PharmaSUG 2025 - Paper DV-160

An Integrated R Shiny Solution for Dynamic Subgroup Adjustments and Customizations

Yi Guo, Pfizer Inc.

ABSTRACT

A major challenge in using SAS® macros and traditional R tools such as ggplot2 for figure generation arises when categories within a variable change with dataset updates. Such changes, including the addition of new categories as data accumulate or the exclusion of existing ones due to updated filtering criteria, require code adjustments to subgroup settings like ordering, colors, markers, or other style options. Done manually, this process is inefficient, error-prone, and particularly challenging in clinical trial reporting, where data get frequent updates and accurate and timely visualizations are critical. While effective for static visualizations, these tools lack flexibility for dynamic, no-code customization, limiting their scalability.

To address this, we introduce an integrated set of interactive features powered by R Shiny, including automatic subgroup detection, interactive legend reordering, and customizable color pickers. These features enable users to dynamically adjust subgroup settings, reorder legends through the interface, and freely select reordered subgroup colors, offering greater design flexibility. All updates are synchronized to ensure consistency, accuracy, and an intuitive workflow for creating visualizations.

This integrated approach bridges the gap between manual adjustments and dynamic customization, improving efficiency and adaptability for dynamic datasets. This paper details the implementation and demonstrates applications through real-world examples.

INTRODUCTION

In clinical trial reporting, figure generation is still predominantly performed using static methods. A typical workflow involves preparing an analysis-ready dataset with all required variables and then passing it into a pre-defined SAS macro (using SAS Graph Template Language (GTL)) or an R function (using ggplot2) to create figures. These macros and functions rely on parameter assignments to control elements such as subgroup classifications, colors, markers, and other style options.

For example, a SAS macro might be used as follows to generate a figure with subgroup-based styling:

```
%create_figure(
  x_var       = x,
  y_var       = y,
  subgroup    = group,
  color_list  = #0000FF #FF0000 #008000,
  marker_list = circle square triangle
);
```

Similarly, in R with ggplot2, an equivalent function call would look like this:

```r
create_figure(
  x_var       = "x",
  y_var       = "y",
  subgroup    = "group",
  color_list  = c("#0000FF", "#FF0000", "#008000"),
  marker_list = c(16, 17, 18)
)
```

To update parameter values and modify visual outputs, users have to manually update the underlying code. This process can be inefficient and prone to errors, especially in clinical studies where datasets are frequently updated, requiring the rapid generation of multiple figures. As a result, traditional static visualization approaches face several key limitations, including:

1. Uncertainty in determining categorical levels

Without an initial frequency check using `PROC FREQ` in SAS or `table()` in R, identifying the exact categories within a variable can be challenging, especially with new or updated datasets. Typically, programmers first check the categories in the grouping variable and then adjust the order of categories, color values, marker values, etc.

2. Dependency on manual factor conversion for correct legend ordering in R

In R, categorical variables must be explicitly converted to factors with predefined levels to ensure correct legend ordering, whether the data comes from Excel files, SAS datasets, delimited text files, or any other formats. Without proper factor definition, legends may appear in the wrong order even if colors or markers are assigned correctly.

3. Inefficient manual updates for subgroup mappings and visual attributes

Traditional methods typically define category mappings within a variable, assigning attributes such as color and marker based on the Case Report Form (CRF) for predefined variables and ADaM specifications for derived variables. The ordering of subgroups, including their assigned color and marker values, is manually defined in the code.

Each category requires a predefined color, marker, line type, etc. However, when subgroup compositions change, such as when categories are added or removed, these mappings must be manually updated to stay aligned with the latest data. Additionally, programmers must cross-reference external documentation, such as hex codes for colors, marker identifiers in SAS or numeric values in R, and often switch between multiple files to ensure accuracy. This process is cumbersome and adds to the maintenance burden.

These limitations of traditional methods highlight the importance of a more flexible tool for efficiently generating and updating figures. R Shiny is a great choice, as it eliminates predefined settings and offers an interactive web-based solution. It enables direct adjustments to subgroup classifications and style settings while ensuring seamless visualization updates as data evolves, making it particularly well-suited for repetitive tasks.

In the following sections, we compare traditional methods and R Shiny for figure generation across various aspects. Based on these comparisons, we present an R Shiny integrated solution to address the three key limitations of traditional methods. This solution incorporates automatic subgroup detection, interactive legend reordering, and customizable color selection. The discussion includes code snippets, an overview of relevant R packages, and real-world applications.

COMPARISON OF TRADITIONAL METHODS AND R SHINY FOR DYNAMIC FIGURE GENERATION

The following table compares R functions with ggplot2, SAS macros with GTL, and R Shiny with ggplot2, covering key aspects from user accessibility to modification flexibility, data handling, and maintenance. It summarizes the key differences and considerations for each approach.

Table 1. Comparison of Traditional Methods and R Shiny for Figure Generation

Aspect	R Functions (ggplot2)	SAS Macros (GTL)	R Shiny (ggplot2)
User-Friendliness	For programmers; coding required	For programmers; coding required	All users; no coding needed
Static vs Interactive	Static, requires manual code adjustments	Static, requires manual code adjustments	Interactive, allows real-time modifications
Modification Method	Modify via function parameters	Modify via macro code	Modify via UI controls
Customization	Customizable via code	Customizable via code	Highly customizable via UI
Scalability	Reusable for the same specific figure design; manual edits needed for different variables, structures, or layouts	Reusable for the same specific figure design; manual edits needed for different variables, structures, or layouts	Highly scalable when pre-configured; supports dynamic updates via UI with minimal coding for variable or layout changes; major structural changes may still require code updates
Data Upload	Requires preprocessing	Requires preprocessing	Supports direct file uploads via UI
Data Updates	Manual reload and rerun	Manual updates and rerun	Auto-updates with data and inputs
Code Maintenance	Low; code is simpler and easier to read; function encapsulates common settings but must be modified for new needs	High; GTL code is complex, and modifications are not intuitive	Moderate; modular and reusable components reduce maintenance

Table 1 shows that for simple and reusable data visualization workflows, R functions with ggplot2 offer an efficient solution. Compared to SAS, R code is more concise and easier to read. As a declarative approach, ggplot2 simplifies maintenance by eliminating the need for manual management of every detail. For standardized yet complex or highly customized figure outputs, SAS macros with GTL provide greater control but come with higher maintenance costs. However, for interactive and user-friendly data visualization, R Shiny combined with ggplot2 is the most flexible and scalable solution.

Based on these insights, R Shiny combined with ggplot2 can fully address those three limitations of static visualization methods. Its visualization capabilities remove the need to pre-examine categories, while its interactive features allow users to adjust legend order directly within the interface and customize colors, shapes, and other style options for subgroups in real time.

