PharmaSUG 2025 - Paper DV-219

Zero to Hero: The Making of a Comprehensive R Shiny Figures App

Yi Guo, Matthew Salzano, Nicholas Sun and Sean Healey, Pfizer Inc.

ABSTRACT

Creation of an interactive R Shiny application from scratch may seem like assembling a complex puzzle, but with the right approach, it becomes an achievable and rewarding journey. This paper will take you through the development process for an enterprise-level application designed for the generation of high-quality figures for clinical research and development, including publications, presentations and regulatory submissions.

Development was done with a unique eye towards integration, as a small step in a larger transformative and integrated strategy of delivering dynamic outputs. The app utilized the R Shiny framework and Golem's modular architecture to deliver a no-code interface for users to create customizable and dynamic visualizations from clinical data. It is centered around three core types of modules: data upload, figure generation, and download. Users can upload and filter their data, select from 15 commonly used plot types, and download the finalized plot along with the corresponding code for reproducibility and traceability.

This paper examines the app development process, including app design and production of individual modules, with a focus on future integration, user experience and statistical analysis. Specifically, we discuss how a detailed understanding of each figure's purpose guided the design of user-friendly widgets. We provide practical examples of relevant R packages and code used to effectively implement various functionalities. By sharing our experience and practical insights, we aim to help others simplify their development process, overcome common challenges, and fully leverage the capabilities of R Shiny to create effective and user-friendly applications.

INTRODUCTION

In the pharmaceutical and clinical research industry, data visualization is essential for effective scientific communication and decision-making. Researchers rely on graphical representations to explore trends, identify patterns, and communicate results in publications, presentations, and regulatory submissions. However, traditional visualization methods, such as those relying on static plotting tools like base R, ggplot2, SAS®, and Python, often require substantial programming expertise and manual adjustments, which can hinder efficiency and reproducibility. These challenges highlight the growing need for an interactive, modular, and user-friendly visualization solution.

This paper presents the development of an R Shiny-based figures application designed to address these gaps. The development process involved gathering cross-functional user requirements, incorporating statistical considerations, and designing an intuitive user interface. The resulting application supports 15 commonly used clinical figures, including eDish, box, Kaplan-Meier (KM), proportional hazard (PH), forest, waterfall, swimmer, spider, scatter, bar-scatter, line, series, step, spaghetti and Schoenfeld plot.

Built on the R Shiny and Golem framework, the application provides a no-code solution that allows users to upload data, select visualization types, adjust parameters interactively, and export both the generated figure and corresponding R code to ensure reproducibility and traceability.

PRE-DEVELOPMENT PHASE

Before diving into the design steps, some readers might wonder what kind of background knowledge is helpful for building a Shiny app like this. In our experience, it is important to have basic familiarity with R programming, particularly data wrangling using packages such as {dplyr} and creating plots with {ggplot2}. A general understanding of how Shiny works is also essential. These skills can be acquired through company training sessions, external workshops, or self-study using online resources. For this particular application, it was also important to have a base knowledge about the industry as well as the goals and nuances of plot generation by pharmaceutical end users. Our background in study work and collaboration with prospective users helped provide this perspective.

This app was developed using the golem framework, which provides a structured approach to building Shiny applications as R packages. Although it is not strictly required to follow this paper, some knowledge of R package development can be beneficial, especially when working on more complex applications or collaborating with others. In our experience, it significantly improves code organization and supports better scalability and maintainability.

Step 1: Defining the scope and initial design considerations

In the early stages of design, we first did preliminary research into the most common figure types and their various customizations used within the company from submission work to ad hoc projects. We identified 15 commonly used figures to cover all but the most specialized use cases and discussed the scope we aimed to cover. After further deliberation, we determined that our coverage should fall within the range of 80%-90% of typical use cases for each figure type. This range was selected based on our experience building tools for real-world programming scenarios, with the goal of supporting the most practical needs while keeping the app efficient and user-friendly. In actual implementation, the final feature coverage for each module may fall within 70% to 90%, depending on iterative feedback collected during internal testing and user requirement discussions. Generally, starting with broader coverage during the design phase helps us focus on more detailed design aspects and statistical requirements.

During the user requirement collection phase, we can verify whether these features are commonly used. If not, we can exclude them from further development.

Step 2: Identifying internal figure templates and common figure types

In alignment with the defined scope, we conducted a review of commonly used figure templates (figure shells) within the company. Our analysis provided a clearer understanding of the standard figure types frequently utilized, informing our design decisions for the tool. To ensure comprehensive coverage, we also consulted with representatives from various departments and therapeutic areas, including experienced statisticians and statistical programmers, involved in figure standardization. Note that figure layouts can differ across organizations, and such variations are not addressed in this paper.

Step 3: Consultation and gathering statistical and design insights

After reviewing numerous figure shells, key users were identified from various departments, primarily statisticians and programmers, to gather feedback. After obtaining feedback from users, several statistical and design questions were summarized, along with expectations to cover majority of common use cases.

Most statisticians and programmers primarily use SAS with Graph Template Language (GTL) to write macros and generate figures. While SAS GTL figures closely resemble figures produced in R using {ggplot2}, the feedback we received noted some layout differences, such as the positioning of legends. Although the differences are minor, the feedback was invaluable to our application design.

For example, when designing a forest plot, we asked statisticians whether it was frequently used in integrated or meta-analyses. If it was not, consider simplifying the plot by removing elements such as heterogeneity results (τ^2, χ^2, df, I^2). Since forest plots primarily display ratios (e.g., hazard ratio, odds ratio, relative risk) and differences (e.g., mean difference, risk difference, standardized mean difference), the reference line is set to 1 for ratios and 0 for differences. Providing these two options is sufficient for most cases.

Similarly, when designing the eDish plot module, data per sample, per subgroup and per quadrant displays were identified to confirm if necessary. If the likelihood of using three or more reference lines (including both horizontal and vertical) was less than 20%, limiting the design to two lines, one horizontal and one vertical reference line is optimal. Additionally, consultation with the Drug Safety department to confirm whether displaying subject IDs alone in the Hy’s law cases quadrant was sufficient, and that no additional demographic information, such as age or sex, was required.

We also met with statistical programmers responsible for figure standardization to review and verify technical details so that plots comply with Pfizer submission requirements. For example, they were able to provide a default Pfizer-specific theme for plots.

Step 4: Discussion on user experience and widget layout design

With the feedback gathered from users and our understanding of the statistical requirements for each figure, we began targeted discussions on how to design each widget to be both user-friendly and capable of meeting all statistical needs. The layout design for the R Shiny user interface followed a general rule: within each module, the figure title and subtitle always appear at the top, followed by the widget controls for axes (such as x and y axis labels, scales, etc.). The final section of the layout is reserved for the figure download options, ensuring a logical flow and ease of use for the end-user.

CROSS-TEAM FEEDBACK AND REFINEMENT PHASE

Following the pre-development phase, we developed an initial functional prototype of the Figure App over a period of 12 weeks. We then entered an iterative refinement phase, conducting demo sessions with users from various departments and teams to present the app's design principles, individual modules, and their corresponding functionalities, demonstrating its core features and how to use the app. The goal was to gather further feedback on usability, feature completeness, and alignment with user expectations.

During these meetings, we collected feedback on figure layout, customization options, and user experience. For instance, statisticians provided insights into the necessity of certain statistical elements, while programmers assessed the feasibility of integrating the tool into their workflows.

Based on the feedback received, we identified areas requiring further improvement. In addition to adding or refining widgets in some figures, several key suggestions were made:

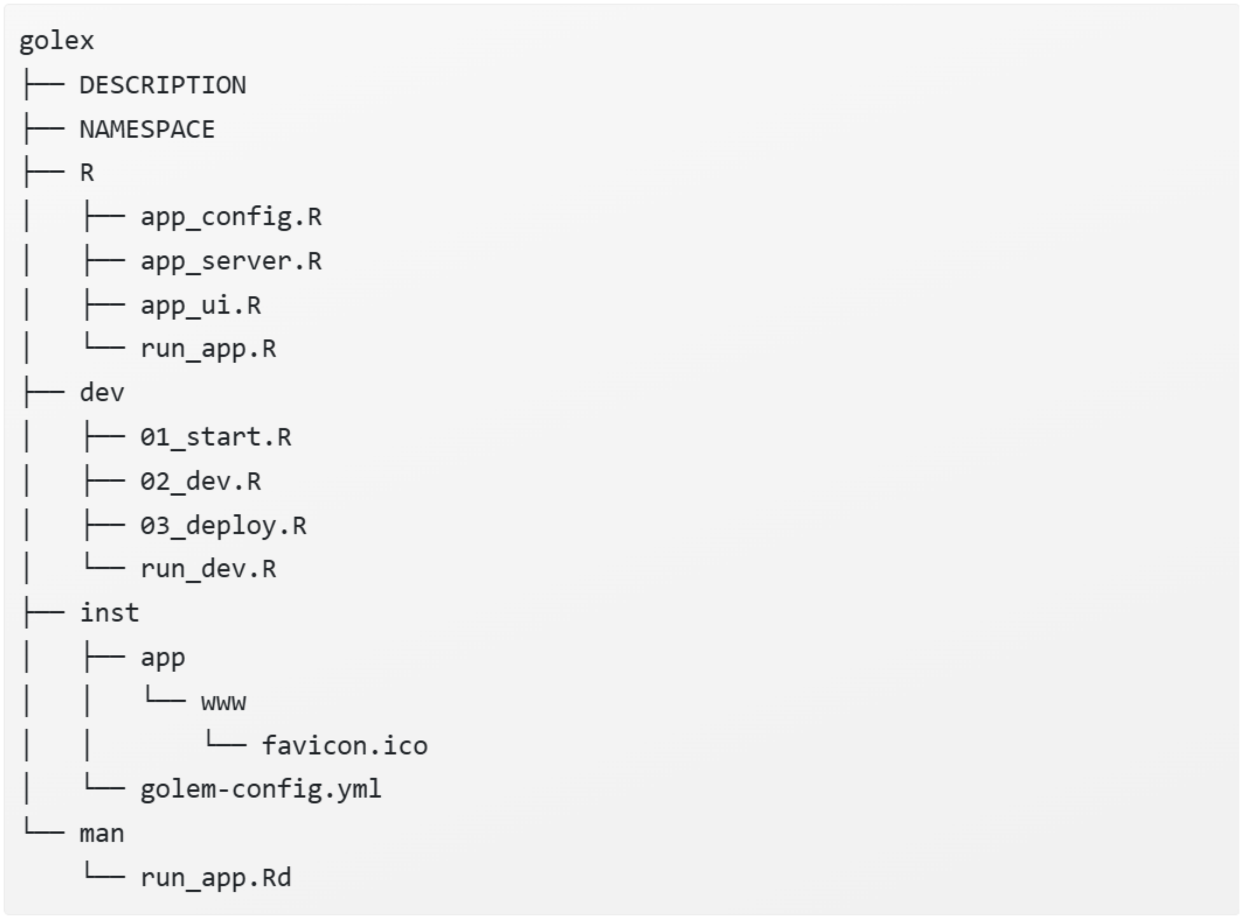
1. Adding a dedicated section to display the `ggplot()` function so that changes in parameters, figure styles, and other settings would be reflected in real time.
2. Introducing a code export feature, allowing users to download the corresponding ggplot code. This enables users to first generate a basic figure within the R Shiny app and then export the code for further customization in R, providing greater flexibility for specific figure requirements.
3. Implementing a "Reset to Defaults" button, which resets all parameters, style options, and file names to their original settings. This feature significantly improves efficiency by allowing users to quickly start over when creating figures.

