PharmaSUG 2025 - Paper DV - 244

Shining a Light on Adverse Event Monitoring with R Shiny

Abigail Zysk and Joe Lorenz, PPD, part of Thermo Fisher Scientific

ABSTRACT

In pharmacovigilance (PVG), timely and precise reporting of serious adverse events (SAEs) is vital for patient safety and regulatory compliance. To enhance the display and usability of SAE data, we developed an R Shiny dashboard for the automated reporting of SAEs. R Shiny offers several advantages over traditional static reporting for dynamic and interactive data visualization and reporting. Key features of the R Shiny dashboard include:

1. Real-time Data Integration: Seamlessly extract and process SAE data from existing SAS programs.
2. Interactive Visualization: Dynamic charts and graphs provide immediate insights into SAE trends.
3. Customizable Alerts: Configurable notification settings to alert users of specific SAE criteria.
4. Automated Reporting: Scheduled and on-demand report generation with automated email delivery.
5. User-Friendly Interface: An intuitive interface that accommodates users with varying levels of technical expertise.

We discuss the technical architecture, including the integration of R packages for data manipulation, visualization, and email automation. Additionally, we share insights from the deployment process, highlighting challenges and customization solutions to ensure robust performance.

The implementation of an R Shiny dashboard represents a significant enhancement in SAE reporting, offering a dynamic and efficient solution for PVG teams. By improving the accessibility and quality of SAE reports, this tool supports better decision-making which ultimately contributes to improved patient safety. This paper aims to provide valuable insights for statistical programmers, data scientists, and PVG professionals interested in leveraging R Shiny to develop automated reporting solutions in the pharmaceutical industry.

INTRODUCTION

Monitoring Adverse Events (AEs) is crucial in pharmacovigilance (PVG) to ensure the safety and efficacy of pharmaceutical products. Accurate and prompt reporting of Serious Adverse Events (SAEs) is essential for ensuring patient safety and adhering to regulatory standards. Adherence to stringent reporting timelines, as emphasized in regulatory guidelines, is essential to avoid penalties and ensure that appropriate measures are taken to address potential risks associated with drug therapies.

A Suspected Unexpected Serious Adverse Reaction (SUSAR) is any SAE that is unexpected, meaning it is not consistent with the applicable product information. These types of SAEs could significantly impact a drug's safety profile, and therefore, strict regulatory timelines are enforced to ensure compliance. Non-compliance can lead to penalties and large fines, emphasizing the importance of urgent reporting to protect subjects adequately.

To address these challenges and enhance the efficiency of SAE reporting, we developed an R Shiny dashboard for the automated reporting of SAEs. This tool aims to streamline the identification and reporting process, ensuring timely compliance with regulatory requirements and improving overall drug safety monitoring.

R SHINY BASICS

R Shiny is a web application framework for R that enables users to turn their data and analyses into interactive web applications. With Shiny it becomes more straightforward to build dynamic data visualizations and perform real-time analysis. Shiny leverages R's package ecosystem for seamless

prototyping and deployment. Its interactivity, with features like sliders and drop-down menus, allows for instant data filtering and updates. Shiny is highly customizable, allowing applications to be tailored to specific client needs, making it ideal for generating personalized PVG SAE reports and empowering users to explore data and gain insights quickly.

A Shiny app typically consists of two main components:

1. UI (User Interface) Definition: This part of the app defines the layout and appearance of the user interface.
2. Server Function: This part contains the bones of your app including the data, tables, or the definition of visualizations, etc.

Here is a basic example of an R Shiny app that produces an interactive histogram using randomly generated data. This is inspired by the basics part of a shiny app, however this example also incorporates ggplot2. The first step is to load the necessary libraries:

- shiny: This library is essential for creating Shiny apps and accessing all its functions.
- ggplot2: This library is used for creating visualizations, including the geom_histogram() function needed for our histogram.