R SHINY-BASED DYNAMIC CUSTOMIZATION SOLUTIONS

In this section, we demonstrate how Automatic Subgroup Detection, Interactive Legend Ordering, and Customizable Color and Shape work together to enhance interactive visualization. The eDish (Enhanced Drug-Induced Serious Hepatotoxicity) plot is selected as the example because it requires both color and shape customizations, whereas plots such as Kaplan-Meier plot typically involve only color adjustments. Since color and shape are parallel attributes, shape selection can be introduced as an extension when discussing color customization. All figures presented in this paper are based on dummy data and are intended for illustrative purposes only.

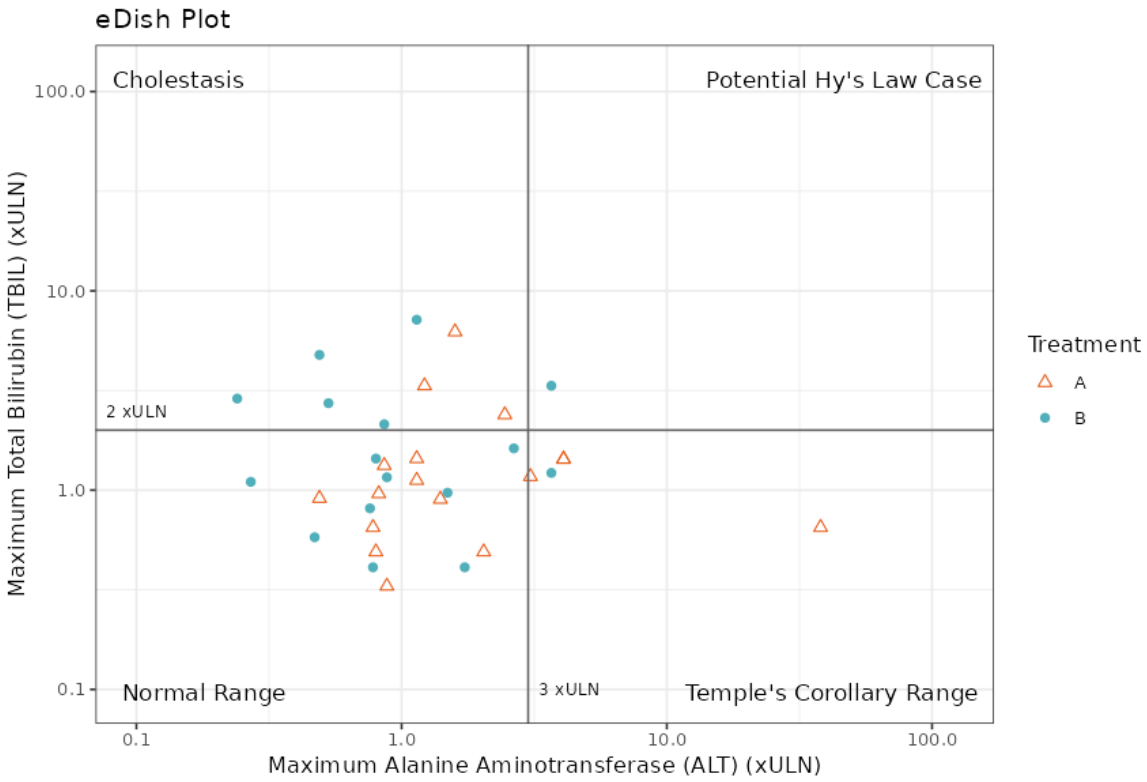

Figure 1. eDish Plot Example with Color and Shape Customizations

In the eDish plot, these features are integrated to improve subgroup representation and user interaction:

a. **Automatic Subgroup Detection**: Identifies all categories within a selected grouping variable and automatically applies them to color, shape, and legend ordering. This allows users to see all subgroup categories without manually inspecting the data or writing additional code.
b. **Interactive Legend Ordering**: Works seamlessly with Automatic Subgroup Detection, allowing users to reorder subgroups directly through the interface without modifying the underlying data or

writing additional code. The updated order is automatically reflected in the legend, ensuring alignment with the predefined statistical order.

c. **Customizable Color and Shape**: Enables users to customize colors and shapes for reordered subgroups, improving visual clarity.

MODULAR DESIGN WITH {GOLEM} AND NS()

`{Golem}` is a widely used framework for modular Shiny development, providing a structured approach for scalability and maintainability. In a modular Shiny app, each UI and server component is encapsulated within a module, preventing variable conflicts and improving application management. In each module, we use the namespace function `ns()` to keep `input$` and `output$` variables isolated across modules. For example, in the eDish plot module:

```
mod_edish_ui <- function(id) {
  ns <- NS(id)  # Generate a unique namespace
  tagList(
    sidebarPanel(
      selectInput(ns("group_var"), "Group", choices = NULL),
      uiOutput(ns("sortable_gvar")),
      uiOutput(ns("dynamic_colors")),
      uiOutput(ns("shape_inputs"))
          …
    ),
    mainPanel(
      plotOutput(ns("edish"))
    )
  )
}

mod_edish_server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$edish <- renderPlot({
      …
    })
```

This structure is essential for implementing automatic subgroup detection, interactive legend ordering, and customizable color and shape selection, ensuring each feature remains independent and reusable. With this foundation in place, we now explore automatic subgroup detection as the first key feature.

AUTOMATIC SUBGROUP DETECTION

When a user selects a grouping variable for an eDish plot through the interface, the system automatically detects all its categories.

1. UI Component

The following line of code creates a dropdown menu for selecting a grouping variable (categorical):

```r
selectInput(ns("group_var"), "Group", choices = NULL)
```

Here, `selectInput(inputId, label, choices)` defines a dropdown menu for selecting a categorical variable. The `ns("group_var")` function applies a namespace to ensure the input ID is unique within the module. The label "Group" is displayed in the UI, and `choices = NULL` initializes an empty selection list, allowing the server to dynamically populate the available options.

2. Server Component

Server-side automatic subgroup detection is achieved through reactive expressions [1] [2], which dynamically update the available grouping variables based on user-selected data. `selectDATA()` retrieves the selected dataset, and `updatedX()` dynamically extracts its column names. The `observe()` function [3] then updates the `group_var` dropdown via `updateSelectInput()`, ensuring that users can seamlessly browse and select a grouping variable from the available options. As a result, when a variable is selected in the “Group” widget, its variable name is immediately reflected in the dropdown.

```r
selectDATA <- reactive({                                    [1]
  filePath <- dataIN()  # Load dataset from input
})

updatedX <- reactive({                                      [2]
  return(colnames(selectDATA()))
})

observe({                                                   [3]
  updateSelectInput(session, "group_var", choices = updatedX())
})
```

3. Subgroup Selection Dropdown Preview

Based on the above UI and server design, the “Group” widget is created to select a grouping variable (as shown in Figure 2). In this example, the “treatment” variable is chosen as the grouping variable. Upon selection, the system automatically detects all available categories, A and B, representing different treatment groups, and displays them in the “Sort Group Categories” dropdown as part of the Interactive Legend Reordering functionality. This process will be further discussed in the following section. If the default order is not as expected, users can manually reorder the categories by removing and reselecting them (as shown in Figure 2). This provides flexibility in defining the order, such as selecting B before A or A before B, based on user preference.