After each round of feedback, we iteratively updated the tool to ensure continuous improvement and better alignment with user needs. This collaborative approach allowed us to refine the app into a more robust and user-friendly solution before finalizing development.

APP STRUCTURE AND DEVELOPMENT APPROACH

FRAMEWORK

Early in the planning stages, we determined we needed a framework that would be easy to use and allow for flexibility in the app. Flexibility in the app allows for enhanced customization of elements (i.e., plots) to achieve our aim of covering most use cases. We also needed a framework that be easy to organize and maintain the backend code. Therefore, we leveraged the {golem} package as the backbone of our application and {tidyverse} for its data processing and visualization engine. We settled on the {golem} package as the base of the R Shiny application itself. The {golem} framework lowers the bar for creating complex applications, especially R users who may not have the necessary knowledge or experience in software development processes. The {golem} package comes with files and functions that simplify some back=end processes (e.g., module creation for R Shiny, app deployment, etc.) and provide a skeleton for users to start building their app. Essentially, {golem} structures the application as a package to keep all production code, development code, application information/configuration, and other necessary files organized and bundled together.



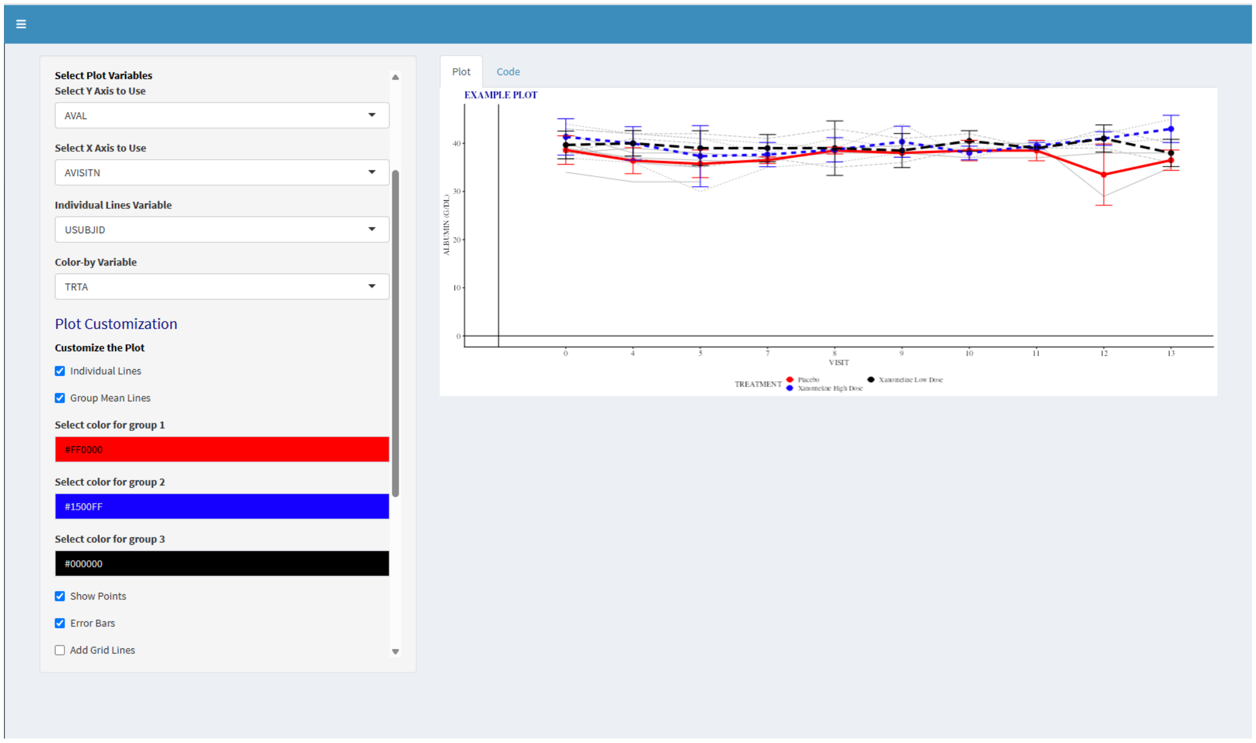
```
golex
├── DESCRIPTION
├── NAMESPACE
├── R
│   ├── app_config.R
│   ├── app_server.R
│   ├── app_ui.R
│   └── run_app.R
├── dev
│   ├── 01_start.R
│   ├── 02_dev.R
│   ├── 03_deploy.R
│   └── run_dev.R
├── inst
│   ├── app
│   │   └── www
│   │       └── favicon.ico
│   └── golem-config.yml
└── man
    └── run_app.Rd
```

Figure 1. Example schematic of application built using {golem} (Fay et al. 2021).

The {tidyverse} collection contains a multitude of packages that are helpful for data science and visualization. Within the {tidyverse} is {ggplot2}, a popular package amongst R users that is well-known for its ability to produce stunning visualizations within its highly customizable framework. Since the focus of our app is on plotting data, {ggplot2} and its accompanying packages became essential to our development process.

We were able to add the desired customization to our application by taking advantage of {ggplot2}'s "layering" technique for plot creation. Typically, plots made using {ggplot2} are created in a single step – with all (or most) layers defined at the same time and chained together with a plus-sign. Although each layer is customizable, this technique requires that all layers be present in the plot. However, users often want the option to overlay points on a bar plot or add a regression line to a scatter plot – each of which requires the addition or removal of a layer. Repeating the plot code under a conditional is viable for one or two options (e.g., overlaid points or a regression line), but the code will become unwieldy as the number of options increases. For example, in our line plot, users can select any combination of individual lines, group lines, overlaid points, and overlaid points with error bars to display on the plot.

Rebuilding each layer of the plot for each condition would result in repetitive and inefficient code.



Select Plot Variables

Select Y Axis to Use: AVAL

Select X Axis to Use: AVISITN

Individual Lines Variable: USUBJID

Color-by Variable: TRTA

Plot Customization

Customize the Plot

- [x] Individual Lines
- [x] Group Mean Lines

Select color for group 1: #FF0000

Select color for group 2: #1500FF

Select color for group 3: #000000

- [x] Show Points
- [x] Error Bars
- [ ] Add Grid Lines

Figure 2. Line plot customized with group mean lines overlaid with points and errors, and subject lines in the background. If selected with group mean lines, subject lines will automatically turn gray while maintaining line type.

Therefore, we decided to first define each layer as a list within a reactive object and then chain the ‘reactive’ layers together in a section that builds the plot. This approach streamlined our code so that we needed fewer and less complex conditional statements. We were also able to eliminate repetitive code to aid in future maintenance of the app.

{ggplot2}’s popularity has resulted in multitude of accompanying packages to aid in creating plots. Some plots have packages dedicated to their creation, with the tedious work to integrate with {ggplot2} completed by the package authors. While some plots have customization built into their functionality, their integration with {ggplot2} allows programmers to use the package to build a base plot in the layering technique mentioned above.

APP DESIGN

GUI design, or graphical user interface design, is an important part of building an app since it drives the user experience. A visually unappealing GUI or one that is hard to navigate may steer users away from an app despite its functionality. Therefore, we wanted to stick with a simple, elegant, yet functional design.

We settled on a basic design for the GUI – a main display panel with two sidebars on the right. The first sidebar allows for navigation around the different plot modules in the app. When a plot module is selected, a second side bar appears beside the main display. The second side bar contains the customization options for each plot. The sidebar size is fixed at 80% of the main panel but is scrollable so that users can access all customizations. The main display panel contains two tabs - the first tab displays the plot, while the second tab displays the core logic behind the plot.

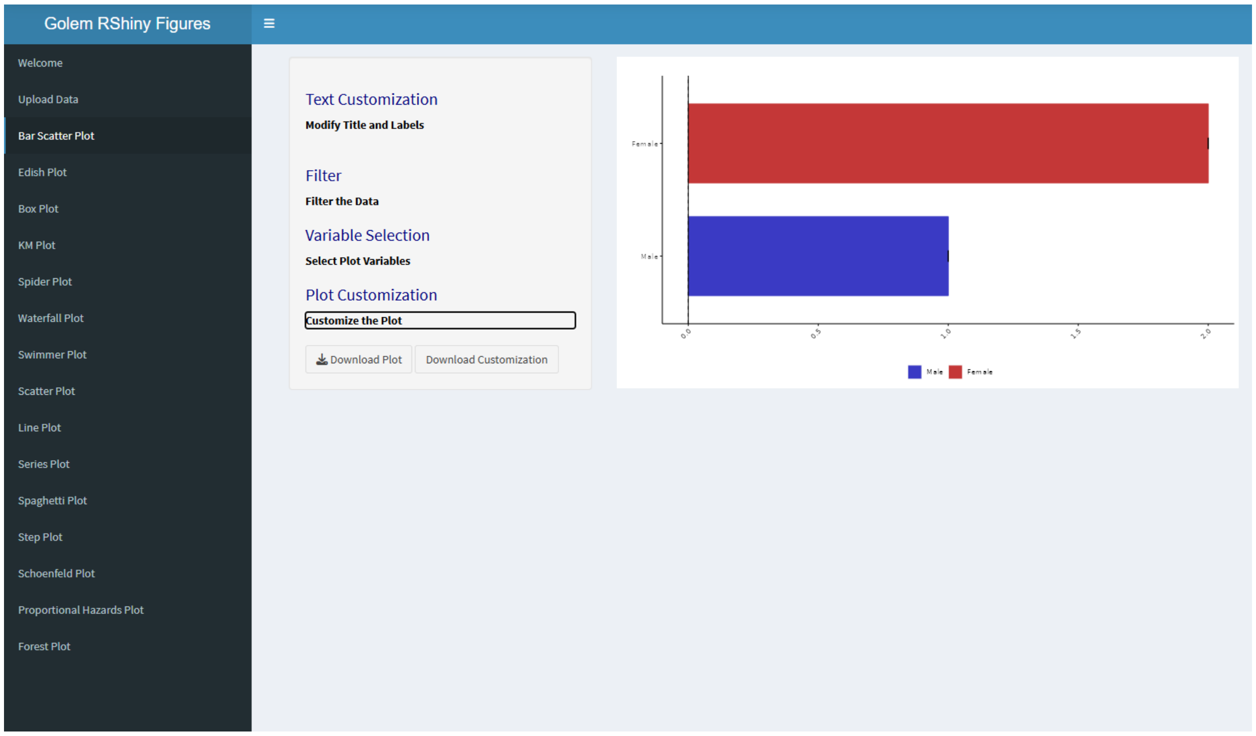


Figure 3. Application layout showing main sidebar on far left in black, the module sidebar on the middle-left, and the plot display on the right.

In the customization sidebar, plot features are divided into five sections common across all plots (except for proportional hazards and Schoenfeld): text customization, filter, variable selection, plot customization, and download customization. Users can set the title, footer, and various labels in the Text Customization section. While we encourage users to do their filtering in the data input module, we do provide a simple filtering option in each plot module. Here, users can only select one variable to filter, but the app recognizes if the variable is numeric or character format and adjusts the filter display accordingly. Users will define the necessary variables for the plot in the Variable Selection section and will customize the plot with the features in the Plot Customization section. A summary of the variables and features available to each plot are summarized in the table below.

Table 1. Summary of Customizations Available for Each Plot Module.