Now that we have the necessary libraries, we will define our UI. The UI consists of a title panel, a sidebar with a slider input to select the number of observations, and a main panel to display the plot. The code below is heavily commented on to explain the use of each function.

```
# UI Definition
ui <- fluidPage(
  # Title of the app
  titlePanel("Interactive ggplot2 Graph in Shiny"),

  # Layout with a sidebar and main panel
  sidebarLayout(
    # Sidebar panel for inputs
    sidebarPanel(
      # Slider input for number of observations
      sliderInput("num",
                  "Number of observations:",
                  min = 10, max = 100, value = 50)
    ),

    # Main panel for displaying outputs
    mainPanel(
      # Output for the plot
      plotOutput("plot")
    )
  )
)
```

After defining our UI, we move on to the Server function, where the core functionality of the app is built. This is where you would create or read in your data, generate plots, compute statistics, create frequency tables, etc. In this example, we create a random data set and a histogram that uses this data. Lastly, the shinyApp() function is used to create our app.

```
# Server function
server <- function(input, output) {
  # Render plot based on input from UI
  output$plot <- renderPlot({
    # Generate fake data based on the number of observations selected
    data <- data.frame(x = rnorm(input$num))

    # Create a ggplot histogram with the generated data
    ggplot(data, aes(x = x)) +
      geom_histogram(binwidth = 0.5, fill = "blue", color = "white") +
      labs(title = "Histogram of Fake Data", x = "Value", y = "Frequency") +
      theme_minimal()
```

```
})
}
# Combine UI and server to create the Shiny app
shinyApp(ui = ui, server = server)
```

Here is the app that is created after running the above code. In Figure 1, we have a histogram with the slider set to 28, so only 28 observations are used in constructing the histogram. In Figure 2, we use 100 observations to construct the histogram, which illustrates the classic Law of Large Numbers.