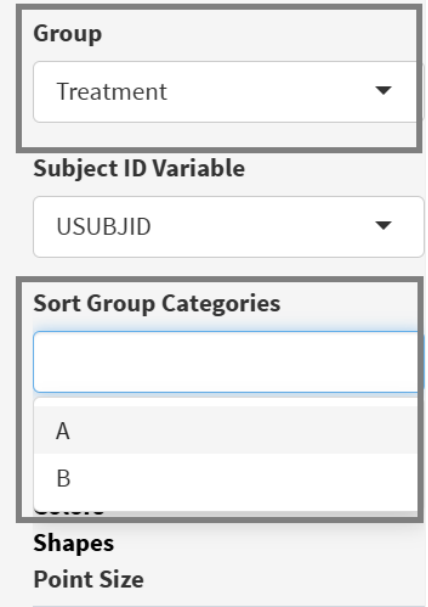

Figure 2. Subgroup Selection and Reordering Interface

INTERACTIVE LEGEND REORDERING

After all the analysis variables are selected, including x, y and grouping variables, a legend is generated based on the detected grouping variable. If the original order is not as expected, users can reorder the legend interactively to customize the display.

1. UI Component

Both legend ordering and reordering are handled using the same UI component:

```
uiOutput(ns("sortable_gvar"))
```

Of note, `uiOutput(ns("sortable_gvar"))` does not define the dropdown itself but acts as a placeholder where the UI element will be generated dynamically in the server component using `renderUI()`.

2. Server Component

In the server component, the legend order is updated by adjusting the factor levels of the grouping variable before passing the data into `ggplot()`. This can be achieved in two key steps:

Step 1: Rendering a sortable UI for legend reordering.

Step 2: Reordering the legend before plotting.

Step 1: Rendering a sortable UI for legend reordering

The following `renderUI()` function generates a sortable dropdown (`selectInput`) where users can manually reorder the grouping variable `input$group_var` categories:

```
output$sortable_gvar <- renderUI({
  req(selectDATA(), input$group_var)                                    [1]

  gvar_levels <- unique(selectDATA()[[input$group_var]])                [2]


  selectInput(ns("gvar_order"), "Sort Group Categories",                [3]
              choices = gvar_levels,
              selected = gvar_levels,  # Default order
              multiple = TRUE)

})
```

Brief explanation of the code logic:
[1] Ensure data and input exist.
[2] Extract unique subgroup categories.
[3] Create a selectable list for users to reorder group categories.

The code retrieves unique subgroup categories (`gvar_levels`) and displays them in a multi-select dropdown using `selectInput()`, allowing users to adjust the order. The selected order is stored in `input$gvar_order` and later applied in the plot to ensure the legend aligns with user preferences.

Step 2: Reordering the legend before plotting

After creating an interactive legend reordering UI, the next step is to apply the user-defined factor levels to the data before passing them to `ggplot()` for visualization.

```
generate_edish_plot <- reactive({
  req(input$xvar, input$yvar, input$group_var, …)                       [1]

  ggdata <- data.frame(x = selectDATA()[[input$xvar]],                  [2]
                       y = selectDATA()[[input$yvar]],
                       g = selectDATA()[[input$group_var]], …)

  ggdata$g <- factor(ggdata$g, levels = na.omit(unique(input$gvar_order)))   [3]

  …

  p -> ggplot(ggdata, …) + other functions                              [4]

  return(p)
```

```
})
```

Brief explanation of the code logic:
[1] Ensure required inputs exist.
[2] Create a dataframe for ggplot.
[3] Apply user-defined legend order.
[4] Generate ggplot with reordered legend.

The code extracts the grouping variable from `selectDATA()` and retrieves the user-defined group order from `input$gvar_order`. It then applies `factor()` to reorder `ggdata$g` according to the selected order, ensuring that the legend in `ggplot()` reflects the updated grouping. With this approach, the subgroup order remains consistent between plot legend and the UI selection.

3. Legend Reordering UI Preview

After selecting the grouping variable (e.g., treatment) in the blue box, users can adjust the subgroup order using the "Sort Group Categories" widget in the red box on the left. The order set in this widget is reflected in the legend order in the plot on the right. As shown in Figures 3 and 4, changing the order of A and B in the widget updates the legend accordingly, keeping the display consistent with the specified subgroup arrangement.

