Paper OS04 – PHUSE EU Connect 2024

Leveraging Shiny Modules for Modular and Scalable Web Applications for clinical trials/research

ABSTRACT

R Shiny offers a dynamic platform for developing interactive and customizable tools tailored to specific study needs but shiny modules revolutionize web application development by offering a modular and scalable approach to within the shiny framework. These encapsulated units of code enhance code reusability, maintainability, and collaboration among developers. Modules facilitate the division of complex applications into manageable components and enable easier integration of new features. This abstract presents an overview of Shiny modules, highlighting their benefits in promoting modularity, scalability, and code quality. By embracing modular programming practices in shiny development, developers can streamline their workflows, improve code organization, and deliver high quality applications that meet the needs of users and stakeholders effectively. In the context of drug discovery and clinical research, Researcher can leverage modular architecture to create customizable interfaces for exploring complex datasets, tracking clinical trial progress, and analyzing experiment results.

INTRODUCTION

Data has become the driving force of the 21st century, transforming business operations across industries. With vast digital data at hand and rapidly advancing technology, companies are increasingly seeking ways to harness the synergy between data and technology to maximize their benefits. While traditional, validated tools have long been the backbone of data analysis, there is a growing shift towards embracing open-source tools, which have been waiting for their opportunity to shine. These tools have the potential to unlock valuable insights hidden within the data.

One of the most evident advantages of open-source technologies is the significant cost savings from avoiding licensing fees. Additionally, they foster collaborative problem-solving through active open-source communities. Programmers also benefit from the flexibility to build applications using a wide array of packages and libraries that evolve continuously with community contributions.

As data volumes grow, automation and interactive visualization have become essential for effective analysis. Automation allows for efficient handling of large datasets, while visualization helps uncover patterns and outliers that traditional tabular methods may overlook.

This paper demonstrates how open-source technology frameworks can be leveraged to automate processes and create interactive visualizations, resulting in a versatile web application for clinical programming users. Such an application can enhance decision-making through improved visualization and increase efficiency through automated data handling.

IMPROTANCE OF WEB APPLICATION IN CLINCAL TRIALS

Implementing Shiny Dashboards for clinical trial data management offers more than just a technological improvement—it has a direct impact on the bottom line and overall business performance.

By streamlining data processes, organizations can make quicker, more informed decisions, enhancing both efficiency and market competitiveness.

a. **Enhanced Decision-Making**
   Shiny Dashboards provide stakeholders with real-time insights into clinical trial progress, patient outcomes, and potential risks, enabling faster and more informed decision-making. With timely access to critical data, teams can adjust strategies, allocate resources efficiently, and proactively manage risks. This allows businesses to refine trial protocols, identify promising candidates for further study, and expedite drug development timelines.

b. **Ensuring Regulatory Compliance**
   In the tightly regulated pharmaceutical industry, adherence to stringent standards is essential. Shiny Dashboards offer features like robust access controls, audit trails, and data validation, ensuring compliance with regulatory requirements such as those set by the FDA. By safeguarding data integrity and ensuring traceability, organizations can smoothly navigate regulatory processes, avoiding costly delays or

penalties related to non-compliance. For more insights, check out our blog post on R-based submissions to the FDA.

c. **Enhanced Visualization:** Access to a wide range of visualization options makes data analysis more insightful and accessible. Shiny Dashboards empower users to explore data visually, uncovering trends, patterns, and insights that might otherwise remain hidden. This enhanced visualization capability facilitates clearer communication of findings and facilitates more informed decision-making across all levels of the organization.

d. **Streamlined ETL Process:** The Extract, Transform, Load (ETL) process is an essential component in Shiny applications. This involves extracting relevant data, transforming it into a suitable format, and loading it into the application. Tools like **dplyr, tidyr, purrr from tidyverse, DBI package** are instrumental in this process, ensuring data is managed efficiently and is scalable.

R SHINY APP STRUCUTURE

Shiny applications are built around a simple but two parts structure: the User interface (UI) and the server interface. The UI is responsible for the app presentation and the Server is responsible for the app logic.

1. Overview of Shiny Application

A shiny module consists of two parts:

- **UI Element**: Defines the layout and inputs for the specific module.
- **Server Element**: Handles the logic and computation for that module

The UI is defined using Shiny's built-in functions that generate HTML elements, such **as fluidPage(), sidbarLayout(),mainPanel(),** etc. These functions enable developers to create well-structured web pages without writing HTML code directly.

The Server function defines the backend logic of the application, processing user inputs and producing outputs based on those inputs. It controls the interaction between the UI and the data or computations logic.