Figure Type	Key Features
eDish	x-axis variable, y-axis variable, grouping variable, point customization (shape, size, color), reference lines, axis transformation (e.g., log scale), axis ranges, display subject IDs, grid lines, legend position, text customization (title, axis labels, footnote, quadrant labels), reset inputs
Box	x-axis variable, y-axis variable, grouping variable, box customization (color, width, spacing), y-axis log scale option, show means, show all points, grid lines, legend position, text customization (title, axis labels, footnote), reset inputs
Kaplan-Meier	time variable, event variable, grouping variable (strata), filter-by-category, line color, show p-value, axis break, confidence interval, risk table, survival median line, grid lines, legend position, text customization (title, axis labels, footnote), reset inputs
Proportional Hazards	model parameters, point customization (shape, color), text customization (title, axis labels, footnote), reset inputs
Forest	variable selector, reference line, axis scale, error bar width, text customization (title, footnote), figure dimensions (width, height and DPI), reset inputs
Waterfall	x-axis variable, y-axis variable, grouping variable, filter-by-category, show x-axis tick labels, bar color, threshold lines, legend position, text customization (title, axis labels, footnote), reset inputs
Swimmer	ID variable, treatment start/end, event variable, grouping variable, response customization (symbols, colors), bar customization (color, width), custom time axis, legend position, text customization (title, axis labels, footnote), reset inputs
Spider	ID variable, time variable, response variable, grouping variable, filter-by-category, line customization (color, style), point customization (color, shape), reference lines, grid lines, legend position, text customization (title, axis labels, footnote), reset inputs
Scatter	x-axis variable, y-axis variable, grouping variable, point customization (shape, size), trend line, grid lines, legend position, text customization (title, axis labels, footnote), reset inputs

Bar-scatter	x-axis variable, y-axis variable, grouping variable, plot-by variable, bar customization (color, fill/no fill), show points, horizontal bars, error statistic, reference line (default = 0 = x axis), grid lines, legend position, text customization (title, axis labels, footnote), reset inputs
Line/Series/Step/Spaghetti	ID variable, grouping variable, show individual lines, show group lines, show points, show error bars, grid lines, point size, legend position, text customization (title, axis labels, footnote), reset inputs
Schoenfeld	model parameters, point customization (shape, color), text customization (title, axis labels, footnote), reset inputs

KEY FEATURES AND FUNCTIONALITIES

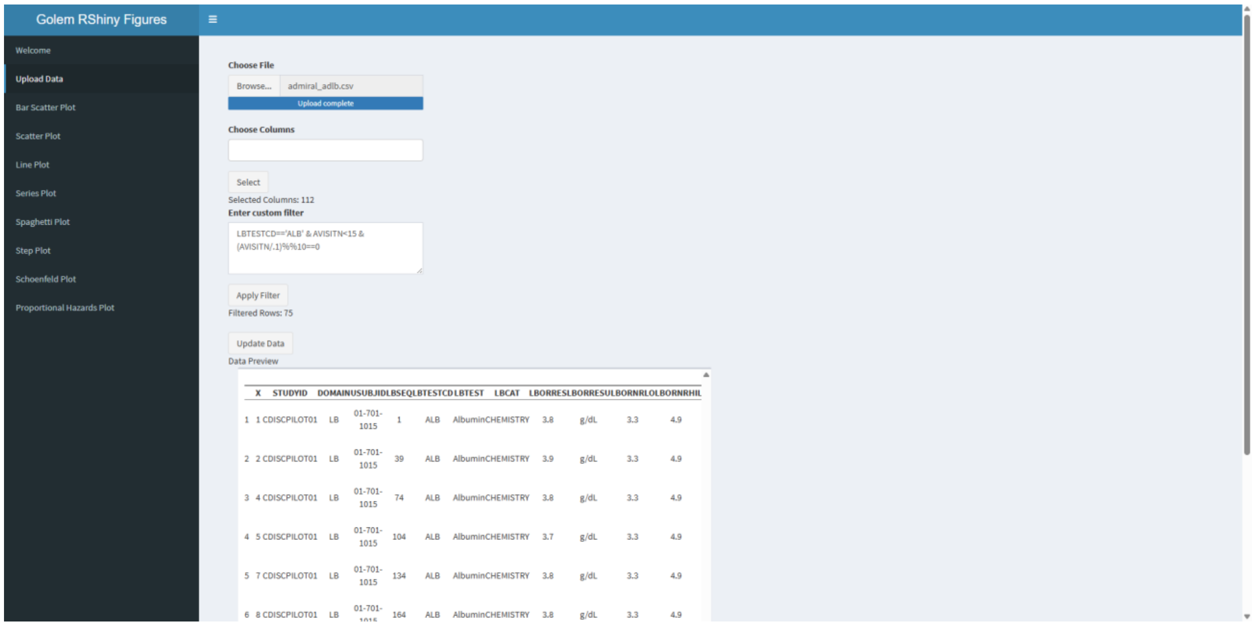
DATA INPUT MODULE

Building on the flexibility mentioned in the previous section, we wanted to allow our users 1. the ability to import multiple types of datasets and 2. have more freedom over their data once imported. We considered adding the data import functionality to every plot module, but we wanted to separate the data import process from the plot generation. Therefore, we created a data import module that allows users to import and view their data, as well as filter the data (by column, row, or both) if desired.

Although the SAS7BDAT format is common for clinical trial data, some users may be working with data in other formats, such as TXT, CSV, XPT, etc. This was easily addressed by the {tidyverse} suite of packages. {haven} contains the function necessary for importing SAS datasets (`read_sas()`), but it also contains functionality to import data from other statistical programs. For more ubiquitous data types, {readxl} and {readr} were used to import Excel files and other delimited files, respectively. The `fileInput()` function, which lives in the UI section of the app, contains an argument that allows the programmer to set which data types are allowed (e.g., `accept = c('.sas7bdat', '.csv', '.xlsx')`). A conditional statement was created on the server side of the application to select the correct import function based on the file path selection in the `fileInput()` controller. Future updates may require compatibility with other data types. Luckily, {tidyverse} also contains packages to import other file types (e.g., JSON, XML, etc.), which should make any future integration quite easy.

We built our application with an aim towards submissions, thus users would upload a validated dataset. Nonetheless, we wanted to allow users a little flexibility regarding their data for special cases, like an exploratory data analysis. This required striking a balance to provide limited control over the data. Users may select certain columns or filter rows to meet their criteria, but they are not able to manipulate data (i.e., create new variables). However, should the manipulation functionality be requested by users in the future, the structure of the code should allow for seamless integration. The final dataset used for plotting is built from a series of conditional statements that check for any row or column filtering inputs and then apply any that exist to the original dataset using the {tidyverse} `filter()` and `select()` functions. Therefore, if needed for future purposes, another layer could be added to the set of conditional statements to include an input for column creation.

Column selection was simple – column names from the data are passed into the `selectInput()` function with the argument `multiple = TRUE`. The filtering functionality proved more complicated due to some philosophical issues. The goal of our application was a no-code solution to TLF generation for users, and we already had a working filtering technique in the plot modules. However, in certain complex scenarios, such as when multiple conditions are involved or when a continuous variable is used instead of a categorical one for filtering, user-defined inputs are required, like the one seen in Figure 4. Therefore, we decided to abandon our philosophy in this instance and added a free-text box to the data module. The free-text box allows users to write a custom filtering scheme using R syntax. We embedded the custom filter code in a `tryCatch()` to save the application from crashing if an invalid filtering scheme was entered due to a typo or incorrect code syntax. In its current state, the custom filtering option is only suggested for exploratory analyses.



Choose File

admiral_adlb.csv

Choose Columns

Selected Columns: 112

Enter custom filter

```
LBTESTCD=='ALB' & AVISITN<15 &
(AVISITN/.1)%%10==0
```

Filtered Rows: 75

Data Preview

	X	STUDYID	DOMAIN	USUBJID	LBSEQ	LBTESTCD	LBTEST	LBCAT	LBORRES	LBORRESU	LBORNRLO	LBORNRHI
1	1	CDISCPILOT01	LB	01-701-1015	1	ALB	Albumin	CHEMISTRY	3.8	g/dL	3.3	4.9
2	2	CDISCPILOT01	LB	01-701-1015	39	ALB	Albumin	CHEMISTRY	3.9	g/dL	3.3	4.9
3	4	CDISCPILOT01	LB	01-701-1015	74	ALB	Albumin	CHEMISTRY	3.8	g/dL	3.3	4.9
4	5	CDISCPILOT01	LB	01-701-1015	104	ALB	Albumin	CHEMISTRY	3.7	g/dL	3.3	4.9
5	7	CDISCPILOT01	LB	01-701-1015	134	ALB	Albumin	CHEMISTRY	3.8	g/dL	3.3	4.9
6	8	CDISCPILOT01	LB	01-701-	164	ALB	Albumin	CHEMISTRY	3.8	g/dL	3.3	4.9

Figure 4. Data upload module. In this example, the admiral_adlb.csv (Straub et al. 2023) dataset is loaded into the data module. The custom filter only maintains Albumin lab test data that excludes unplanned (visit codes ending in .1) and follow-up visits (visit codes > 15).

We also wanted to display the final data for our users, so that they could use it for review or exploratory purposes. We added an action button under the column and row selection features that display how many columns and rows are selected. These action buttons are not necessary for the final dataset creation but rather exist as an option to the user. The final dataset is created when users press the “Update Data” action button, which applies the necessary subset procedures (described above). This action must be performed upon initial data creation, even if the subset inputs are empty, and any time updates are made to either subset feature. Besides creating the final dataset, the “Update Data” button also performs a second action by triggering the app to display the data in a HTML tabular format.

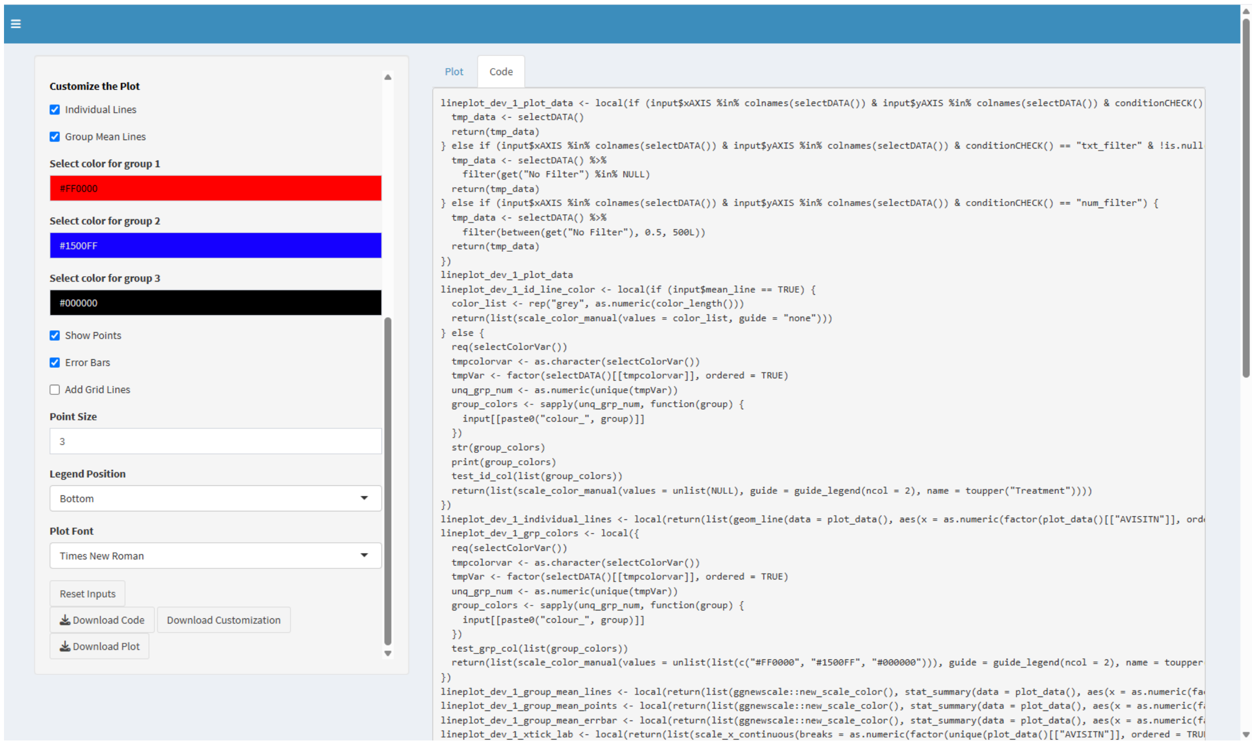
EXPORT OPTION

The option to export figures was necessary to our app, otherwise it would be useless to users. An export feature also provided an opportunity to take advantage of the {golem} framework. The export feature needed to be on every plot module, but it would be highly inefficient to copy the code to 15 different modules (and any future plot modules). Therefore, we created a module specifically for exporting plots, which is called by every plot module.