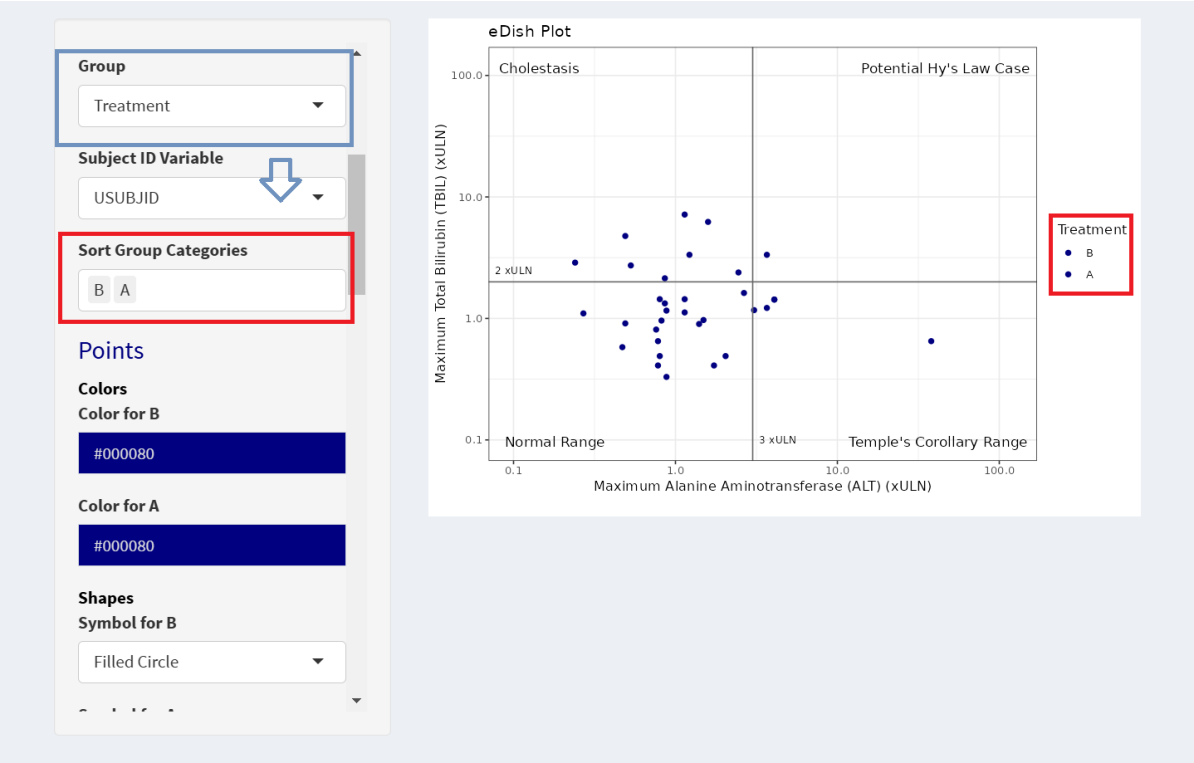

Figure 3. eDish Plot with Legend Order: B Before A

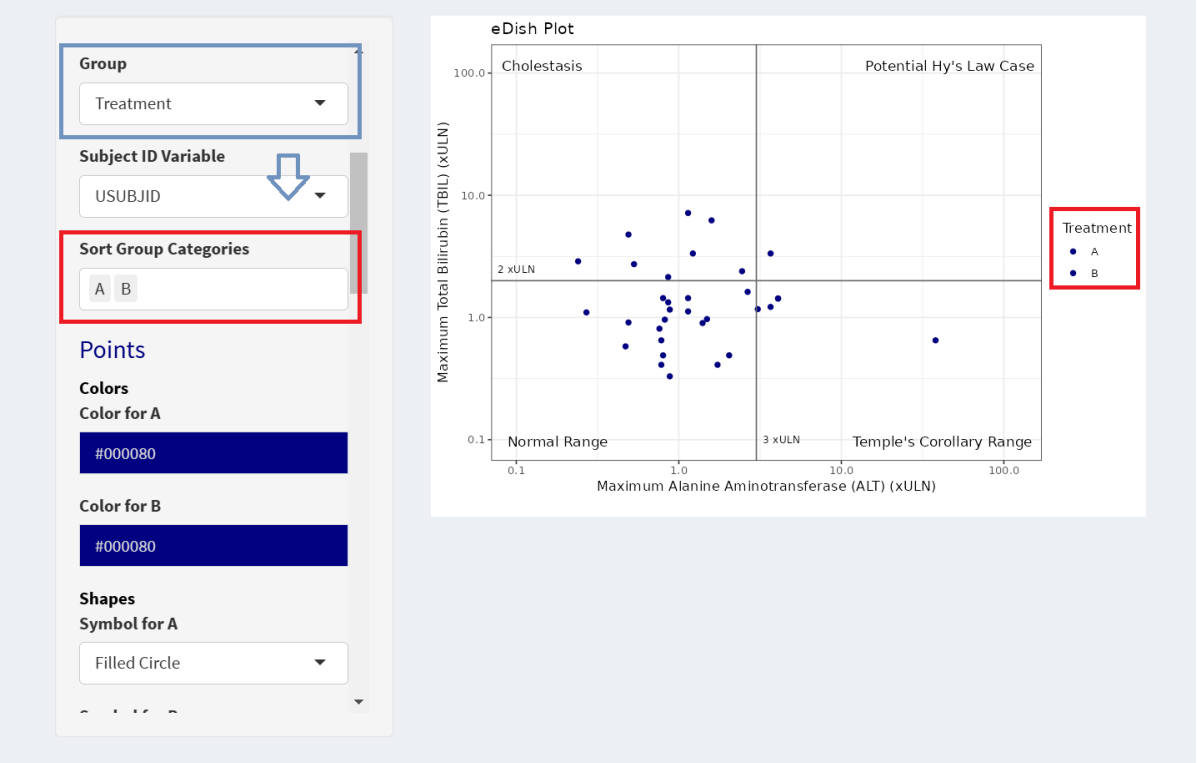

Figure 4. eDish Plot with Legend Order: A Before B

CUSTOMIZABLE COLOR AND SHAPE SELECTION

After determining the legend order, the next step is to apply user-defined colors and shapes to each subgroup.

This section is divided into five parts to provide a structured explanation of how user-defined colors and shapes are reordered and integrated into the visualization. We begin by introducing the UI component for selecting colors and shapes, followed by the server-side implementation for dynamically generating these inputs. Next, we explain how the selected attributes are applied within `ggplot()`, and finally, we demonstrate how the customized plot is rendered.

The five components are:

1. Color and Shape Selection UI Component
2. Generate Dynamic UI for Color Selection (Server Component)
3. Generate Dynamic UI for Shape Selection (Server Component)
4. Apply Color and Shape Selections in ggplot() (Server Component)
5. Render the Customized Plot (Server Component)

To implement dynamic color selection, we utilize the `{colourpicker}` package, which provides a graphical interface for selecting colors interactively. This eliminates the need to manually enter hex codes and allows users to adjust subgroup colors directly within a Shiny application.

The key function, `colorInput()`, generates a color picker UI that lets users select and modify colors in real time. In a dynamic Shiny UI, multiple `colorInput()` elements can be created based on user-defined categories, making it highly adaptable for customizing subgroup colors in ggplot-based visualizations.

To use `{colourpicker}`, it first needs to be installed and loaded:

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

1. Color and Shape Selection UI Component

The following two lines of code define UI placeholders for dynamically generated inputs, allowing users to customize the color and shape of each subgroup in the eDish plot.

```
uiOutput(ns("dynamic_colors"))
uiOutput(ns("shape_inputs"))
```

2. Generate Dynamic UI for Color Selection (Server Component)

This server-side code dynamically generates color pickers in the Shiny UI for each detected subgroup based on the user-defined order. If the legend order changes, the color pickers will automatically reorder to match the updated subgroup sequence.

```
observe({
  unique_groups <- na.omit(unique(input$gvar_order))                    [1]

  output$dynamic_colors <- renderUI({
    lapply(unique_groups, function(i){
      colourInput(                                                      [2]
        inputId = ns(paste0("color_", gsub(" ", "_", i))), # Unique ID
        label = paste("Color for", i),
        value = "#000080"  # Default color
      )
    })
  })
})
```

Here, [1] retrieves the user-defined subgroup order by extracting unique values from `input$gvar_order` and omitting any missing values. This ensures that the legend reflects the selected order rather than the dataset's default arrangement. Notably, this step must be executed after `output$sortable_gvar <- renderUI({ ... })` from Interactive Legend Reordering Server Component section, as `input$gvar_order` from [1] is only available once the UI component for reordering groups has been rendered.

In [2], `colourInput()` dynamically creates a color selector for each subgroup. `gsub(" ", "_", i)` replaces spaces in subgroup names to ensure valid and unique `inputId`s.

