization approach improves testing as each module is self-contained, making it easier for that code to be tested in isolation. This allows developers to write tests for specific parts of the app, ensuring that each feature works correctly before integrating it with the rest of the app.

However, perhaps the main benefit of using modules is not to organize the code; but to help us clarify our thinking.

PACKAGE DEVELOPMENT INITIALIZATION

Packaging an R Shiny app is a great way to organize, distribute, and maintain the code of a Shiny application, especially when sharing it across multiple projects or among team members. By packaging a Shiny app into an R package, we can also take advantage of the associated R tools for version control, dependency management, deployment and testing (such as R CMD checks). We have packaged our app in a conventional package folder structure. This has the added benefit of providing all the associated automation features for continuous development, testing and deployment when managing the app on the cloud.
There are multiple packages useful for package development. We have used the `{usethis}` package (handily automates repetitive tasks) for packaging the Shiny app and organized the files into the following 7 directories:



PACKAGE DEVELOPMENT DESCRIPTION

The DESCRIPTION file is a critical component of an R package to store important metadata about a package, including its name, version, author, dependencies, and more. Every package must have a DESCRIPTION file and it is used by R when installing and managing it. In fact, it is the defining feature of a package (RStudio and `{devtools}`consider any directory containing DESCRIPTION to be a package).

PACKAGE DEVELOPMENT NAMESPACE

The NAMESPACE file is used to define which of our functions should be made visible to the package user either for rendering (Shiny module functions) or for other purposes, and which objects are to be imported from dependencies. It is generated from in-line documentation using `{roxygen2}`. The content of our namespace file with functional description of the keywords is shown below.

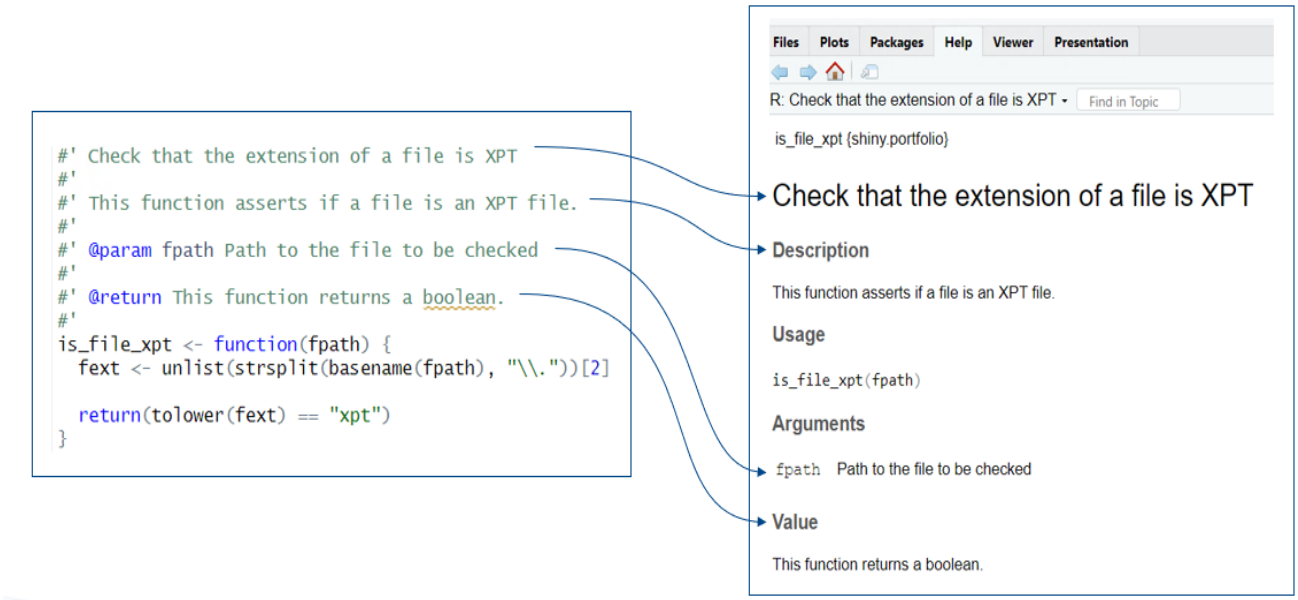
export
Functions of a package that makes it available for others to use

importFrom
Imports functions from other packages

```
export("%>%")
export(all_distinct_assert)
export(dsfilter_server)
export(dsfilter_ui)
export(form_anova_desc)
export(form_anova_server)
export(form_anova_ui)
export(form_surv_desc)
export(form_surv_server)
export(form_surv_ui)
export(import_folder_adam)
export(stat_surv_km_desc)
export(stat_surv_km_server)
export(stat_surv_km_ui)
importFrom(emmeans,contrast)
importFrom(emmeans,lsmeans)
importFrom(forester,forester)
importFrom(survival,Surv)
importFrom(survival,survfit)
importFrom(survminer,ggsurvplot)
importFrom(survminer,surv_fit)
```