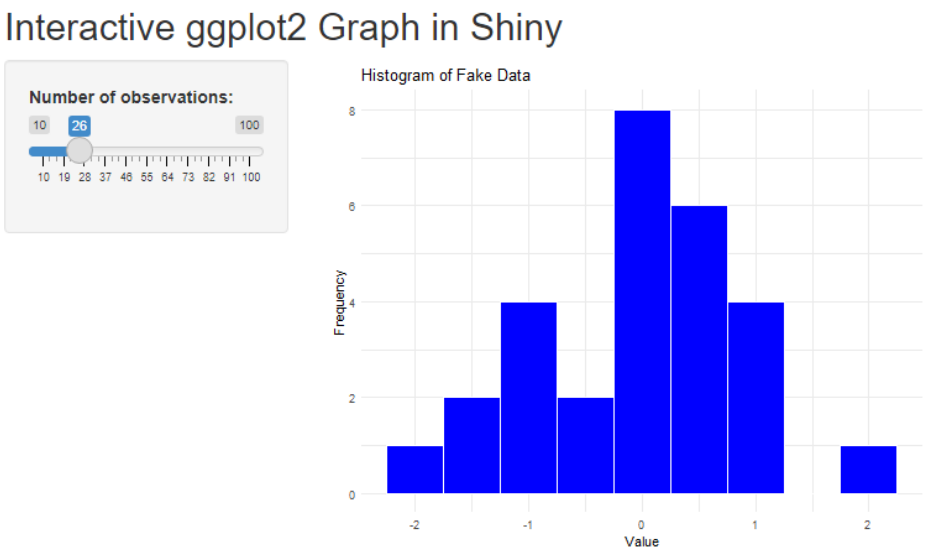

Figure 1. Shiny App showing interactive Histogram with slider set to 26

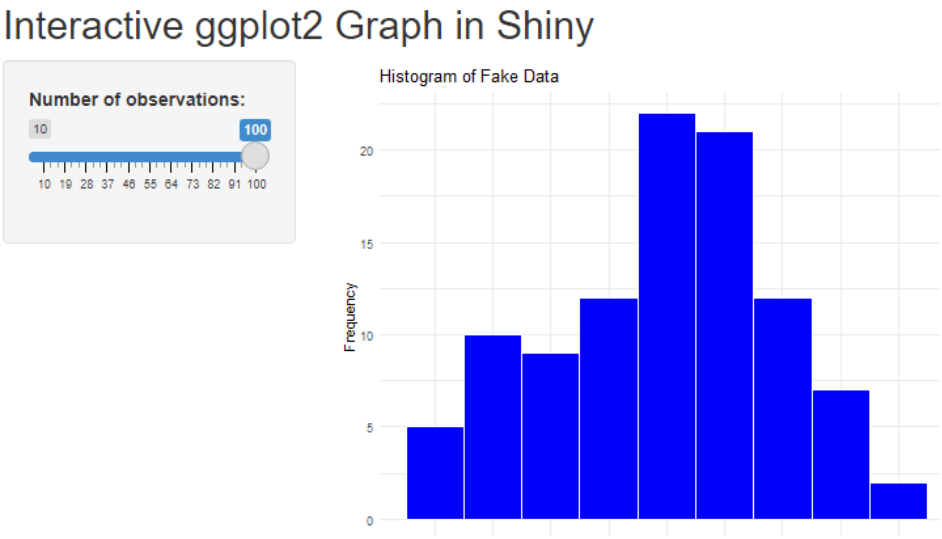

Figure 2. Shiny App showing interactive Histogram with slider set to 100

FEATURES AND FUNCTIONALITY

We use several libraries in our Shiny app to leverage their functionalities:

- shiny: This is the core library required to create Shiny apps and access all its functions.
- dplyr: This library is essential for data manipulation and transformation. It provides functions like mutate, filter, and count that help modify and filter datasets efficiently.

- DT: This library is used for creating interactive data tables within the Shiny app, enhancing data presentation and user interaction.
- ggplot2: This library is utilized for creating visualizations. In our app, we use it to generate a boxplot and a barchart.
- haven: This library is used for reading in SAS datasets.

MULTILINGUAL APPROACH

The foundation of a PVG report includes extracting data, scheduled runs to ensure daily generation, and automated notifications to confirm correct execution. Traditionally, at PPD, we generate the PVG SAE report in SAS, producing PDFs of individual AEs.

To leverage our existing infrastructure, we continued using the data cleaning process in SAS but added a step to output a dataset for further processing in R. This facilitated the development of the R Shiny app. For simplicity, the SAS program was executed through R. Below is the code to execute a SAS program within R:

```
sas_executable <- "C:/Program Files/SASHome/SASFoundation/9.4/sas.exe"
sas_program <-        "C:/path/to/your/my_sas_program.sas"

command <- paste(sas_executable, sas_program)
```

Please note that depending on how your SAS is installed on your system this method may not work for you and may need to be modified to fit your installation. In cases where you may not be able to access your SAS executable file, and therefore not integrate your daily extraction we suggest looking into packages like the taskscheduleR, to help schedule your R to run after your scheduled SAS extract.

FILTERING