3. Generate Dynamic UI for Shape Selection (Server Component)

This server component dynamically generates shape selection inputs for each detected subgroup. It first ensures that the necessary data and grouping variable are available. Then, a predefined set of shape options is provided, allowing users to assign different shapes to each subgroup. The detected subgroup categories, based on the user-defined order, are retrieved and used to generate dropdown selectors, where each subgroup is assigned a unique shape selection input. By doing so, users can customize the shape representation of subgroups in the plot, ensuring consistency with the legend order. If no shape is specified, a default shape (filled circle) is applied.

```
output$shape_inputs <- renderUI({
  req(selectDATA(), input$group_var)                                    [1]

  shape_choices <- c(                                                   [2]
    "Square" = 0,
    "Circle" = 1,
    …
    "Filled Circle" = 16,
    …
  )

  gvar_levels <- na.omit(unique(input$gvar_order))                      [3]

  lapply(gvar_levels, function(level) {                                 [4]
    selectInput(
      ns(paste0("shape_", gsub(" ", "_", level))),
      paste("Shape for", level),
      choices = shape_choices,
      selected = 16. # Default shape
    )
  })
})
```

Brief explanation of code logic:
[1] Ensure the data and grouping variable exist.
[2] `shape_choices` defines shape encodings for different shape styles.
[3] Retrieve user-defined group order.
[4] `lapply()` dynamically generates `selectInput()` components, allowing users to assign unique shapes to different subgroups.

Of note, since [3] depends on the reordered legend, it should be executed after `output$sortable_gvar <- renderUI({ … })` mentioned in the Interactive Legend Reordering Server Component section to maintain consistency. Similar to the dynamic color picker UI, this step must be placed after legend reordering to ensure the correct subgroup order is applied.

4. Apply Colors and Shapes Selections in ggplot() (Server Component)

Once the user-defined colors and shapes are determined, the next step is to incorporate them into the visualization. These selections are mapped to the corresponding subgroups and applied in ggplot2 to ensure consistency in the final output.

```
generate_edish_plot <- reactive({
  req(input$xvar, input$yvar, input$group_var, …)

  ggdata <- data.frame(
    x = selectDATA()[[input$xvar]],
    y = selectDATA()[[input$yvar]],
    g = selectDATA()[[input$group_var]], …)

  # Apply user-defined legend order
  ggdata$g <- factor(ggdata$g, levels = na.omit(unique(input$gvar_order)))

  # Collect selected colors for each group
  colors <- sapply(unique_groups, function(group) {                    [1]
    group_fixed <- gsub(" ", "_", group)
    color_value <- input[[paste0("color_", group_fixed)]]
    if (!is.null(color_value) && color_value != "") {
      return(color_value)
    } else {
      return("#000080")  # Default color
    }
  })
  group_colors <- setNames(as.list(colors), unique_groups)

  # Collect selected shapes for each group
  group_shapes <- sapply(unique_groups, function(group) {              [2]
    group_fixed <- gsub(" ", "_", group)
    shape_value <- input[[paste0("shape_", group_fixed)]]
    if (!is.null(shape_value) && shape_value != "") {
      return(shape_value)
    } else {
      return(16)  # Default shape
    }
  })

  p -> ggplot(ggdata, …) +

  #plot with custom group colors
  scale_color_manual(values=unlist(group_colors))+                     [3]
  #plot with custom group shapes
  scale_shape_manual(values = as.numeric(unlist(group_shapes)))+       [4]
  …
```

```
  return(p)

})
```

Here, [1] retrieves user-selected colors for each subgroup using `sapply()`, ensuring that each group is assigned a valid color. Instead of a `for` loop, `sapply()` is more concise, and it automatically simplifies output to a vector, making it easier to use in `ggplot()`. To maintain consistency with dynamically generated `inputId`s, `gsub(" ", "_", group)` replaces spaces with underscores in subgroup names. The function then extracts the selected color from the UI (`input[[paste0("color_", group_fixed)]]`) and assigns a default color (`#000080`, presenting navy blue) if no selection is made. Finally, `setNames(as.list(colors), unique_groups)` creates a named list mapping each subgroup to its corresponding color for use in `ggplot()`.

[2] uses `sapply()` to retrieve user-selected shapes for each subgroup, ensuring that the correct shape is applied in the plot. Similar to [1], spaces in subgroup names are replaced with underscores using `gsub(" ", "_", group)` to match the dynamically generated `inputId`. The function then extracts the selected shape from `input[[paste0("shape_", group_fixed)]]`. If no shape is selected, a default value (16, representing a filled circle in ggplot2) is assigned. The result is a vector where each subgroup is mapped to its corresponding shape, ensuring consistency in visualization.

In [3] and [4], `scale_color_manual()` and `scale_shape_manual()` ensure that user-defined colors and shapes are correctly applied to each subgroup in `ggplot()`.`unlist(group_colors)` and `unlist(group_shapes)` extract user selections, while `as.numeric()` converts shape values to numeric to compatible with `ggplot()`.