The module is simplistic in its design. It contains an action button that toggles whether to show or hide the export options. Users then input their file name, select the file extension from a drop-down menu, and then click the “Download Plot” button to export the file to the user’s local drive. In the drop-down menu, we supplied multiple format types for the users to choose from: PDF, PNG, JPEG, and TIFF – although other compatible formats can be easily added upon request.

REPRODUCIBILITY

It was important for us to add functionality for reproducibility so that users could use our app for submissions. Here, we relied on the framework provided by the {shinymeta} package, which was built for purposes of reproducing RShiny code for distribution. This method requires using the {shinymeta} versions of reactive objects. The `expandChain()` function within the package can then read these reactive objects to reproduce the backend code as a text output. Furthermore, variable values can replace variable names in the resolved code. For example, in the screenshot below, the axis and grouping variable values are displayed in the code rather than variable names. The resolved code can then be saved using the “Download Code” button, which saves the code as a RDS or R file. We also gave users the option to view the resolved code in real-time by adding a “Code” tab to the display area of the GUI.



Customize the Plot

- [✓] Individual Lines
- [✓] Group Mean Lines

Select color for group 1: #FF0000

Select color for group 2: #1500FF

Select color for group 3: #000000

- [✓] Show Points
- [✓] Error Bars
- [ ] Add Grid Lines

Point Size: 3

Legend Position: Bottom

Plot Font: Times New Roman

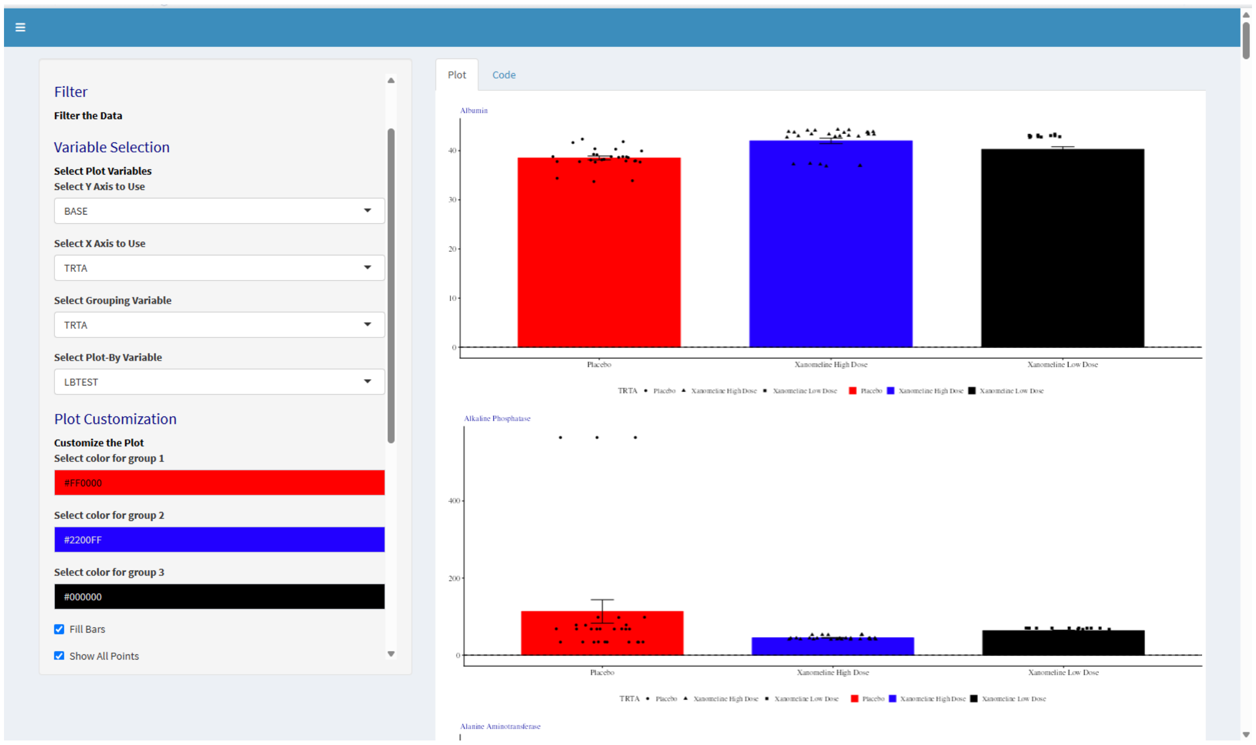
```r
lineplot_dev_1_plot_data <- local(if (input$xAXIS %in% colnames(selectDATA()) & input$yAXIS %in% colnames(selectDATA()) & conditionCHECK()
  tmp_data <- selectDATA()
  return(tmp_data)
} else if (input$xAXIS %in% colnames(selectDATA()) & input$yAXIS %in% colnames(selectDATA()) & conditionCHECK() == "txt_filter" & !is.null
  tmp_data <- selectDATA() %>%
    filter(get("No Filter") %in% NULL)
  return(tmp_data)
} else if (input$xAXIS %in% colnames(selectDATA()) & input$yAXIS %in% colnames(selectDATA()) & conditionCHECK() == "num_filter") {
  tmp_data <- selectDATA() %>%
    filter(between(get("No Filter"), 0.5, 500L))
  return(tmp_data)
})
lineplot_dev_1_plot_data
lineplot_dev_1_id_line_color <- local(if (input$mean_line == TRUE) {
  color_list <- rep("grey", as.numeric(color_length()))
  return(list(scale_color_manual(values = color_list, guide = "none")))
} else {
  req(selectColorVar())
  tmpcolorvar <- as.character(selectColorVar())
  tmpVar <- factor(selectDATA()[[tmpcolorvar]], ordered = TRUE)
  unq_grp_num <- as.numeric(unique(tmpVar))
  group_colors <- sapply(unq_grp_num, function(group) {
    input[[paste0("colour_", group)]]
  })
  str(group_colors)
  print(group_colors)
  test_id_col(list(group_colors))
  return(list(scale_color_manual(values = unlist(NULL), guide = guide_legend(ncol = 2), name = toupper("Treatment"))))
})
lineplot_dev_1_individual_lines <- local(return(list(geom_line(data = plot_data(), aes(x = as.numeric(factor(plot_data()[["AVISITN"]], ord
lineplot_dev_1_grp_colors <- local({
  req(selectColorVar())
  tmpcolorvar <- as.character(selectColorVar())
  tmpVar <- factor(selectDATA()[[tmpcolorvar]], ordered = TRUE)
  unq_grp_num <- as.numeric(unique(tmpVar))
  group_colors <- sapply(unq_grp_num, function(group) {
    input[[paste0("colour_", group)]]
  })
  test_grp_col(list(group_colors))
  return(list(scale_color_manual(values = unlist(list(c("#FF0000", "#1500FF", "#000000"))), guide = guide_legend(ncol = 2), name = toupper
})
lineplot_dev_1_group_mean_lines <- local(return(list(ggnewscale::new_scale_color(), stat_summary(data = plot_data(), aes(x = as.numeric(fa
lineplot_dev_1_group_mean_points <- local(return(list(ggnewscale::new_scale_color(), stat_summary(data = plot_data(), aes(x = as.numeric(f
lineplot_dev_1_group_mean_errbar <- local(return(list(ggnewscale::new_scale_color(), stat_summary(data = plot_data(), aes(x = as.numeric(f
lineplot_dev_1_xtick_lab <- local(return(list(scale_x_continuous(breaks = as.numeric(factor(unique(plot_data()[["AVISITN"]], ordered = TRU
```

Figure 5. Core logic to create line plot in Figure 2 as captured by {shinymeta}. Code can be downloaded using the 'Download Code' button in the customization sidebar.

ENHANCING USABILITY WITH INTERACTIVE COMPONENTS

MULTI-PLOT VIEW

Circumstances may dictate that users create the same plot for different groups of participants–e.g., a bar chart for males versus females with a separate plot for each treatment arm. {ggplot2} has built-in options to create a matrix of plots, but two issues arise with this method. The plot matrix grows with the number of levels in the “plot-by” variable. With a constrained display size, such as the GUI window or a PDF document, a large matrix will shrink each plot to the point that it is illegible. Also, users may want to save each plot as its own picture file or on its own page in a PDF, which is not possible with the facet options provided by {ggplot2}.


Filter

Filter the Data

Variable Selection

Select Plot Variables

Select Y Axis to Use: BASE

Select X Axis to Use: TRTA

Select Grouping Variable: TRTA

Select Plot-By Variable: LBTEST

Plot Customization

Customize the Plot

Select color for group 1: #FF0000

Select color for group 2: #2200FF

Select color for group 3: #000000

☑ Fill Bars

☑ Show All Points

Plot
Code
Albumin
40
30
20
10
0
Placebo
Xanomeline High Dose
Xanomeline Low Dose
TRTA ● Placebo ▲ Xanomeline High Dose ■ Xanomeline Low Dose 🟥 Placebo 🟦 Xanomeline High Dose ⬛ Xanomeline Low Dose
Alkaline Phosphatase
400
200
0
Placebo
Xanomeline High Dose
Xanomeline Low Dose
TRTA ● Placebo ▲ Xanomeline High Dose ■ Xanomeline Low Dose 🟥 Placebo 🟦 Xanomeline High Dose ⬛ Xanomeline Low Dose
Alanine Aminotransferase

Figure 6. Example of a bar-scatter plot in multi-plot view.

We were able to solve the plot display and plot save issues by saving every plot into a reactive list. If a user selects a “plot-by" variable, the code will loop through each level of the variable to subset the data, generate the plot, and save the plot into the list. If no “plot-by” variable is chosen, the single plot generated will be saved into a list of length one. Users are no longer able to have plots in a grid form. Rather, using the `map()` function in conjunction with `createUI()` displays each plot in the list (regardless of length) in its full size (see Code Snippet 2 in Appendix). If the number of plots exceeds the size of the GUI screen, {shiny} automatically creates a scrollbar.