The following example shows the snip of the UI and Server code of Shiny applications.

```
library(shiny)

clinical_data <- data.frame(
  age = round(rnorm(100, mean = 50, sd = 10))
  )

ui <- fluidPage(
  titlePanel("Clinical Data Histogram"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 5,
                  max = 50,
                  value = 10)),
    mainPanel(
      plotOutput("ageHist")
    ))
)
```

- fluidPage(): Defines the overall layout of the app.
- titlePanel("Clinical Data Histogram"): Displays the app title at the top.
- sidebarLayout(): Organizes the layout into two sections: a sidebar and a main panel.
- sidebarPanel(): Contains a sliderInput where the user can choose the number of bins (between 5 and 50) for the histogram. The default value is set to 10.
- mainPanel(): Displays the output plot called "ageHist".

The server function processes the input from the slider and generates the histogram.

- renderPlot({...}): This creates the plot output.Inside the renderPlot,
- the hist() function generates a histogram. using the clinical_data$age column.
- input$bins: Sets the number of bins in the histogram based on the user's input from the slider.

```
server <- function(input, output) {

  output$ageHist <- renderPlot({
    hist(clinical_data$age,
         breaks = input$bins,
         col = 'darkgray',
         border = 'white',
         main = "Age Distribution",
         xlab = "Age")
  })
}
```

INTRODUCTION TO MODULAR SHINY APPLICATIONS

As shiny applications grow and become complex, maintaining a single script with UI and server logic can become tedious. To address this, Shiny introduces modules- a way to encapsulate small reusable components of the UI and server into separate functions. These modules help organize code, improve maintainability, encourage reusability and make it easier to scale application:

With shiny modules, you will be writing a combination of UI and server functions. Think of them as small, standalone shiny apps, which handle a fraction of your global application. If you develop R packages, chances are you have split your functions into series of smaller functions. With shiny modules, you are doing the exact same thing: with just a little bit of tweaking, you can split your application into a series of smaller applications.

A shiny module consists of two main parts:

1. **UI Module (mod_boxplot_ui):** This defines the user interface of the module. In the following example in the Module UI function, Create a function with the ID argument.
2. **Server Module (mod_boxplot_server):** This contains the server-side logic specific to that module. Call the shiny::moduleServer () and pass the id and a function that looks like a regular server function.

Note: Thanks to that id the shiny:moduleServer() will be able to bring the inputs and outputs created in the UI function

```
# Module for Boxplot
mod_boxplot_ui <- function(id) {

  ns <- NS(id)

  plotlyOutput(ns("boxplot"))
}
```

Purpose:
- mod_boxplot_ui function generates the UI for the boxplot.
- id is a unique identifier for the module instance.
- NS(id) creates a namespace function, ensuring that the input and output IDs inside this module don't conflict with other parts of the app.
- plotlyOutput(ns("boxplot")) is the placeholder for the Plotly boxplot.

```
mod_boxplot_server <- function(id, selected_data, adsl) {

  moduleServer(id, function(input, output, session) {
    output$boxplot <- renderPlotly({
      the_plot <- ggplot(data = adsl, aes(x = TRT01A,
                                          y = .data[[selected_data()]],
                                          fill = TRT01A)) +
        geom_boxplot() +
        theme_minimal() +
        theme(legend.position = "none",
              text = element_text(size = 15)) +
        labs(
          title = "ADSL Data",
          subtitle = "Comparing Treatment Groups",
          x = "",
          y = attributes(adsl[[selected_data()]])$label
        ) +
        scale_x_discrete(labels = label_wrap(10))

      ggplotly(the_plot)
    })
  })
}
```

Purpose:

- mod_boxplot_server function defines the server-side logic for the boxplot.
- It takes three arguments:
  - id: The unique identifier (for module registration).
  - selected_data: A reactive input that specifies which column from adsl to plot on the y-axis.
  - adsl: A dataframe containing clinical data, including treatment groups.

Key Components of the Server Logic:

1. moduleServer:
   - This function registers the server logic for the given id. It ensures the module is isolated and can be reused multiple times in the app.

2. renderPlotly:
   - Renders the boxplot as a Plotly object, making it interactive.

3. Creating the Plot (ggplot):
   - Uses adsl data with TRT01A (a column representing treatment groups) on the x-axis.
   - The y-axis shows a column selected reactively via selected_data().
   - Boxplot is created using geom_boxplot().
   - theme_minimal() and theme() are used for styling (like setting text size and hiding the legend).