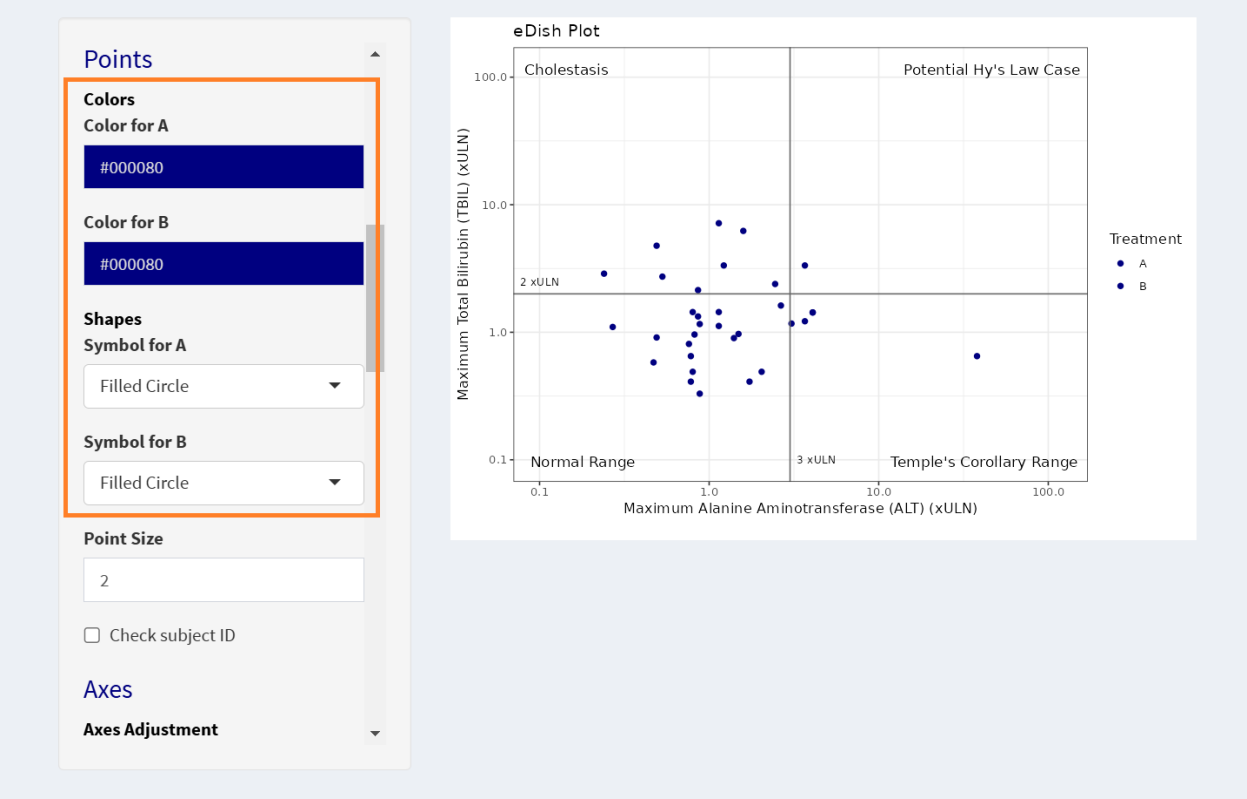

Figure 5. Color and Shape Selection Widgets Aligned to the Reordered Legend

Figure 5 shows the color and shape selection widgets, arranged according to the determined legend order, with subgroup A appearing before subgroup B. In this figure, all widgets automatically follow this order, ensuring that A remains before B as users customize colors and shapes.

5. Render the Customized Plot (Server Component)

After integrating user-defined color and shape values into ggplot2, the final step is to render the customized eDish plot using `renderPlot()`.

```
output$edish_plot <- renderPlot({
  edish_plot <- generate_edish_plot()
  print(edish_plot)
})
```

Here, `generate_edish_plot()` creates the ggplot object, and `print(edish_plot)` ensures the plot is displayed in the Shiny app. This completes the process, enabling users to dynamically visualize customized subgroup colors and shapes in the eDish plot.

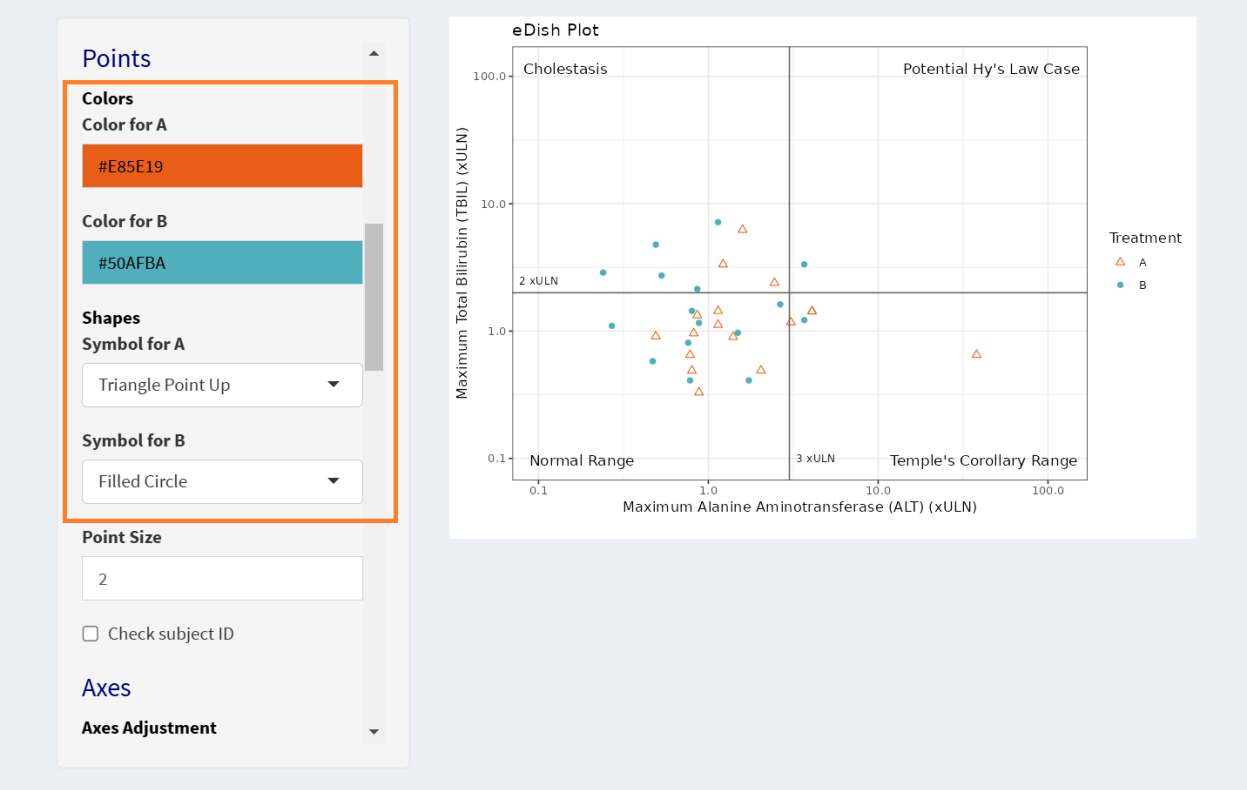

Figure 6. Final eDish Plot with User-Selected Colors and Shapes Aligned to the Reordered Legend

After selecting the colors and shapes in Figure 5, the final customized eDish plot is generated and displayed. Figure 6 presents the completed output after users have chosen their preferred style options for each subgroup. The rendered plot fully incorporates these selections while maintaining the predefined legend order, providing a clear and structured visualization of the subgroups.

ADDITIONAL FIGURE APPLICATION EXAMPLES

Besides eDish plot, this integrated solution can be widely applied in various types of figures, such as Kaplan-Meier (KM) plot and swimmer plot.

1. Kaplan-Meier (KM) Plot

Figures 7 and 8 demonstrate its use in adjusting subgroup order and customizing colors within a KM plot. Figure 7 shows the initial KM plot output upon data import, where the default legend order places male before female, and no modifications have been made. Figure 8 presents the updated version, where the legend order is adjusted to display female before male, and custom colors are applied to each subgroup.