PACKAGE DEVELOPMENT MANUAL

The man (as in “manual”) folder contains the documentation files that become the “help” pages of the application package. These files describe each function, dataset, or object in the package. They are written in the R documentation (Rd) format. The contents of these files are automatically generated by `{roxygen2}`.
Each Rd file in the man folder corresponds to a specific function or object in the package and contains information such as a description of what the function does, the arguments it takes, the values it returns, examples of how to use it, and more... Rd files use custom syntax, loosely based on LaTeX, and can be rendered to HTML, plain text, or PDF, as needed, for viewing in different contexts. Our app also has all its objects/features documented, as shown below.



The test files are the vital components of an application package development used to validate the functionality and correctness of your package through automated testing. We have used `{testthat}` to create and manage tests of the scripts, which provides a simple and flexible framework for writing unit tests. Testing R package is essential for ensuring that code behaves as expected, especially as the package evolves. Tests help catch bugs, ensure that functions produce the correct output, and confirm that edge cases are handled properly. Using them improve code robustness significantly. In the future, we might use tests as “specifications” for collaborators to develop new functionalities, in an approach called “test-driven development” (the tests are written before the tested code). Examples of unit tests in our app are shown below:

expect_equal(): Does code return the expected value?

expect_type(): Does code return an object inheriting from the expected base type?

expect_gt(): Does code return a number greater than the expected value?

expect_length(): Does code return a vector with the specified length?

```
datasets_unfiltered <- import_folder_adam(adam_folder_test)
datasets_filtered <- import_folder_adam(adam_folder_test, "adsl")

test_that("output type is list", {
  expect_type(datasets_unfiltered, "list")
  expect_type(datasets_filtered, "list")
})

test_that("output length is correct", {
  expect_length(datasets_unfiltered, 10)
  expect_length(datasets_filtered, 1)
})

test_that("output elements have all character columns
          converted to factors", {
  d_ufltr <- datasets_unfiltered[["adsl"]]
  d_fltr <- datasets_filtered[["adsl"]]
  expect_equal(dim(d_ufltr[, sapply(d_ufltr, class) == "character"])[2], 0)
  expect_equal(dim(d_fltr[, sapply(d_fltr, class) == "character"])[2], 0)
  expect_gt(dim(d_ufltr[, sapply(d_ufltr, class) == "factor"])[2], 0)
  expect_gt(dim(d_fltr[, sapply(d_fltr, class) == "factor"])[2], 0)
})
```

SHINY APPLICATION REPRODUCIBLE ENVIRONMENT

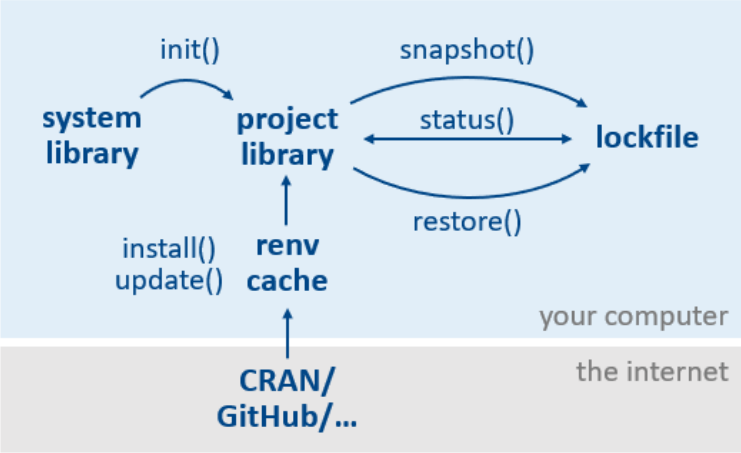
{renv} is an essential tool for us, specifically designed to manage the dependencies of any project developed with R. Its primary role is to participate in the reproducibility of environments across different computers or platforms, by providing us with tools that record and maintain each dependency, and that automatically install them from the desired source. This can be particularly useful for ensuring that the code runs across environments with the same versions of dependencies that it was developed and tested with. The basic workflow of {renv} is depicted below.