One of Shiny's powerful features is the ability to dynamically filter data based on user inputs. The filtering process in Shiny applications is typically driven by input controls such as select inputs, sliders, and text inputs that users manipulate to specify their filtering criteria. For example, in Figures 1 and 2, in our basic shiny example, we saw how bar charts could have a slider to filter the number of data points included.

In a Shiny application, the server logic contains reactive expressions that respond to changes in these input controls. When a user modifies an input, such as selecting a different patient, the reactive expressions automatically re-evaluate, updating the filtered dataset and subsequently, the visualizations and tables that depend on this data. This ensures that the user always sees the most relevant subset of data based on their specified criteria.

A crucial piece of code for implementing this functionality involves the use of reactive expressions and the observe function. For instance, consider the following code that filters a dataset based on multiple user inputs:

```
filtered_df <- reactive({
  data <- df()
  if (input$patient_id != "All Patients") {
    data <- data %>% filter(PatientID == input$patient_id)
  }
  if (!is.null(input$aes_filter) && length(input$aes_filter) > 0) {
    data <- data %>% filter(AESI %in% input$aes_filter)
  }
  if (!is.null(input$seriousAE_filter) && length(input$seriousAE_filter) > 0) {
    data <- data %>% filter(SeriousAE %in% input$seriousAE_filter)
  }
  if (!is.null(input$aerel_filter) && length(input$aerel_filter) > 0) {
    data <- data %>% filter(AEREL %in% input$aerel_filter)
  }
  if (!is.null(input$caused_discontinuation_filter) &&
length(input$caused_discontinuation_filter) > 0) {
    data <- data %>% filter(CausedDiscontinuation %in% input$caused_discontinuation_filter)
  }
```

This code defines a reactive object filtered_df that filters the dataset based on user selections from various input controls. Each if statement checks whether the user has specified a filter and applies it to the dataset using the dplyr package's filter function.