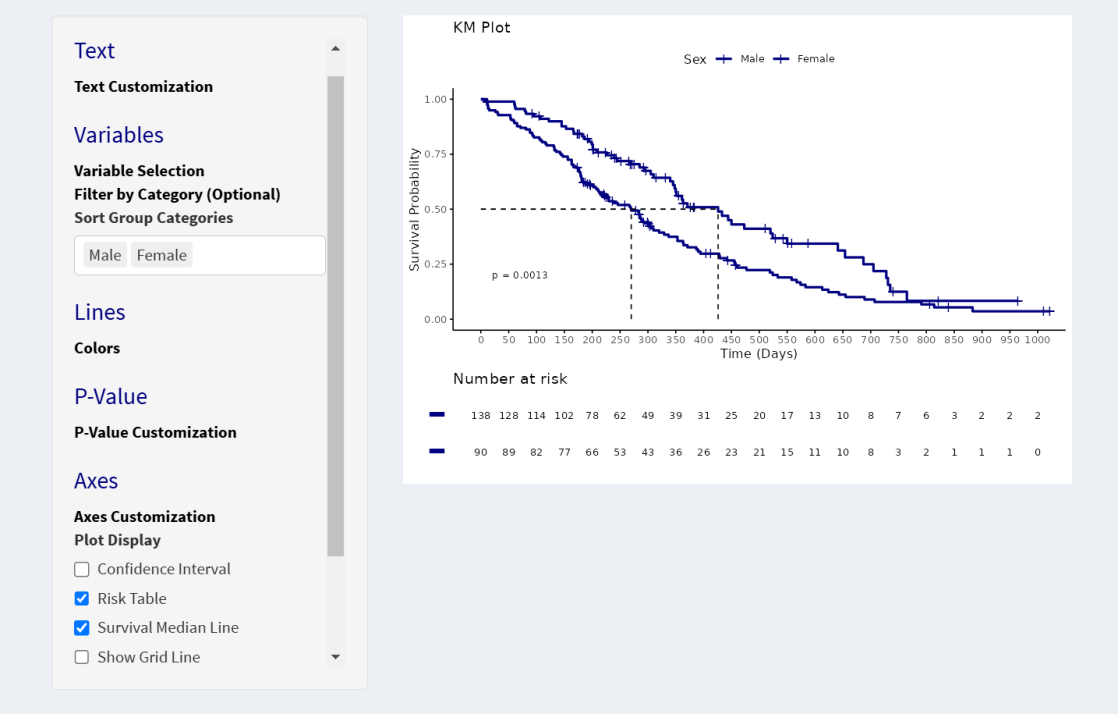

Figure 7. Kaplan-Meier Plot with Default Legend Order (Male Before Female)

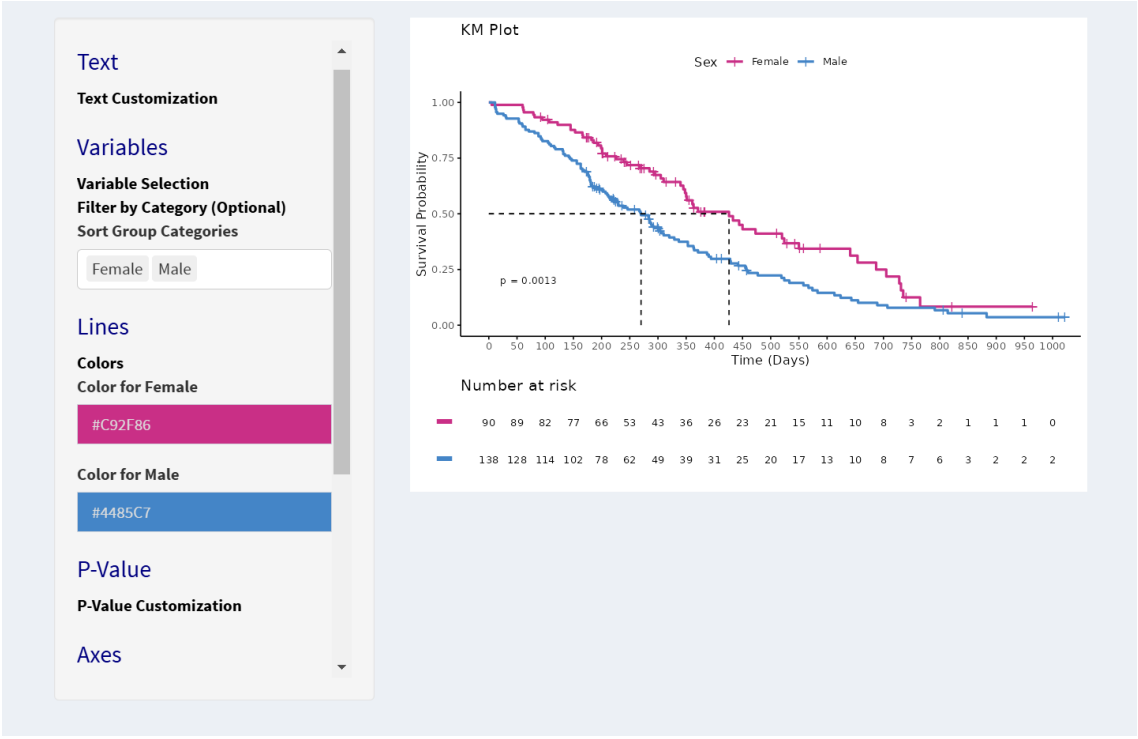

Figure 8. Kaplan-Meier Plot with Reordered Legend (Female Before Male) and Custom Colors

2. Swimmer Plot

Figures 9 and 10 demonstrate the implementation of the integrated solution in a swimmer plot example.

Figure 9 shows the initial figure output after data import. At this stage, the treatment groups and response categories follow the original order from the dataset, and no visual customization has been applied.
Figure 10 presents the finalized version of the swimmer plot after adjusting the order of treatment groups and response categories, and assigning corresponding colors and shapes. Bar colors represent treatment groups, while response types are distinguished using both color and shape.