System library
Contains all installed packages

Project library
Gives each project its own independent collection of packages

Lockfile
Records metadata about packages to re-install them on a new machine

renv cache
Uses package cache, is shared across all projects



Some key benefits of renv are:

- **Reproducibility:** with renv, it is possible to take a snapshot of the current state of the project's package library into an "renv.lock" file which records the exact versions of the installed R dependencies. This makes it easy to recreate the same environment later, ensuring that the analysis or code can be reproduced with the same versions of R dependencies. In addition, renv.lock can be version-controlled, so that collaborators or future users can restore the environment (or at least its R dependencies) to the same state as when the snapshot was taken.
- **Environment isolation:** each project using {renv} can have its own isolated package library. This prevents conflicts between projects that might depend on different versions of the same package. The use of these isolated libraries reasonably ensures that packages installed for one project do not interfere with packages in another project or the global R environment.
- **Ease of version control and collaboration:** collaborators can use `renv::restore()` to install the exact versions of the packages defined in the lockfile, ensuring that they are working with the same

versions of dependencies as the rest of the team. This eliminates the common issue of sharing code on different platforms. Nevertheless, by sharing the renv.lock file, teams can keep track of package dependencies in a consistent and transparent way.

- **Cross-platform scaling:** The renv.lock file can be used to restore environments across different operating systems (e.g., macOS, Windows, Linux), making it easier to ensure consistency across teams working on different platforms. Moreover, the lockfile can track both CRAN and GitHub package versions, allowing users to restore even non-CRAN dependencies.
- **Combination with global and local libraries**: while `{renv}` creates isolated libraries, it can use a global cache for packages. This eliminates the need to reinstall the same package multiple times for different projects, thus saving disk space while maintaining the advantages of isolation.
- **Compatibility with continuous integration:** the renv.lock file can be used by CI/CD pipelines, ensuring that automated testing and deployment happens in a reproducible environment.

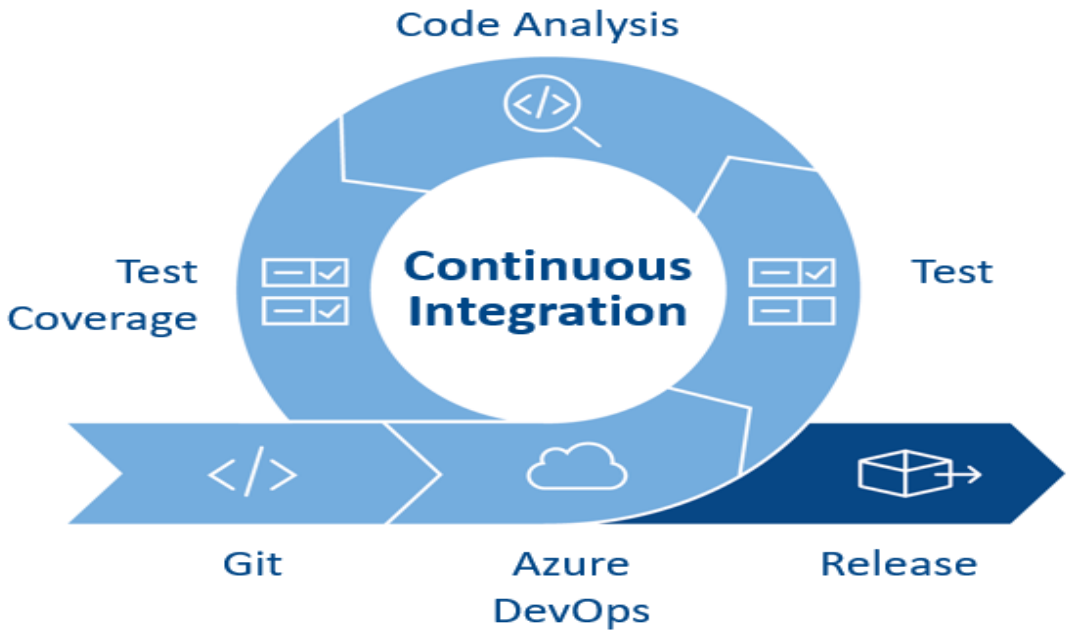
Although `{renv}` offers numerous benefits for managing R projects, there are some drawbacks to be aware of. Even though `{renv}` can use a global package cache to reduce duplication, each project still has its own package library, which can lead to higher disk space usage compared to using a shared global library. For projects with a lot of dependencies, the renv.lock file can become large and cumbersome, making version control slightly more challenging. Despite these limitations, we rely on `{renv}` to reproduce, version-control, and environment isolation of our Shiny package so that the application can be integrated, tested, deployed and released on the cloud with the same dependencies it was developed with.