INTERACTIVE CUSTOMIZATION OF COLOR SCHEMES

Feedback from the pre-development phase indicated that statisticians and programmers prefer customizable colors over predefined palettes when creating figures. To meet this need, we integrated the {colourpicker} package into our application. It is an interactive color selection tool that enables users to dynamically customize colors, adjust alpha transparency, and choose from predefined or custom palettes—all without manually entering color codes. Compared to standard color palettes, {colourpicker} eliminates the restriction of selecting from a fixed set of colors, providing greater flexibility in visualization customization.

In all our modules, we used color pickers to customize data points, lines, bars, etc. However, for reference lines, we opted for a limited palette rather than allowing fully customizable colors.

The reason is that reference lines are primarily used to indicate key values (e.g., means, thresholds) rather than to draw visual attention. According to best practices in data visualization, reference lines are usually designed with formal, subtle, and easily distinguishable colors. Besides, in fields such as clinical research, reference line colors typically follow standardized conventions to ensure consistent interpretation of graphical outputs. Common conventions include:

a. Black/Gray: Often used for baseline indicators, such as zero lines, mean lines, and median lines.
b. Blue: Commonly used for confidence intervals or other reference indicators.
c. Red: Typically used for threshold lines, such as risk limits or warning markers.

To align with these conventions, we used `palette = "limited"`, an option available in {colourpicker}. When using the `colourInput()` function from {colourpicker}, setting palette = "limited" restricts the available color choices to 40 commonly used colors. These 40 commonly used colors cover essential shades such as black, gray, blue, and red, which are sufficient for reference line selection while maintaining clarity and consistency in visualization.

Implementation in R Shiny

To install and use the {colourpicker} package in R, run the following command:

```r
install.packages("colourpicker")
```

A basic implementation of {colourpicker} in a Shiny app is as follows:

```r
library(shiny)
library(colourpicker)

ui <- fluidPage(
  colourInput("col", "Select a color:", value = "#000000"),
  plotOutput("plot")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    plot(cars, col = input$col, pch = 16, cex = 1.5)
  })
}
```

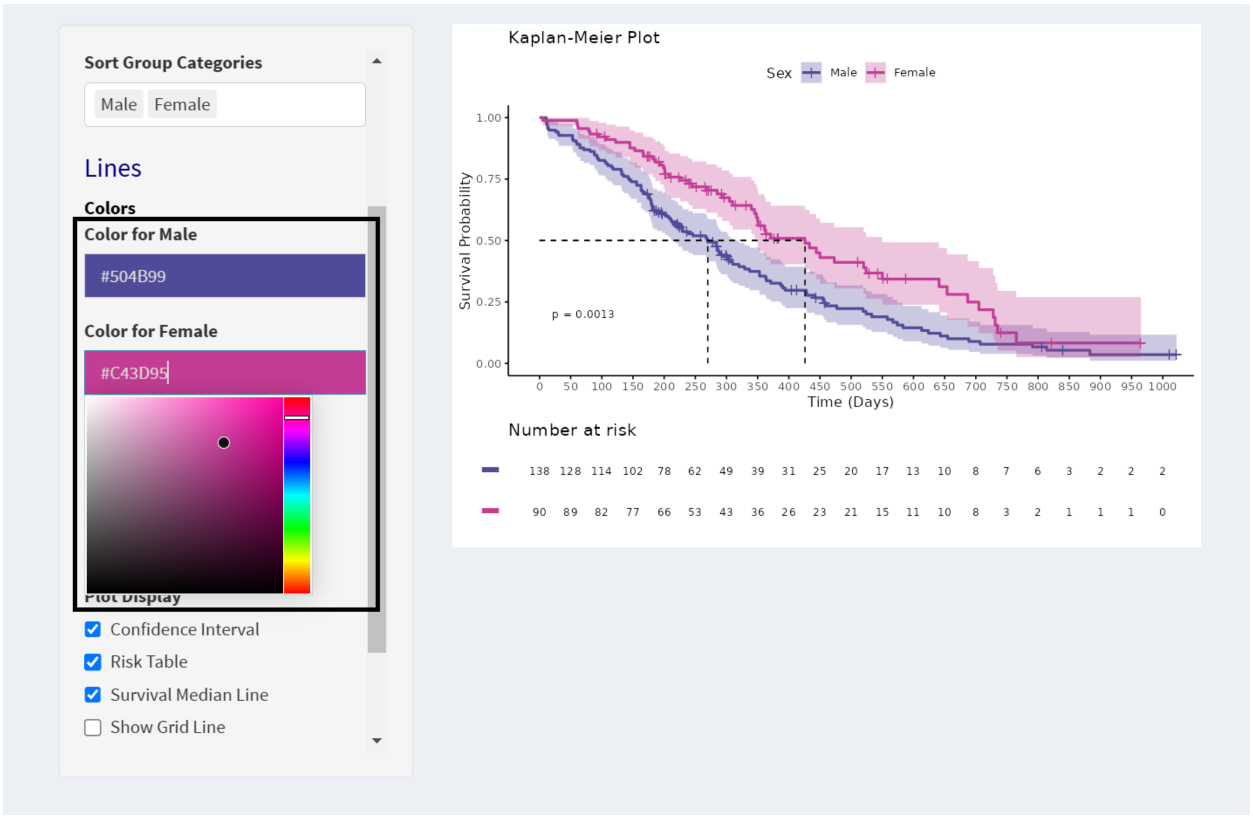
```
shinyApp(ui, server)
```

Examples

The {colourpicker} package has been integrated into every module of our app, allowing users to customize colors for data points and lines. Here we provide several figure examples to illustrate the implementation of {colourpicker} within our app.

1. Kaplan-Meier (KM) Plot

In KM plot module, users can customize the color of each survival curve by group. The example below presents a KM plot stratified by sex, with an independent color picker for each group to customize its color (Figure 7).



Color for Male: #504B99

Color for Female: #C43D95

- [✓] Confidence Interval
- [✓] Risk Table
- [✓] Survival Median Line
- [ ] Show Grid Line

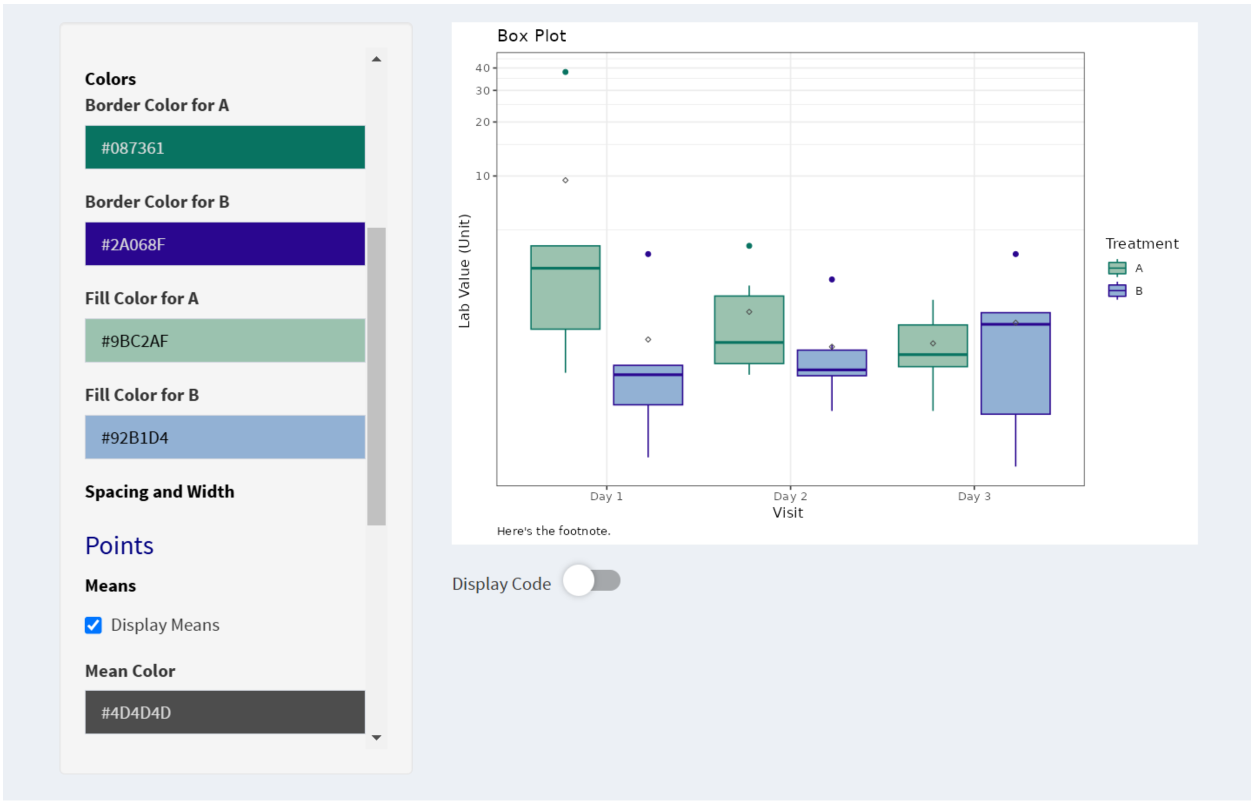
Number at risk

🟦	138	128	114	102	78	62	49	39	31	25	20	17	13	10	8	7	6	3	2	2	2
🟪	90	89	82	77	66	53	43	36	26	23	21	15	11	10	8	3	2	1	1	1	0

Figure 7. Example of plot colors from {colourpicker} to customize the male and female data in a KM plot.

2. Box Plot

In the box plot module, users can adjust the box borders (including median line), fill color, and mean point using the color picker tool. The figure below presents an example of a box plot by visit for laboratory data, incorporating color pickers to enhance customization. Selecting a color dynamically updates the corresponding hex code, providing precise control over the plot's appearance.