Moreover, Shiny's observe function can be used to update the choices available in input controls based on the current state of the data. For example:

```
observe({

  updateSelectInput(session, "aes_filter", choices = unique(filtered_df()$AESI), selected = input$aes_filter)

  updateSelectInput(session, "seriousAE_filter", choices = unique(filtered_df()$SeriousAE), selected = input$seriousAE_filter)

  updateSelectInput(session, "aerel_filter", choices = unique(filtered_df()$AEREL), selected = input$aerel_filter)

  updateSelectInput(session, "caused_discontinuation_filter", choices = unique(filtered_df()$CausedDiscontinuation), selected = input$caused_discontinuation_filter)

  updateSelectInput(session, "patient_id", choices = c("All Patients", unique(filtered_df()$PatientID)), selected = input$patient_id)

}) shinyApp(ui = ui, server = server)
```

This observe block dynamically updates the choices available in the select inputs based on the filtered dataset. This ensures that the user is always presented with relevant options, enhancing the interactivity and user experience of the application. In Figure 3 below we show an example of our listing review with no filters selected, and in Figure 4 how it filters when PatientID is set to TFPPD001.

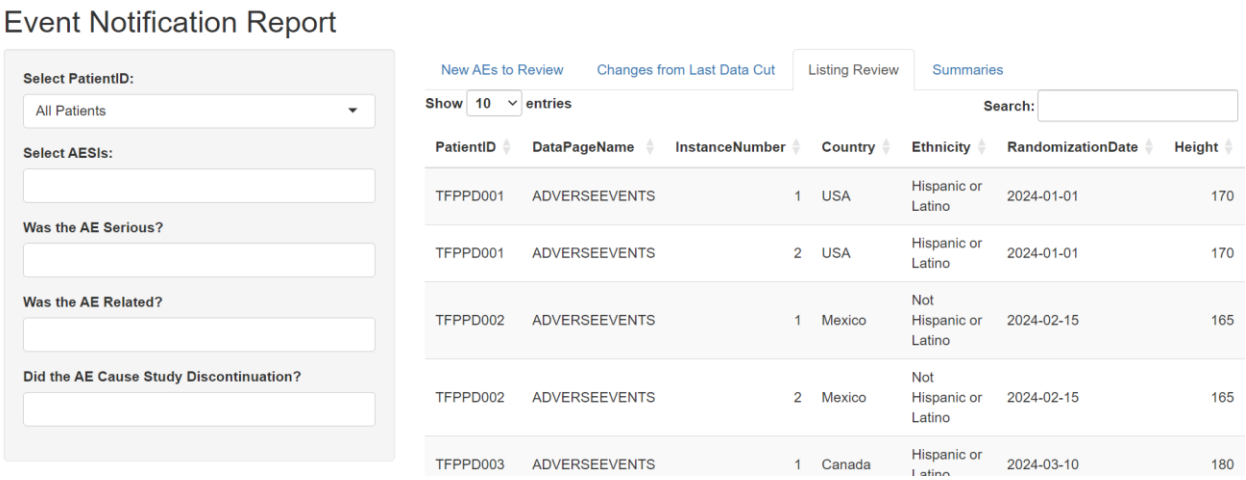

Figure 3. Shiny Dashboard with no filter selected

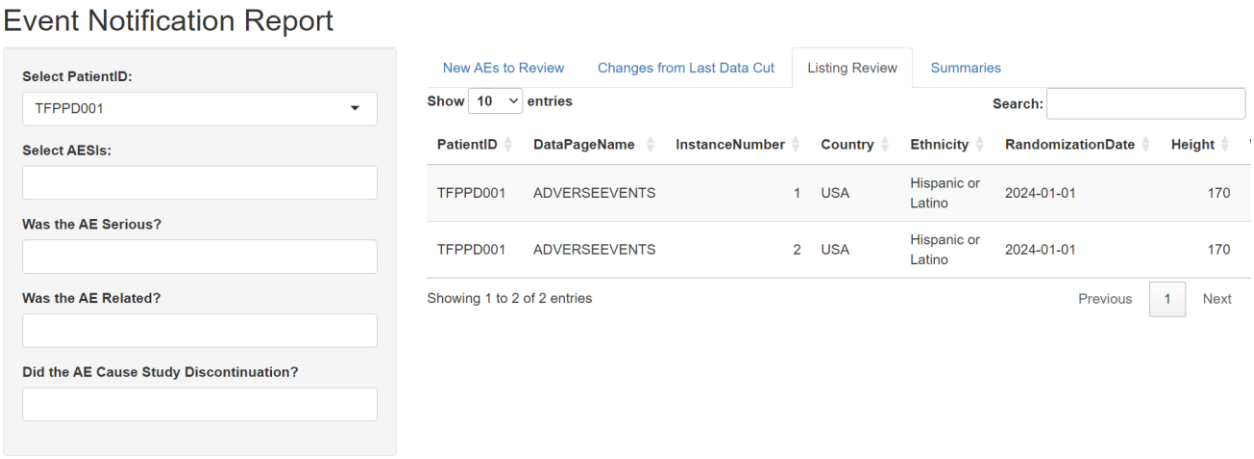

Figure 4. Shiny Dashboard with PatientID filtered to TFPPD001

Filtering Visualizations

In Shiny, visualizations are typically rendered using reactive objects. By utilizing the reactive object filtered_df as our dataset for creating visualizations, we can dynamically filter and transform the data based on user inputs. In the code below we can see how filtered_df well developing our visualizations.

```
# Plot output for the boxplot of weight by AESI
output$weight_aesi_barchart <- renderPlot({
  ggplot(filtered_df(), aes(x = AESI, y = Weight)) +
    geom_boxplot() +
    geom_jitter(width = 0.2, alpha = 0.5) +
    labs(title = "Boxplot with Jittered Points of Weight by AESI", x = "AESI", y =
"Weight")
})

# Plot output for the bar chart of AEs over time
output$aes_overtime_barchart <- renderPlot({
  filtered_df() %>%
    mutate(DaysDifference = as.numeric(difftime(StartDate, RandomizationDate, units =
"days"))) %>%
    count(DaysDifference) %>%
    ggplot(aes(x = DaysDifference, y = n)) +
    geom_bar(stat = "identity") +
    labs(title = "Number of AEs Over Time", x = "Days Between Start Date and Randomization
Date", y = "Number of AEs")
})
```

Program 1. Code for visualizations

Below, in Figures 5 and 6, we illustrate the application of data filtering to the boxplots. In Figure 5, all patients are displayed. In contrast, Figure 6 shows the boxplot filtered based on the selection of "Related AEs" in the side panel, displaying data only from subjects with related adverse events (AEs). As seen, subjects with related AEs exclusively had AESIs.