SHINY APP IN CLOUD

CONTINUOUS INTEGRATION

Continuous Integration (CI) is the systematic practice of regularly integrating code changes from multiple developers into a shared repository using a version control tool (e.g., Git), accompanied by an automated process that builds, tests and validates the merged code in an isolated environments (e.g., Podman/Docker containers) before the merge occurs in the main branch. In Azure DevOps, this process is enabled by Azure Pipelines. This helps find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates. Collaboration is also enhanced, as contributors are encouraged to integrate small pieces of code on a regular basis.

The Microsoft Azure DevOps suite includes tools for version control, build automation, testing, release management, and more. With Azure Pipelines, developers can create custom workflows to automatically compile code, run tests, and provide immediate feedback to the team. The diagram below illustrates the CI process that we use to develop the Shiny app.



GIT

As part of the CI process, developers use the version control system Git to isolate their work into short-lived feature branches. When the feature is complete, the developer submits a pull request from the feature branch to the main branch. When the pull request is approved, the changes are merged into the main branch, and the feature branch can be deleted.

TEST

The business logic should be contained within easy-to-test functions.

The CI and CD pipelines automatically run unit tests as part of the R CMD checks to verify the quality and structure of an R package. They run a series of checks to ensure that an R package meets the necessary standards before it is merged with the main branch or deployed/delivered. This command is a vital tool in the Shiny package development process, as it helps the system identify problems in the package code, documentation, and overall structure. In fact, R CMD checks help development team by checking their package before it is sent to an environment where problems could cause damage. In addition, for each check, R CMD briefly describes what it does, what the most common problems are, and how to fix it. This helps developers quickly troubleshoot problems.

To incorporate these checks in a CI/CD pipeline, one needs to add this command to the appropriate configuration file (in YAML format, in the case of Azure Pipelines).

When the CI/CD process runs R CMD check, it performs a key checking of the shiny package, including:

- **File structure**: ensure that the package directory has the proper structure, including files like DESCRIPTION and NAMESPACE, as well as folders like man, and R.
- **Code and syntax**: check the R code for syntax errors, missing documentation, and adherence to some important best practices.
- **Unit tests**: if unit tests are included (e.g., using `{testthat}`), run these tests to ensure that the package works correctly.
- **Documentation**: ensure that every exported function in the package has appropriate documentation, and that the man directory contains correct Rd files.
- **Vignettes**: if the package contains vignettes (long-form documentation), verify that they are correctly written and that they compile without errors.
- **Dependency management**: verify that all dependencies listed in the DESCRIPTION file are available and that the package handles them correctly.

We have integrated the automated code analysis tool `{lintr}` for R into CI/CD processes, so that R code is consistently checked for syntactic and semantic errors, possible design issues and adherence to style guidelines. It helps developers write clean, readable, and well-structured code, providing developers with immediate feedback and helping teams maintain a high standard of code quality throughout the development lifecycle. Incorporating `{lintr}`into an Azure DevOps CI pipeline ensures that code quality issues are caught early in the development process.

In a CI/CD pipeline, we need to install `{lintr}` and other dependencies by updating the configuration file. Once the pipeline runs, Azure DevOps will show the results of the `{lintr}`analysis in the pipeline logs. If any issues are found, (the pipeline can either proceed (with warnings) or fail (depending on pipeline rules).

TEST COVERAGE

`{covr}` is incorporated in the Azure DevOps CI pipeline to measure test coverage. `{covr}`provides detailed reports on how much of the R code in the Shiny application is covered by tests, helping identify untested code. We set up R test coverage in Azure Pipelines by instructing it, through YAML configuration files, to run tests using `testthat}` as part of R CMD checks, then to collect and report test coverage using `{covr}`. Once the pipeline has run, Azure DevOps displays the coverage percentage and details in the ‘Code Coverage’ section of the pipeline summary, and developers/reviewers can drill down into the details of which lines or files are covered by the tests.