Colors

Border Color for A

#087361

Border Color for B

#2A068F

Fill Color for A

#9BC2AF

Fill Color for B

#92B1D4

Spacing and Width

Points

Means

✓ Display Means

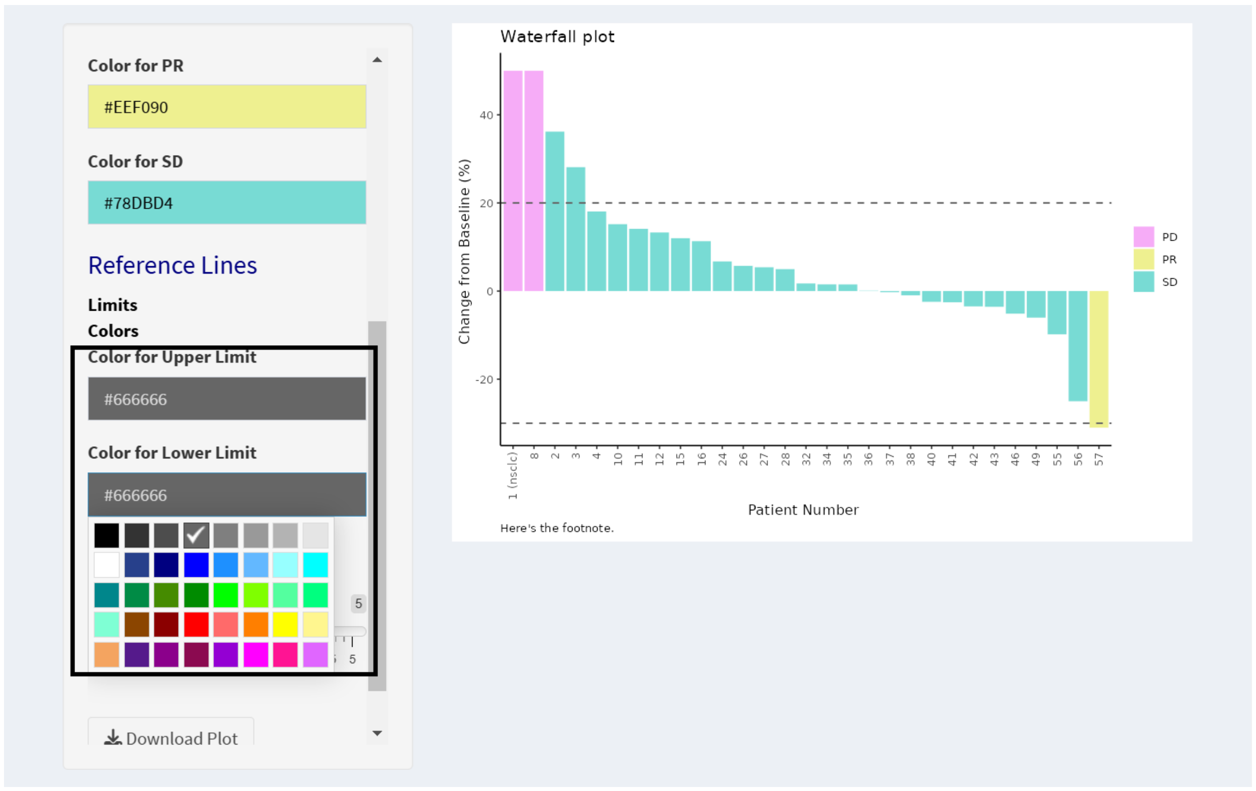
Mean Color

#4D4D4D

Figure 8. Example of a box plot for lab data with box border, box fill, median line, and mean point colors assigned using {colourpicker}.

3. Reference Line

Figure 9 presents a waterfall plot for a specific treatment group. A limited palette is applied independently to two reference lines to enhance clarity and consistency in visualizing key clinical thresholds.



Color for PR

#EEF090

Color for SD

#78DBD4

Reference Lines

Limits

Colors

Color for Upper Limit

#666666

Color for Lower Limit

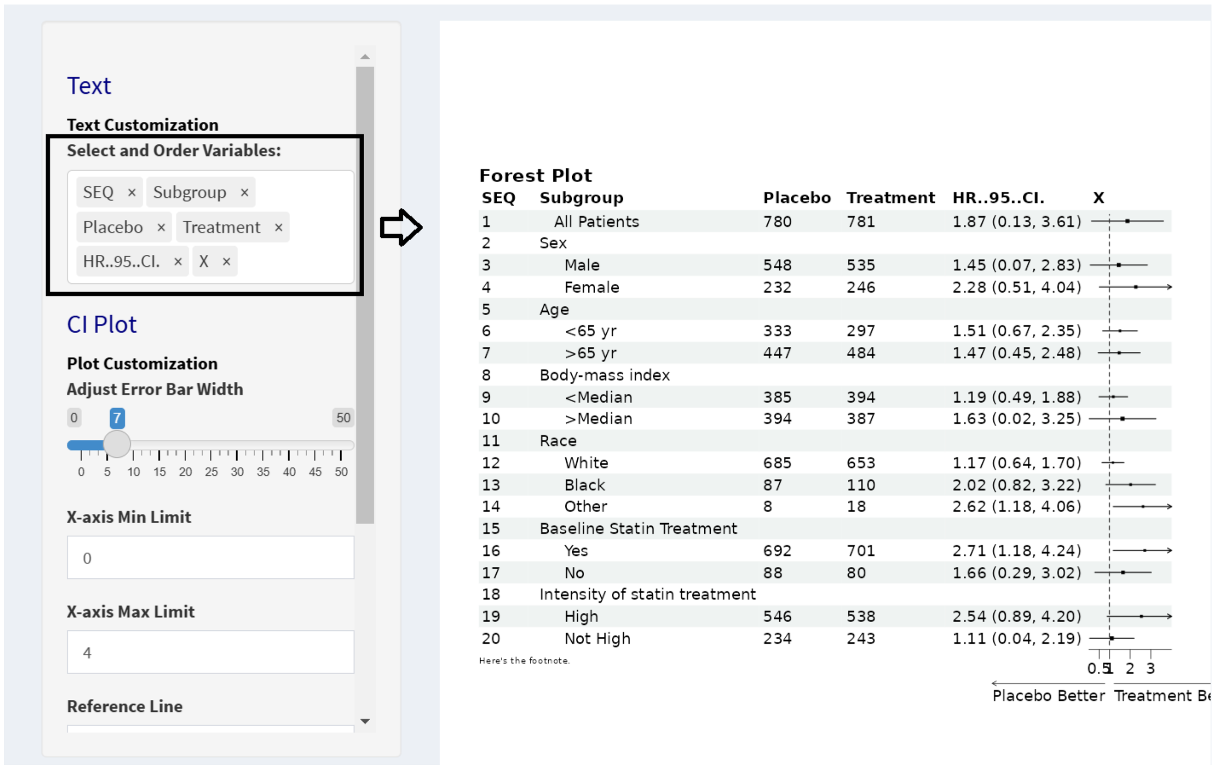
#666666

Figure 9. Example of a waterfall plot with a limited palette for reference lines.

INTERACTIVE VARIABLE SELECTION AND ADJUSTMENT

Forest plots often contain confidence intervals with varying structures, where the placement and number of columns on either side of the plot can differ depending on the dataset and analysis requirements. In some cases, the plot is centered with results displayed on both sides, while in others, it is positioned on the far right with all statistical results on the left. This variability necessitates a flexible and customizable approach to structuring the visualization.

To address this, we integrated an interactive variable selector, allowing users to dynamically select, reorder, and rename variables. This feature enhances flexibility in structuring the forest plot based on specific analytical and reporting needs. Users can select relevant variables, reorder them via drag-and-drop, and rename them within the UI for clarity. The selected variables are immediately reflected in the forest plot, ensuring a streamlined and efficient workflow without requiring modifications to the original dataset.


Text

Text Customization

Select and Order Variables:

SEQ × Subgroup × Placebo × Treatment × HR..95..CI. × X ×

CI Plot

Plot Customization

Adjust Error Bar Width: 7 (0–50)

X-axis Min Limit: 0

X-axis Max Limit: 4

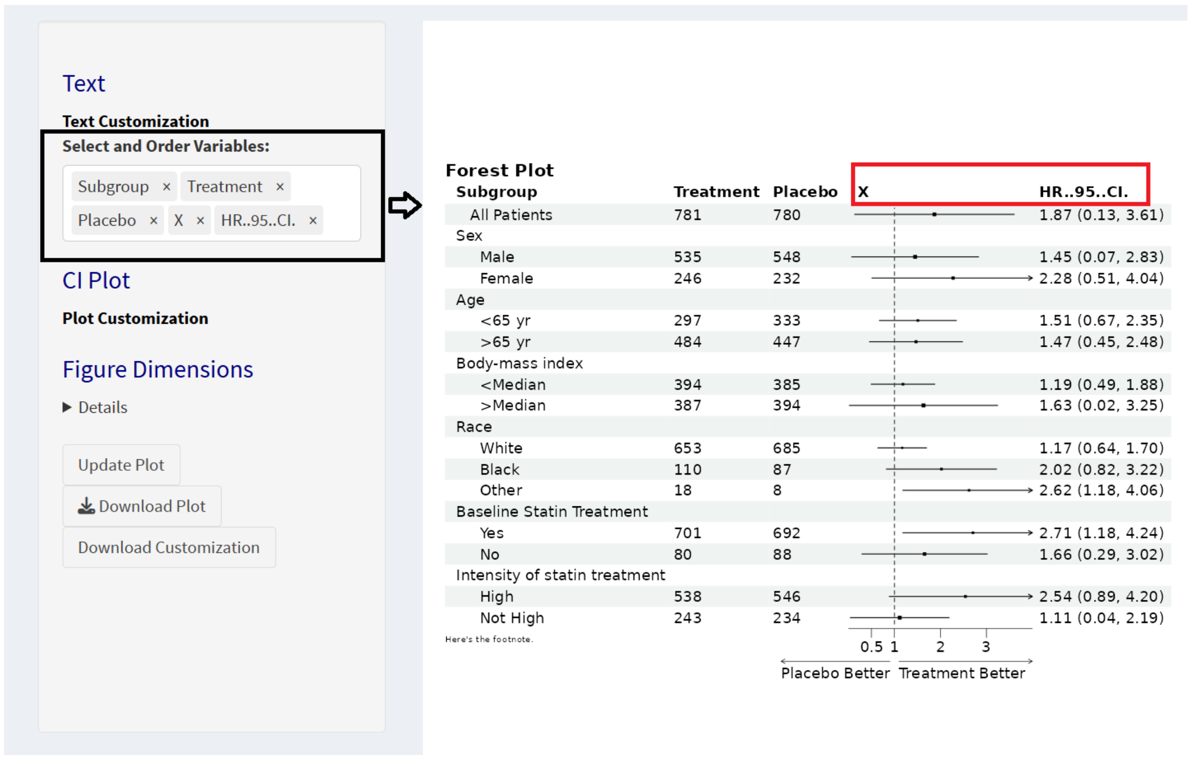
Reference Line

Forest Plot

SEQ	Subgroup	Placebo	Treatment	HR..95..CI.	X
1	All Patients	780	781	1.87 (0.13, 3.61)	
2	Sex				
3	Male	548	535	1.45 (0.07, 2.83)	
4	Female	232	246	2.28 (0.51, 4.04)	
5	Age				
6	<65 yr	333	297	1.51 (0.67, 2.35)	
7	>65 yr	447	484	1.47 (0.45, 2.48)	
8	Body-mass index				
9	<Median	385	394	1.19 (0.49, 1.88)	
10	>Median	394	387	1.63 (0.02, 3.25)	
11	Race				
12	White	685	653	1.17 (0.64, 1.70)	
13	Black	87	110	2.02 (0.82, 3.22)	
14	Other	8	18	2.62 (1.18, 4.06)	
15	Baseline Statin Treatment				
16	Yes	692	701	2.71 (1.18, 4.24)	
17	No	88	80	1.66 (0.29, 3.02)	
18	Intensity of statin treatment				
19	High	546	538	2.54 (0.89, 4.20)	
20	Not High	234	243	1.11 (0.04, 2.19)	

Here's the footnote.

Figure 10. Default forest plot before applying the variable selector.