4. labs():
   - Adds the plot title, subtitle, and axis labels.
   - The y-axis label is dynamically fetched using attributes(adsl[[selected_data()]])$label.

5. ggplotly():
   - Converts the ggplot object into an interactive Plotly graph.

WHEN TO USE SHINY MODULE

1. **Code Maintainability:** As Shiny applications grow larger, managing complex and lengthy logic becomes increasingly difficult. By organizing the app into modules, each functionality or feature is encapsulated and maintained independently. This not only leads to cleaner code but also simplifies collaborative development, as different developers can work on separate modules simultaneously. Furthermore, modularity makes debugging and testing easier, since issues can be isolated within specific modules without needing to sift through the entire codebase. For example, a data upload module can be maintained separately, allowing focused troubleshooting if issues arise with file imports.

2. **Code Reusability:** Modules are highly reusable across different sections of the same app or even in entirely different projects. This reduces duplication and promotes efficient development practices. For instance, a module that displays patients' demographics in a clinical trial can be reused for multiple studies, regardless of the specific dataset used. Similarly, input widgets (such as dynamic filters for data selection) can be reused in multiple dashboards. This not only saves development time but also ensures consistent functionality and design across applications.

3. **Scalability:** As applications grow in complexity, adding new features can introduce bugs or performance issues if not carefully managed. Modules enable incremental development, allowing new components to be added without disrupting existing features. For example, an application tracking clinical trial progress can begin with basic functionality (e.g., patient enrollment) and later expand with modules for adverse event reporting or treatment group analysis. This modular approach ensures that scaling the application remains smooth and manageable, reducing the likelihood of performance degradation. Additionally, this structure makes performance optimization easier, as bottlenecks can be identified and resolved within specific modules rather than the entire app.

4. **Separation of Concerns:** Shiny modules encourage a clear separation between UI and server logic. Each module manages its own UI elements and backend functionality, leading to better code organization. This makes it easier to understand and maintain individual parts of the app. For example, a visualization module might consist of a plot, associated inputs, and the logic to generate it. Developers can tweak the visualization independently without affecting other parts of the app.

5. **Simplifying User Management and Permissions**: In larger applications, especially those used in clinical

research or business operations, role-based access control (RBAC) becomes essential to restrict access to sensitive data and features. Shiny modules allow for granular control over user permissions, as each module can be designed to provide access only to specific user roles. For example, a data entry module could be restricted to trial coordinators, while a dashboard module showing high-level metrics is accessible to managers and stakeholders. This modular approach ensures the application is secure and compliant with privacy regulations, such as GDPR or HIPAA, by controlling who can access what data.

6. **Improving Testing and Modular Deployment:** Modular design also facilitates testing and validation of individual components. Each module can be unit tested in isolation, ensuring that the logic works as expected without needing to run the entire app. This is particularly useful when multiple developers are working on different parts of the application—testing modules separately reduces the likelihood of introducing bugs during integration. Additionally, continuous integration/continuous deployment (CI/CD) pipelines can be set up to deploy updated or new modules incrementally. This approach minimizes downtime and makes the app more resilient, as only the affected modules need to be redeployed, not the entire app.

7. **Facilitating Collaboration Across Teams:** With applications becoming increasingly complex, collaboration between teams of developers, data scientists, and domain experts is common. Shiny modules help break down the development workload into smaller, manageable components. Different teams can work on modules independently, with each module focusing on a specific feature or functionality. For example, the data preparation team can develop a module to clean and preprocess clinical trial data, while the analytics team creates a module to perform statistical analysis. This parallel development boosts productivity and ensures that different expertise areas are utilized efficiently.

8. **Enhancing Performance and Resource Management:** When dealing with large datasets or complex visualizations, resource management becomes critical to maintain application performance. Shiny modules allow for lazy loading of components, meaning that modules can be loaded only when required. For instance, a reporting module displaying patient outcomes can be loaded dynamically based on user input, rather than being preloaded when the app starts. This reduces the initial load time and improves the overall user experience by conserving memory and computational resources. It also ensures the application can handle large-scale data processing without compromising responsiveness.

9. **Improving Maintainability Through Version Control:** Using Shiny modules makes version control more effective, as each module can be tracked separately within a Git repository. Developers can focus on specific modules during updates or bug fixes, reducing the risk of conflicts when multiple contributors push changes. This modular approach also enables rollbacks, where individual modules can be reverted to a previous version if needed, without affecting the rest of the app. For instance, if a newly added data visualization module introduces issues, it can be temporarily removed or replaced with an earlier version without disrupting other parts of the application.