The process can be improved by restricting the coverage calculations to the files that contain the business logic, excluding the Shiny logic which can't easily be unit-tested.

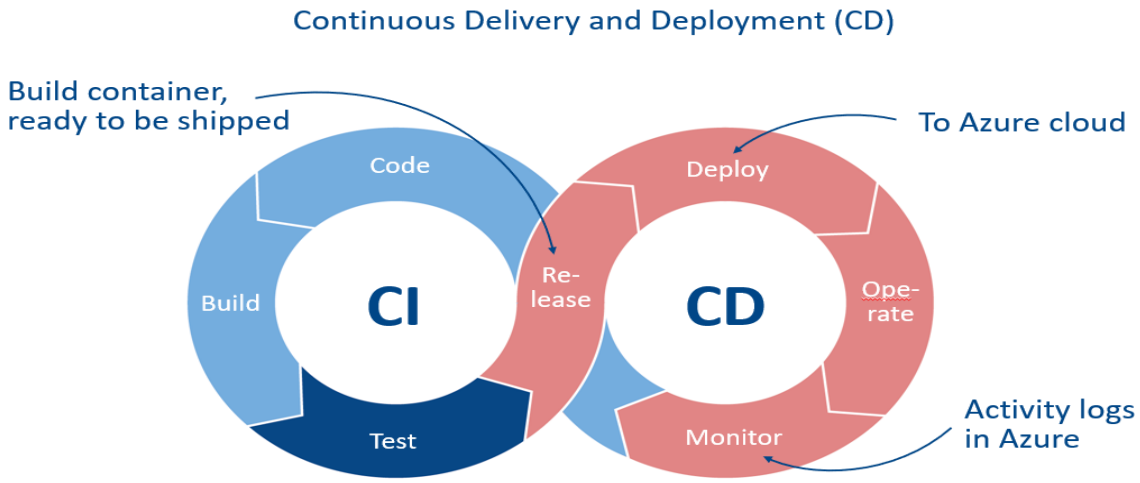
CONTINUOUS DELIVERY AND DEPLOYMENT

Continuous Delivery (CD) is the practice of automating the delivery process, ensuring consistent, efficient, and reliable releases of Shiny apps to production environments with minimal or without manual intervention. The end product of this is, in our case, a container image stored in our private Azure Container Registry. However, we have extended this process with a deployment step using Azure App Service, which automatically pulls the latest container image, so that each authorized collaborator can interact with the Shiny app, which makes it a continuous deployment process.

In fact, CD is an extension of CI since it automatically delivers all code changes to a test and/or production environment after the build stage. This means that, on top of automated testing, an automated release process is configured, and the application can be deployed at any time. However, it typically requires manual approval before going into the final production environment. On the other hand, Continuous Deployment goes one step further than continuous delivery where every change that passes automated tests is automatically deployed to production without any manual approval step. This means that builds, tests, and releases are fully automated, requiring no manual intervention after the code is committed and the quality and performance of the changes are immediately reflected in the live environment.

The key advantage of following a continuous delivery and deployment process is that it enables faster and more frequent releases with less manual intervention and ensures consistent deployment across environments, reducing the risk of human error. It also improves the scalability of Shiny applications by deploying them to cloud-based services such as Azure App Service.

The following diagram illustrates a continuous delivery and deployment process, that successfully followed a continuous integration pipeline.