Figure 5. Boxplots (Weight by AESI) showing All Patients

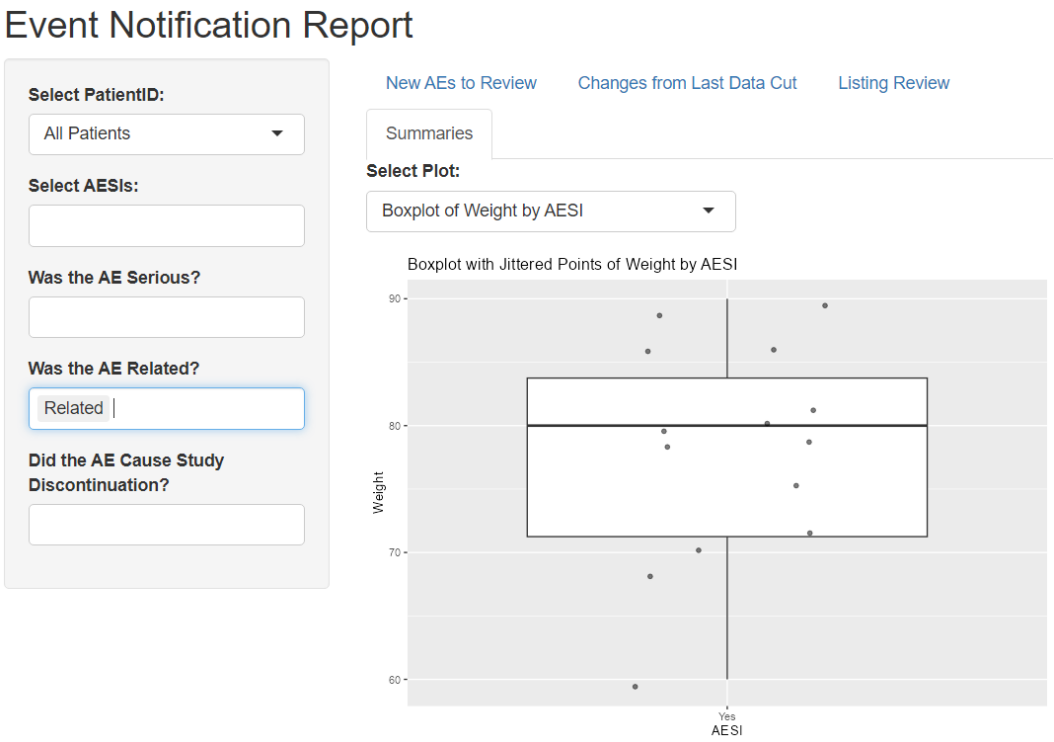

Figure 6. Boxplots (Weight by AESI) showing All Patients Who had Related AEs

RENDERING

In Shiny, the renderPlot function is used to generate and display plots that update based on selections within the user interface. This function is part of Shiny's suite of render functions, which also includes renderTable, renderText, renderUI, and others. The renderPlot function takes a plotting expression as its argument, such as ggplot2, lattice, or other R graphics. The function evaluates this expression and renders the resulting plot in the specified UI output element. In Program 1 above, you can see how the renderPlot function takes the ggplot as an argument. This allowed us to display two graphs in the same spot on our app, the user can switch between the graphs by selecting the desired visual in a drop-down menu, which can be seen in Figures 7 and 8.

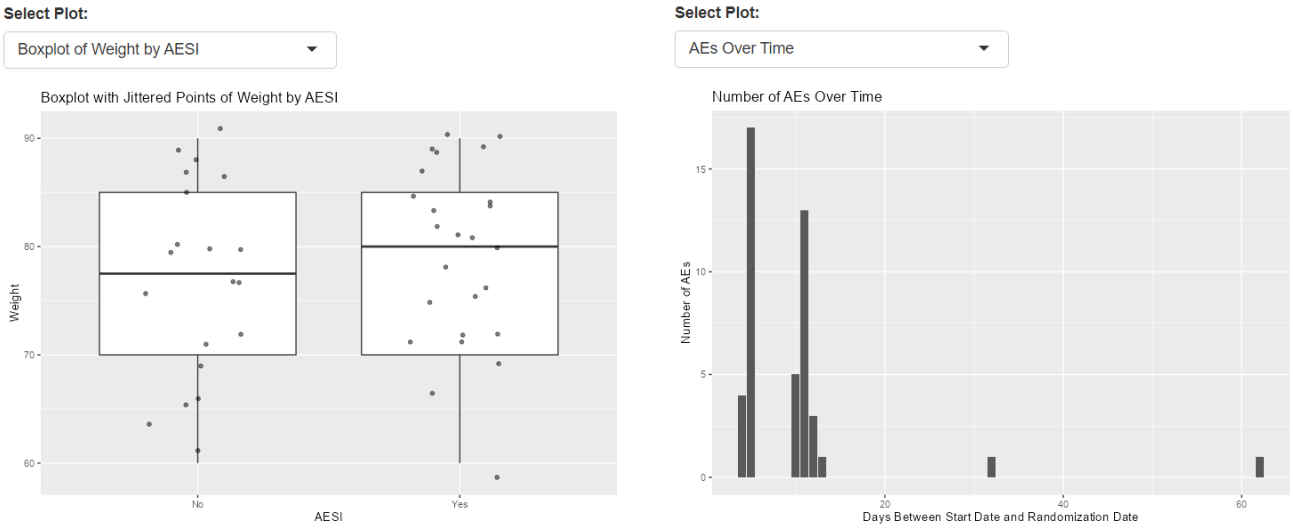

Figure 7 and Figure 8. Visualizations with Drop down menu

CUSTOMIZABLE ALERTS

Auto notifications and email alerts are essential tools for keeping users and administrators informed about important events and issues in real time. This is a crucial feature of any PVG SAE report, to ensure that clients are receiving accurate reports to keep track of AEs.