Text

Text Customization

Select and Order Variables:

Subgroup × Treatment × Placebo × X × HR..95..CI. ×

CI Plot

Plot Customization

Figure Dimensions

▶ Details

Update Plot

Download Plot

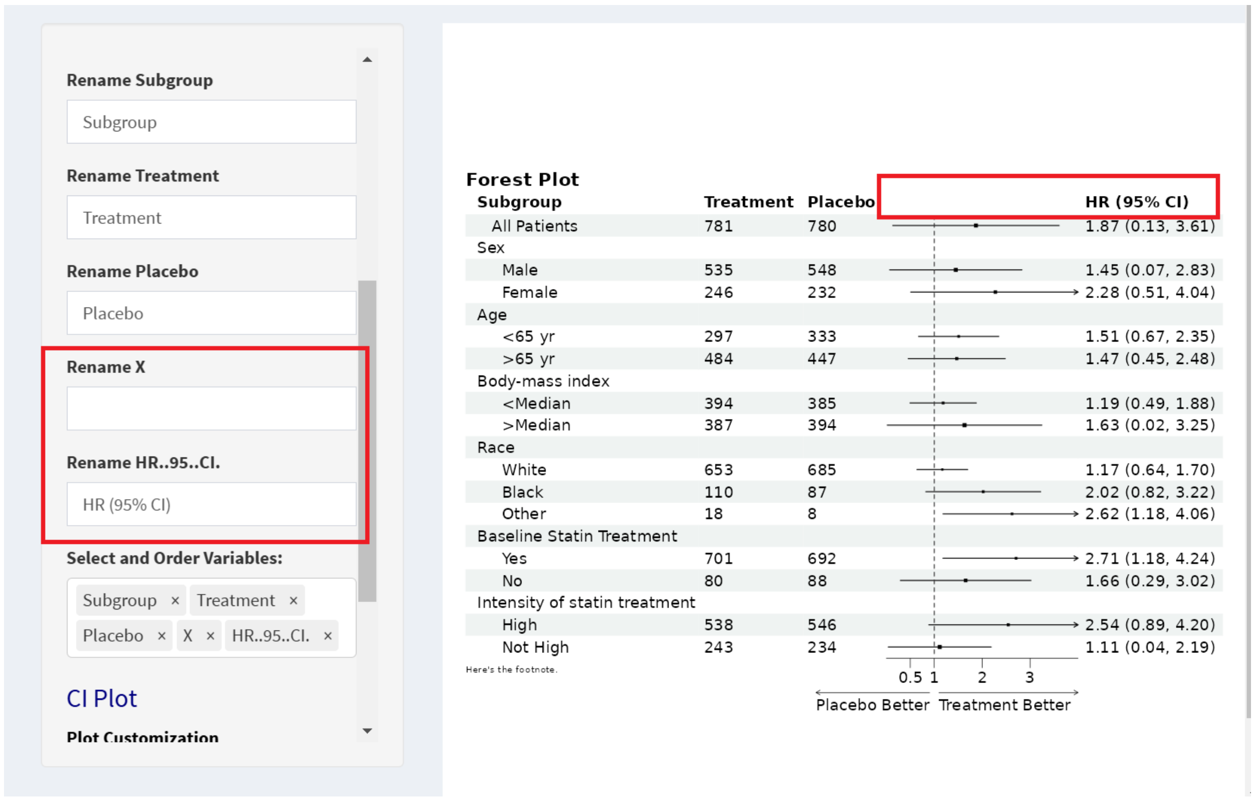
Download Customization

Forest Plot

Subgroup	Treatment	Placebo	X	HR..95..CI.
All Patients	781	780		1.87 (0.13, 3.61)
Sex				
Male	535	548		1.45 (0.07, 2.83)
Female	246	232		2.28 (0.51, 4.04)
Age				
<65 yr	297	333		1.51 (0.67, 2.35)
>65 yr	484	447		1.47 (0.45, 2.48)
Body-mass index				
<Median	394	385		1.19 (0.49, 1.88)
>Median	387	394		1.63 (0.02, 3.25)
Race				
White	653	685		1.17 (0.64, 1.70)
Black	110	87		2.02 (0.82, 3.22)
Other	18	8		2.62 (1.18, 4.06)
Baseline Statin Treatment				
Yes	701	692		2.71 (1.18, 4.24)
No	80	88		1.66 (0.29, 3.02)
Intensity of statin treatment				
High	538	546		2.54 (0.89, 4.20)
Not High	243	234		1.11 (0.04, 2.19)

Here's the footnote.

Figure 11. Forest plot after using variable selector to remove and reorder columns.



Rename Subgroup

Subgroup

Rename Treatment

Treatment

Rename Placebo

Placebo

Rename X

Rename HR..95..CI.

HR (95% CI)

Select and Order Variables:

Subgroup × Treatment × Placebo × X × HR..95..CI. ×

CI Plot

Plot Customization

Forest Plot

Subgroup	Treatment	Placebo	HR (95% CI)
All Patients	781	780	1.87 (0.13, 3.61)
Sex			
Male	535	548	1.45 (0.07, 2.83)
Female	246	232	2.28 (0.51, 4.04)
Age			
<65 yr	297	333	1.51 (0.67, 2.35)
>65 yr	484	447	1.47 (0.45, 2.48)
Body-mass index			
<Median	394	385	1.19 (0.49, 1.88)
>Median	387	394	1.63 (0.02, 3.25)
Race			
White	653	685	1.17 (0.64, 1.70)
Black	110	87	2.02 (0.82, 3.22)
Other	18	8	2.62 (1.18, 4.06)
Baseline Statin Treatment			
Yes	701	692	2.71 (1.18, 4.24)
No	80	88	1.66 (0.29, 3.02)
Intensity of statin treatment			
High	538	546	2.54 (0.89, 4.20)
Not High	243	234	1.11 (0.04, 2.19)

Here's the footnote.

Figure 12. Forest plot after renaming columns.

The figures above demonstrate the forest plot variable selector UI, showcasing its ability to modify the displayed variables interactively. The first figure shows the default forest plot generated from the imported datasets. The second figure shows how users can utilize the variable selector to remove `SEQ` column and reorder the `Placebo` and `Treatment` columns. The final figure presents the result after renaming column headers to correct encoding issues and improve clarity.

Implementation in R Shiny

The "variable selector" is implemented using `selectizeInput`, which supports multi-selection and drag-and-drop reordering:

```
#UI Component
uiOutput(ns("variable_selector"))


#Server Component
output$variable_selector <- renderUI({
  req(selectDATA())
  column_names <- colnames(selectDATA())
  selectizeInput(
    ns("selected_vars"),
    "Select and Order Variables:",
    choices = column_names,
    selected = column_names,
    multiple = TRUE,
    options = list(plugins = list('remove_button', 'drag_drop'))
  )
})
```

Additionally, the “rename UI” dynamically generates text input fields for renaming selected variables:

```
#UI Component
uiOutput(ns("rename_ui"))

#Server Component
output$rename_ui <- renderUI({
  req(input$selected_vars)
  lapply(input$selected_vars, function(var) {
    textInput(ns(paste0("rename_", var)), paste("Rename", var), value = var)
  })
})
```

RESET TO DEFAULT SETTINGS FOR USER CONFIGURATIONS

Interactive visualization applications often include extensive customization options, allowing users to tailor outputs to their needs. However, managing numerous modifications can be challenging. A Reset to Defaults feature ensures users can quickly revert all settings to their initial state, enhancing usability and efficiency.

This feature restores default settings for text inputs, variable selections, colors, markers, and numeric adjustments with a single click, streamlining the user experience. Below is an example of integrating the Reset to Defaults functionality in the Swimmer Plot module. Figure 13 shows a swimmer plot with all applied custom settings, while Figure 14 demonstrates the results after pressing the Reset to Defaults button, returning all settings to their default states.

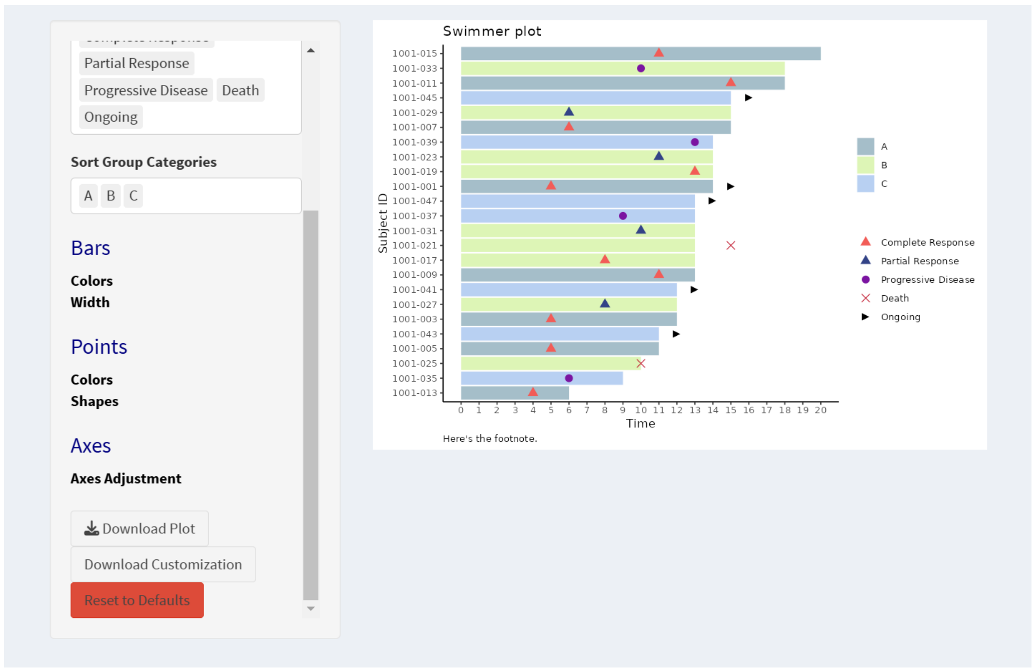


Figure 13. Swimmer plot with customized settings, before use of the 'Reset to Defaults' feature.

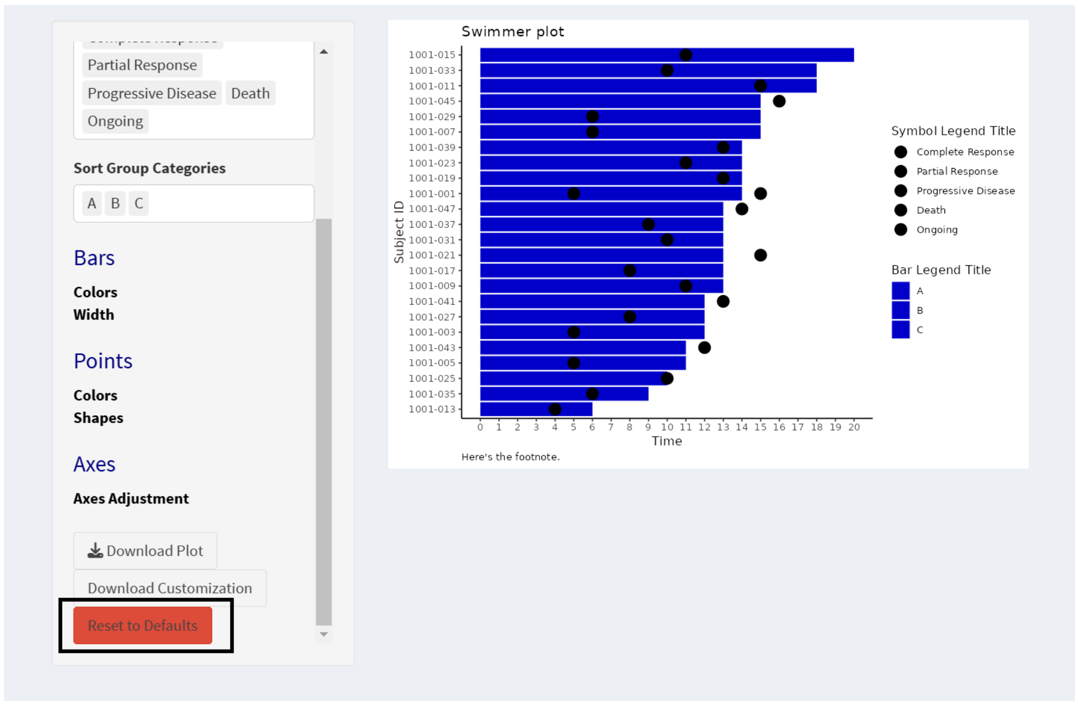


Figure 14. Swimmer plot after applying 'Reset to Defaults' to restore default settings.