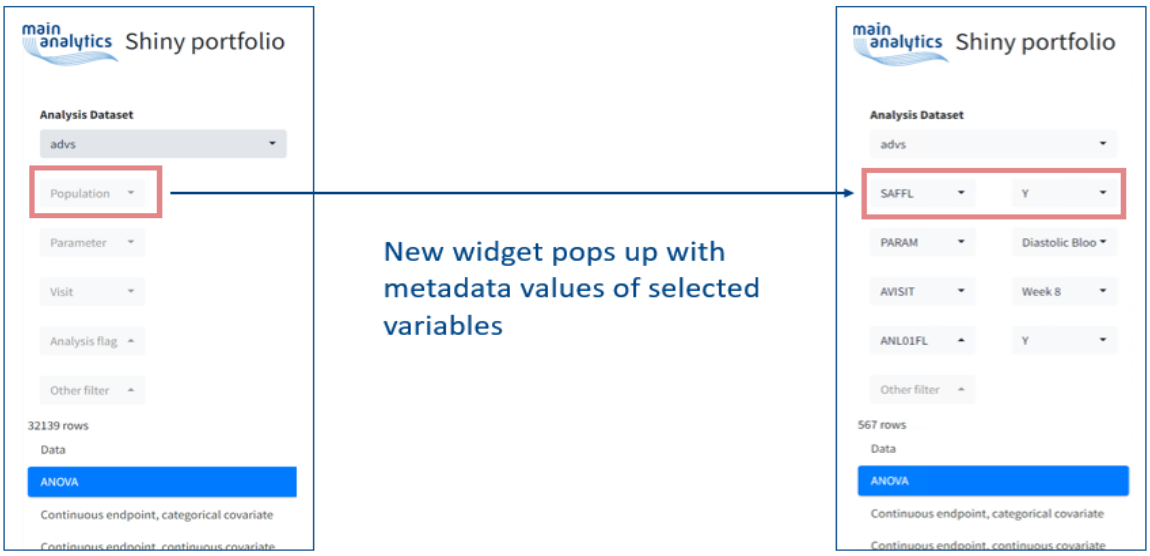


KEY FILES WITHIN THE PROJECT FOLDER FOR CI/CD

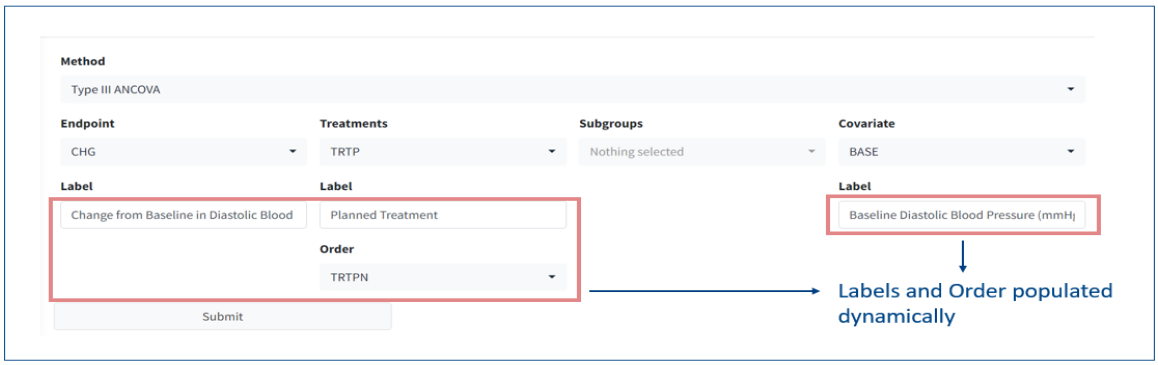
.dockerignore	List of files to be excluded when building the Docker image	Configuration files for the various utilities
.gitignore	List of files to be excluded by default from git operations	
.lintr	lintr configuration file (for code linting in the CI pipeline)	
.mega-linter.yml	Mega Linter configuration file (for general linting in the CI pipeline)	
.Rbuildignore	List of files to be excluded when the package is built	
jscpd.json	JSCPD configuration file (for copy-paste detection in the CI pipeline)	
azure-pipelines.yml	Azure DevOps configuration file for the CI pipeline	Configuration files for CI/CD and containerization
azure-pipelines-1.yml	Azure DevOps configuration file for the CD pipeline	
Dockerfile	Docker image configuration	
sys_deps.txt	List of system dependencies to be installed in CI pipeline and Docker image	

SHINY APPLICATION DASHBOARD

The application can populate new widgets with automated metadata values of selected variables. Moreover, it facilitates dynamic filling of numerical orders and labels. Below are displayed some examples of the selection process on our dashboard for producing ANCOVA tables and related graphics.