In practice, auto notifications and email alerts are implemented by setting up triggers based on specific conditions within the application. When these conditions are met, the system automatically sends notifications or emails to the relevant stakeholders. Tools like the blastula package in R make it easy to compose and send customized emails programmatically, ensuring that important messages are delivered promptly and effectively. The code below attempts to read a CSV file from the specified file_path using read.csv(). If an error occurs during this process, it catches the error, displays a notification in the Shiny app, and sends an email alert with the error details to the specified recipient using the blastula package. If an error occurs, the function returns NULL.

```
# df <- tryCatch({
  read.csv(file_path)
}, error = function(e) {
  showNotification("Error reading the CSV file", type = "error")

  # Send error email
  email <- compose_email(
    body = md(paste("An error occurred while reading the CSV file:", e$message))
  )

  smtp_send(
    email = email,
    from = "abigail.zysk@ppd.com",
    to = "abigail.zysk@ppd.com",
```

```
    subject = "Shiny App Error Notification",
    credentials = creds(
      user = "abigail.zysk@ppd.com",
      provider = "outlook",
      use_ssl = TRUE
    )
  )

  return(NULL)
})
```

CONCLUSIONS AND INSIGHTS

SHARING SHINY

When developing Shiny apps that utilize clinical data, ensuring the ability to share these apps with users securely is a paramount consideration. There are several methods available to host and securely share Shiny apps. Typically, the responsibility for determining the best sharing method may fall outside the purview of a programmer or statistician.

For those just starting out, Shinyapps.io by Posit is an simple platform to explore for hosting test Shiny apps. It offers a user-friendly interface and straightforward deployment process, making it accessible even to those with limited technical expertise. However, there are many downsides for making this a scalable solution.

Fortunately, there are numerous alternatives and resources available to support you in finding the most suitable hosting method for your needs. It's advisable to seek out additional information and consult with IT professionals or cloud service experts to ensure that your Shiny apps are both secure and scalable.

CONCLUSION

The development of an R Shiny dashboard for automated reporting of Serious Adverse Events (SAEs) is a notable progress in pharmacovigilance monitoring. This tool enhances the efficiency and accuracy of SAE reporting, offering dynamic and interactive data visualization that surpasses traditional methods like static reporting. With real-time data integration, customizable alerts, and automated reporting, PVG teams can swiftly respond to emerging safety concerns, thereby improving patient safety and regulatory compliance.

The intuitive design of the R Shiny dashboard ensures accessibility for users with varying technical expertise, allowing all PVG team members to utilize the tool effectively. Automated reporting and email notifications streamline the process, reducing the risk of human error and ensuring timely communication of critical information.

In this paper, we demonstrated a simple example of how R Shiny can be utilized to streamline the review of adverse event (AE) data, making the process more efficient and user-friendly. However, the capabilities of R Shiny extend far beyond this initial example. Numerous customizations can be implemented to further tailor the functionality to the specific needs of your team. For instance, features such as an automated data extraction refresh button.

The potential applications of R Shiny are not confined to Adverse Event data review alone. We envision that a similar approach could significantly enhance Data Safety and Monitoring Board (DSMB) meetings by allowing reviewers to customize their outputs according to their specific inquiries. Moreover, R provides the tools necessary to create a multilingual programming approach, facilitating the integration of existing SAS frameworks while adding greater versatility and functionality.

In conclusion, while our example illustrates just a fraction of what is possible with R Shiny, the framework offers a powerful and flexible solution for a wide range of data review and reporting needs in clinical research and beyond.

RECOMMENDED READING

Posit. (2017, June 28). Getting Started with Shiny [Blog post]. Retrieved from https://shiny.posit.co/r/articles/start/basics/

DISCLAIMER

DISCLAIMER The content of this paper are the works of the author and do not necessarily represent the opinions, recommendations, or practices of PPD, part of ThermoFisher Scientific

ACKNOWLEGEMENTS

A special thanks to Martin Brown for reviewing this paper. Also, thank you to Amy Caison and Wesley Murray for encouraging us to submit to PharmSUG and being supportive managers.

CONTACT INFORMATION

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

Abigail Zysk
Abigail.Zysk@ppd.com

Joe Lorenz
Joe.Lorenz@ppd.com