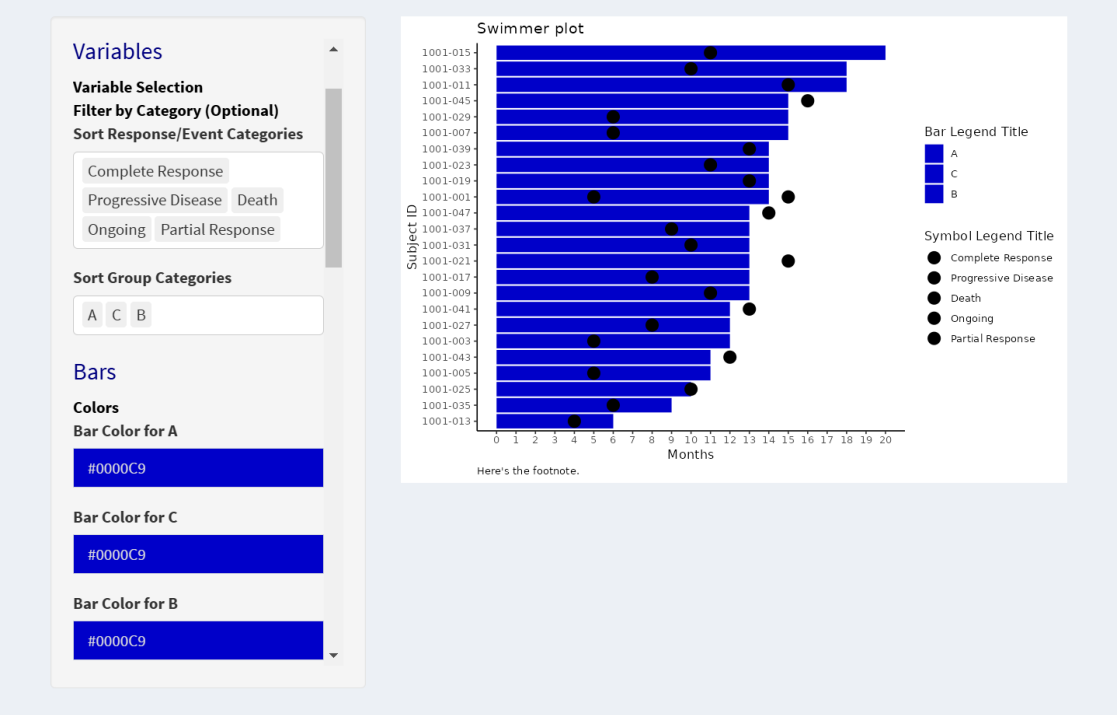

Figure 9. Swimmer Plot with Default Legend Order and No Customization

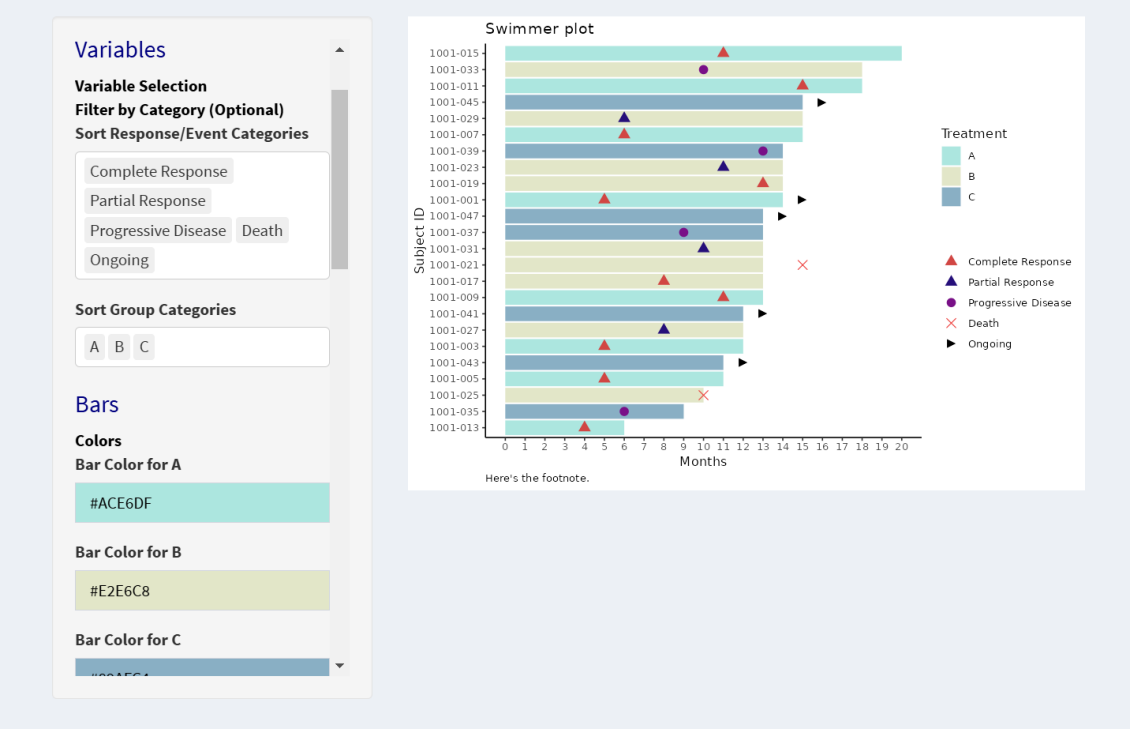

Figure 10. Finalized Swimmer Plot with Reordered Treatment and Response Groups with Custom Colors and Shapes

CONCLUSION

This paper presents an interactive, dynamic solution for figure generation using R Shiny, addressing key limitations in traditional SAS macros and R functions for visualization. By integrating automatic subgroup detection, interactive legend reordering, and customizable color selection, which is further extended to shape selection, our approach eliminates the need for manual parameter adjustments and ensures that visualizations stay in sync with evolving datasets.

This solution has been successfully applied to all analysis figures requiring subgroup classifications, such as Kaplan-Meier (KM) plot, swimmer plot, waterfall plot, and spider plot. Users can seamlessly switch grouping variables or reorder subgroups, with color pickers, shape selectors, and other style options dynamically adjusting to reflect the latest order. This makes it a highly flexible and efficient approach for clinical trial reporting, where frequent data updates require rapid figure generation.

A key consideration in this approach is ensuring the correct order of subgroup customization steps. Legend reordering should be applied first, followed by color picker and other style options like shape selection, ensuring consistency across all subgroup customizations. Since color and shape are parallel attributes, their processing order in the server component does not affect the final output.

Furthermore, this framework can be easily extended by incorporating an additional filtering step before selecting the grouping variable to define the analysis population. This ensures that the subsequent subgroup customizations, including legend reordering and style selections, are applied to the appropriate dataset. While this filtering capability was implemented, it was not covered in detail in this paper.

Additionally, this framework allows for further customization in widget design to enhance user interaction and flexibility. For example, beyond `selectInput()`, drag-and-drop functionality using `rank_list()` from the `{sortable}` package offers a flexible and intuitive way for users to reorder subgroups interactively.

Overall, this integrated R Shiny solution improves efficiency and flexibility in clinical trial reporting by minimizing manual adjustments and ensuring visualizations remain responsive to evolving datasets. Its adaptable framework can be easily incorporated into various analysis processes, offering a scalable alternative to traditional static methods. As clinical data visualization continues to advance, dynamic tools such as R Shiny will play a key role in automating workflows, enhancing reproducibility, and enabling real-time data exploration.

REFERENCES

Dean Attali. "colourpicker: A Colour Picker Widget for Shiny Apps, RStudio, R-Markdown, and 'htmlwidgets'." Dean Attali's Blog. https://deanattali.com/blog/colourpicker-package/ (accessed March 20, 2025).

Engineering Production-Grade Shiny Apps. "Chapter 4: Introduction to {golem}." https://engineering-shiny.org/golem.html (accessed March 20, 2025).

de Vries, A., Schloerke, B., & Russell, K. (2024). sortable: Drag-and-Drop in 'shiny' Apps with 'SortableJS'. R package version 0.5.0.9000. Retrieved from https://rstudio.github.io/sortable/

ACKNOWLEDGEMENTS

This paper was independently developed by the author, drawing inspiration from a broader collaborative project conducted at Pfizer. I am grateful to Kuldeep Sen for his ongoing support throughout this work, and to Shang-Ying Liang and Qiuhong Jia for their input on visualization needs. I also thank colleagues who provided input or support during the process, and acknowledge Pfizer for providing the environment and resources that enabled the development of this solution.

CONTACT INFORMATION

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

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