ANCOVA FORM MODULE



ANCOVA STATISTICAL MODULE

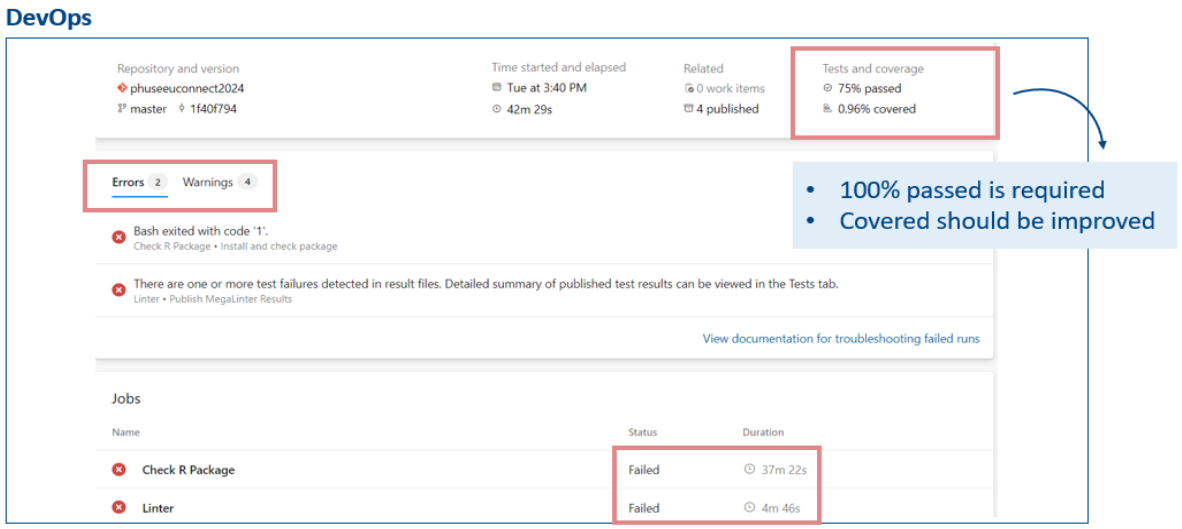
Type III ANCOVA of Diastolic Blood Pressure (mmHg) by Planned Treatment
adjusted for Baseline Diastolic Blood Pressure (mmHg)

	n	Mean	SE	95% CI	p-value
Placebo					
Model Estimates	72	75	0.88	[74, 77]	
Xanomeline Low Dose					
Model Estimates	60	75	0.96	[73, 77]	
Contrast / Placebo		-0.5	1.3	[-3.1, 2.1]	0.698
Xanomeline High Dose					
Model Estimates	56	76	0.99	[74, 78]	
Contrast / Placebo		0.41	1.3	[-2.2, 3]	0.755
Contrast / Xanomeline Low Dose		0.92	1.4	[-1.8, 3.6]	0.507

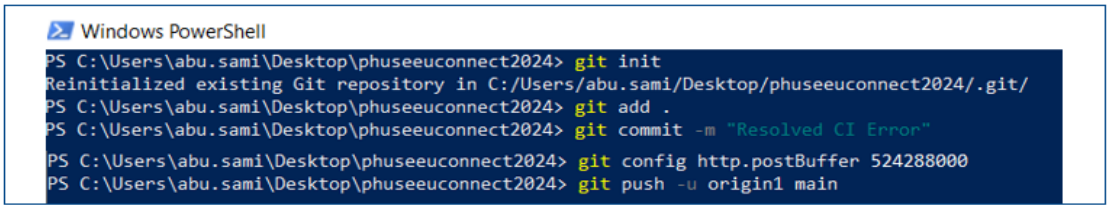
Treatment p-value: 0.800
Covariate p-value: <0.001