Implementation in R Shiny

```
#UI Component
actionButton(ns("reset_button"), "Reset to Defaults", class = "btn-danger")

#Server Component
observeEvent(input$reset_button, {
  updateTextInput(session, "titlename", value = "Swimmer plot")
  updateNumericInput(session, "width", value = 0.9)
  updateSelectInput(session, "filter_value", choices = c("All"))

  # Reset additional UI elements...
  # updateColourInput(session, ...)
  # updateSelectInput(session, ...)
})
```

CONCLUSION

This paper presents the development of an R Shiny figures application, designed to improve accessibility, reproducibility, and efficiency in clinical data visualization. While the tool successfully integrates modular design and interactive widgets, several challenges were encountered during development, highlighting areas for further refinement.

One key challenge was {ggplot2}'s limitations within Shiny, particularly in handling dynamic text placement. For example, in the eDish plot, subject IDs could not be displayed interactively as freely as in table-based outputs due to {ggplot2}'s static rendering behavior. Workarounds like {ggrepel} help with label positioning, but ensuring clarity in interactive settings remains a challenge.

Another issue was data format consistency. The application requires long format data (vertical), but many SAS datasets are in wide format (horizontal), requiring transposition or reshaping before visualization. Not all datasets can be automatically converted, making data standardization before import an important consideration. Additionally, variable assignment inconsistencies sometimes led to errors. For example, in the Kaplan-Meier module, the status variable must be properly formatted (e.g., censored vs. event) since it cannot exceed a certain number of categories. A more structured approach to default variable mapping and input validation could improve stability.

Future challenges include the development of a quality control standard operating procedure for the app. If figures developed in the app are to be used for regulatory submissions, data must be finalized and validated before being imported – and must not be further filtered within the app. However, data manipulated using the Data Upload module cannot be validated in the app's present state. While development of standard operating procedures is outside the scope of our work, future updates would need to address at least two key functionalities necessary for the validation process: 1. data manipulation reproducibility and 2. saving figure data. Currently, the code used to filter the data is not captured by the app, but implementation of {shinymeta} into the module could help solve the issue. The code produced from the "Download Code" feature in plot modules would also need to include the {shinymeta} code from the data upload module. Furthermore, the data manipulated within the app would need to be saved and exported so that it can be validated. Ideally, this would occur simultaneously with the plot download so that data and plot are connected by timestamps. Throughout development, we structured our code to allow for future enhancements while maintaining the scope and usability of the application. As a post-deployment tool, future updates must be seamlessly integrated based on user feedback. For example, users may request additional file format support for data import or export or may find it inefficient to create new variables externally before reimporting data. While adding data manipulation capabilities could improve workflow efficiently, it may also challenge the app's no-code design principle, necessitating a careful balance between usability and feature complexity.

This paper shares our experience in developing an R Shiny figures application, focusing on practical challenges and lessons learned. By outlining key considerations and solutions, we aim to provide insights for those developing similar applications, helping them navigate common obstacles and make informed design decisions when integrating ggplot2, Shiny, and modular app architecture in clinical data visualization.

REFERENCE

Attali D (2023). _colourpicker: A Colour Picker Tool for Shiny and for Selecting Colours in Plots_. https://github.com/daattali/colourpicker, https://daattali.com/shiny/colourInput/.

Chang W, Cheng J, Allaire J, Sievert C, Schloerke B, Xie Y, Allen J, McPherson J, Dipert A, Borges B (2023). _shiny: Web Application Framework for R_. https://shiny.posit.co/, https://github.com/rstudio/shiny.

Cheng J, Sievert C (2021). _shinymeta: Export Domain Logic from Shiny Meta-Programming_. https://rstudio.github.io/shinymeta/, https://github.com/rstudio/shinymeta.

Fay, C., Rochette, S., Guyader, V., & Girard, C. (2021). Engineering Production-Grade Shiny Apps (1st ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9781003029878

Fay C, Guyader V, Rochette S, Girard C (2023). _golem: A Framework for Robust Shiny Applications_. R package version 0.4.1, <https://github.com/ThinkR-open/golem>.

Straub B, Bundfuss S, Dickinson J, Farrugia R, Forys A, Grasselly D, Kulkarni D, Mancini E, Mascary S, Miller G, Shapcott S, Simms E, Thoma S, Zhang K, Zhu Z (2023). _admiral: ADaM in R Asset Library_. https://pharmaverse.github.io/admiral/, https://github.com/pharmaverse/admiral.

Wickham H. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.

Wickham H, Bryan J (2023). _readxl: Read Excel Files_. https://readxl.tidyverse.org, https://github.com/tidyverse/readxl.

Wickham H, Hester J, Bryan J (2023). _readr: Read Rectangular Text Data_. https://readr.tidyverse.org, https://github.com/tidyverse/readr.

Wickham H, Averick M, Bryan J, Chang W, McGowan LD, François R, Grolemund G, Hayes A, Henry L, Hester J, Kuhn M, Pedersen TL, Miller E, Bache SM, Müller K, Ooms J, Robinson D, Seidel DP, Spinu V, Takahashi K, Vaughan D, Wilke C, Woo K, Yutani H (2019). "Welcome to the tidyverse." _Journal of Open Source Software_, *4*(43), 1686. doi:10.21105/joss.01686 <https://doi.org/10.21105/joss.01686>.

APPENDIX

CODE SNIPPET 1. CODE STRUCTURE TO HANDLE CODE

```
# create reactive object to store plot layer for individual lines
individual_lines <- metaReactive({
  return(list(
    geom_line(# (redacted plot code ) #),
      scale_linetype_discrete(guide = guide_legend(ncol = 2), name = toupper(..(
        input$legendTitle
      ))),
      ..(id_line_color())
    ))
})

  # create reactive object to store plot layer for group lines
  group_mean_lines <- metaReactive({
    return(
      list(
        ggnewscale::new_scale_color(),
        stat_summary(# (redacted plot code ) #),
          scale_linetype_discrete(guide = guide_legend(ncol = 2), name = toupper(..(
            input$legendTitle
          ))),
          ..(grp_colors())
        )
      )
  })

    # create reactive object to store plot layer for group mean points
    group_mean_points <- metaReactive({
      return(list(
        ggnewscale::new_scale_color(),
        stat_summary(# (redacted plot code) #),
          ..(grp_colors())
        ))
    })

      # create reactive object to store plot layer for group error bars
      group_mean_errbar <-  metaReactive({
        return(list(
          ggnewscale::new_scale_color(),
          stat_summary(# (redacted plot code ) #),
            ..(grp_colors())
          ))
      })

        # create plot and store in reactive object
        line_gg <- metaReactive({
          #create base layer of plot
          p <- ggplot() + geom_vline(xintercept = 0) +
            geom_hline(yintercept = 0)

          # add layers based on toggles in GUI
          if (input$id_line == TRUE) {
            p <- p + ..(individual_lines())
          }
          if (input$mean_line == TRUE) {
            p <- p + ..(group_mean_lines())
          }
          if (input$mean_points == TRUE) {
            p <- p + ..(group_mean_points())
          }
          if (input$errbar == TRUE) {
            p <- p + ..(group_mean_errbar())
          }

          # add labels and theme to the plot
          pFinal <- p + ..(xtick_lab()) + ..(xlabs()) +
            ..(default_plot_theme())
          return(pFinal)
        })
```

CODE SNIPPET 2. CORE LOGIC BEHIND DATA INPUT MODULE

```
# Upon triggering the "update table" button
observeEvent(input$update_table, {
  # check if column subset and custom filter are empty
  if (length(temp_cols()) == 0 & nchar(temp_filter()) == 0) {
    finalData$dataMrg = finalData$dataIN
  } else if (length(temp_cols()) == 0 &
             nchar(temp_filter()) > 0)  {
    # only keep custom filter
    # check if custom filter is valid
    validate(need(finalData$dataIN %>%
                    filter(eval(
                      str2lang(temp_filter())
                    )), "Check for errors"))
    # save application from failing if invalid filter
    finalData$dataMrg = tryCatch({
      finalData$dataIN %>%
        filter(eval(str2lang(temp_filter())))
    }, error = function(e) {
      showNotification(paste0(e), type = 'err')
    }, warning = function(w) {
      showNotification(paste0(w), type = 'warning')
    })
  } else if (length(temp_cols()) > 0 &
             nchar(temp_filter()) == 0) {
    # only keep columns from selectInput()
    finalData$dataMrg = finalData$dataIN %>%
      select(any_of(temp_cols()))
  } else if (length(temp_cols()) > 0 &
             nchar(temp_filter()) > 0) {
    # combine column subset and custom filter procedures if neither are empty
    finalData$dataMrg = tryCatch({
      finalData$dataIN %>%
        filter(eval(str2lang(temp_filter()))) %>%
        select(any_of(temp_cols()))
    }, error = function(e) {
      showNotification(paste0(e), type = 'err')
    }, warning = function(w) {
      showNotification(paste0(w), type = 'warning')
    })
  }
})
```

CODE SNIPPET 3. CORE LOGIC BEHIND MULTI-PLOT VIEW

```
barscatter <- metaReactive({
  # (redacted set-up code) #
  # (redacted code to create list of colors for plot) #

  ## check if a 'plot-by' variable was selected
  if (!is.null(plotByVar())) {
    # if selected, store the levels of the variable in a list
    unq_plotby <- as.list(unique(barData()[[input$plotBy]]))
  } else {
    # if not selected, assign as 1 so map function will still work properly
    unq_plotby <- 1
  }

  ## Use the map function to loop through the unique levels of the 'plot-by'
  ## variable. For each element, the map function will: 1. subset the data 2.
  ## create the plot 3. save the plot as an element of a list
  tmpPlotList <- purrr::map(unq_plotby, function(x) {
    ## subset data
    if (!is.null(plotByVar())) {
      tmpData <- barData() %>% filter(!!rlang::sym(plotByVar()) == x)
      subT = x
    } else {
      tmpData <- barData()
      subT = NULL
    }
    ## create plot
    # (redacted code to create plots) #
    ## return plot as an element of the list
    return(p)
  })
  ## save list of plots
  return(tmpPlotList)
})

output$plotOut <- renderUI({
  plots <- req(barscatter())
  tagList(map(plots, ~ createUI(.)))
```

ACKNOWLEDGEMENTS

We sincerely thank Gita Motupally, Michael Hagendoorn, and Arlene Coleman for their support and invaluable feedback throughout the development of our paper. We are also grateful to Shang-Ying Liang for her thoughtful visualization suggestions based on biostatistician's needs. Additionally, we appreciate the participation and support of Matt Smith during this process. Finally, we appreciate the collaboration from colleagues across Pfizer, especially those who contributed suggestions and user feedback.

CONTACT INFORMATION

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

Yi Guo
Pfizer Inc.
Yi.Guo@pfizer.com

Mattew Salzano
Pfizer Inc.
Matthew.Salzano@pfizer.com

Nicholas Sun
Pfizer Inc.
Nicholas.Sun@pfizer.com

Sean Healey
Pfizer Inc.
Sean.Healey@pfizer.com