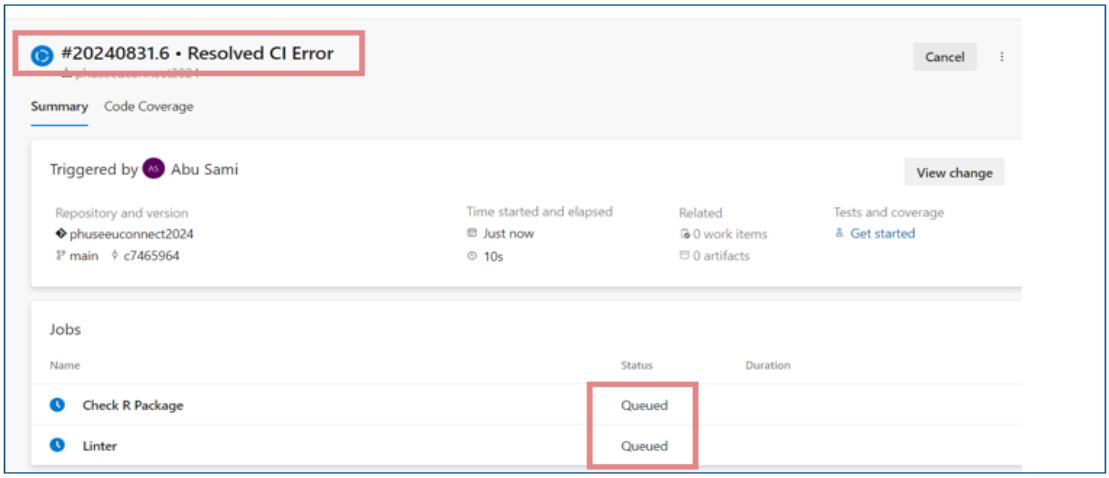
SHINY APPLICATION ON THE CLOUD

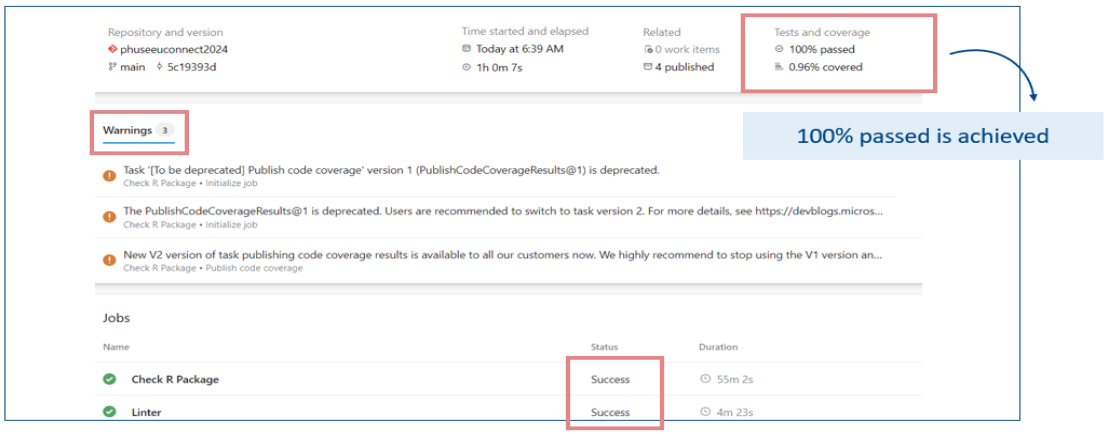
As soon as developers commit any changes to the code and push it to Azure Devops, it automatically triggers the continuous integration process; validates the application package, runs unit tests, and notifies developers by e-mail about the log status of the latest commit, communicating on whether the application passed or failed the process. Nevertheless, all the previous commits are also tracked and displayed on the Azure DevOps dashboard. Below is displayed an example of results of this process.



As an error was issued during the execution of the CI process, the Azure Pipeline does not proceed to further integration steps and, therefore, the CD pipeline can't be triggered on the associated in-progress release. The developers, however, need to rectify the issues and push another change after the desired updates. That commit would be accomplished through Git, and Azure DevOps immediately tracks these latest changes and starts the process execution as shown below.







CONCLUSION

The combination of automation and quality control/assurance is essential for clinical trial analysis, although the pharmaceutical industry has been practicing conservative approaches for decades when analyzing clinical data. In this paper we have demonstrated the development process of an R Shiny application with minimum requirements of manual intervention from development to deployment by using technological advancement. Nevertheless, there are still opportunities for improvement by increasing test coverage percentages through adding more unit tests and excluding the Shiny logic from the coverage calculations. Furthermore, statistical reviewing of modules should be performed as needed when a high level of reliability is expected.

REFERENCES

Posit's package development cheatsheet
Azure Devops Documentation
Introduction to Shiny Modules
Reproducible Environment
Testthat package
R CMD Check
Lintr package
OpenAI's ChatGPT

ACKNOWLEDGMENTS

We would like to acknowledge the contribution of our colleagues Rozeta Simonovska and Katrin Besch, who provided useful insight and support during the preparation of this paper.

CONTACT INFORMATION

Comments and questions are valued and encouraged. Contact the author at:

Author Name: Abu Zafar Mohammad Sami
Company: mainanalytics GmbH
Address: Otto-Volger-Str. 3c, 65843 Sulzbach/Taunus, Germany
Work Phone: +49 6196 766 840
Email: abu.sami@mainanalytics.de
Website: https://mainanalytics.de

Author Name: Yann Féat
Company: mainanalytics GmbH
Address: Otto-Volger-Str. 3c, 65843 Sulzbach/Taunus, Germany
Work Phone: +49 6196 766 840
Email: yann.feat@mainanalytics.de
Website: https://mainanalytics.de