10. **Better User Experience Through Modular UI Design:** Shiny modules allow developers to design progressive interfaces, where additional UI elements appear based on user interaction. This modular UI design ensures that users are not overwhelmed by too many options at once. For example, a clinical trial app can begin by displaying patient enrollment forms, and based on the user's progress, dynamically load additional modules for adverse event reporting or data exports. This approach improves navigation and usability, leading to a more seamless user experience.

Namespace in Module.

Shiny modules are powerful because they create a "namespace" that isolates the controls within the modules from the rest of the app. In a typical shiny app, the IDs of controls are global, meaning all parts of the server can access all parts of the UI.

- All IDs in a shiny app need to be unique
- Use namespaces to make ID names unique for every module instance
- **ns <- NS(id)** generates a function that assigns a namespace for every module instance based on its id.

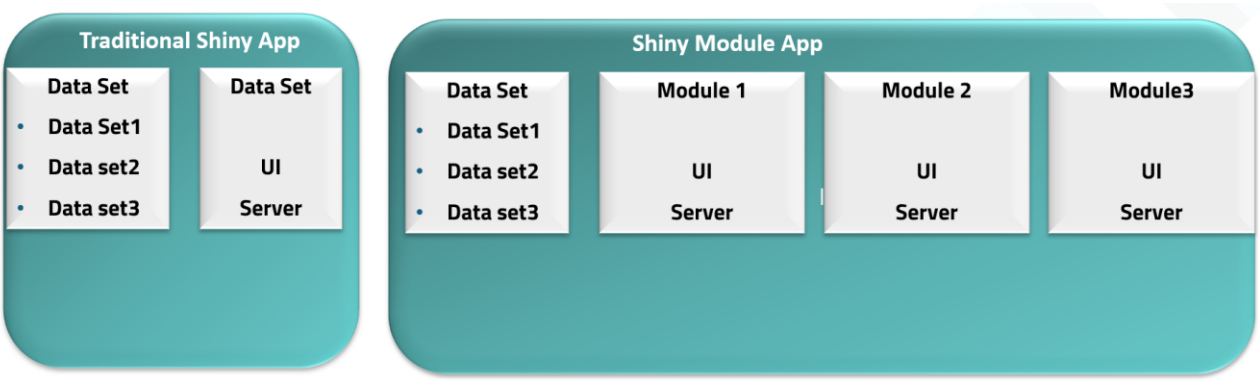
```
innerUI <- function(id) {
  ns <- NS(id)
  "This is the inner UI"
}

outerUI <- function(id) {
  ns <- NS(id)
  wellPanel(
    innerUI(ns("inner1"))
  )
}
```

Nesting modules
Modules can use other modules. When doing so, when the outer module's UI function calls the inner module's UI function, ensure that the id is wrapped in ns(). In the following example, when outerUI calls innerUI, notice that the id argument is ns("inner1"):

As for the module server functions, just ensure that the call to callModule for the inner module happens inside the outer module's server function. There's generally no need to use ns().

TRADITIONAL AND SHINY MODULE ARCHITECTURE



Traditional Shiny App (on the left):

In a traditional Shiny app, you typically have a single server function and a single UI function.
Data Set: The data used in the app is managed globally.
Data Set 1, Data Set 2, Data Set 3: You may have multiple datasets being processed or visualized in one place, without isolation between parts of the app.
UI and Server: All user interface components and server logic (reactivity, data handling, etc.) are managed in one place, which can make it hard to manage as the app grows larger or more complex.

Shiny Module App (on the right)

In a Shiny Module App, the architecture is broken down into modules, each with its own UI and server logic.

Module 1, Module 2, Module 3: Instead of having everything in one place, the app is split into smaller, reusable components (modules). Each module handles its own inputs, outputs, and server logic. This makes the code more modular and easier to maintain.
UI and Server per module: Each module has its own UI and server logic, which provides better isolation, avoiding conflicts between inputs and outputs in different parts of the app. Reactivity is scoped to the individual modules.

Data Set 1, Data Set 2, Data Set 3: The datasets might be shared across modules, but each module can interact with

the data independently.

Key Differences
Organization: The traditional app has all logic in one place, while the module-based app breaks the app into smaller, easier-to-manage pieces.

Reusability: Modules can be reused across different apps or different parts of the same app. Traditional apps don't have this flexibility without manual copying code.

Reactivity Isolation: In a traditional app, everything is part of the same reactive flow, which can become hard to manage as the app grows. Modules isolate reactivity, making the logic more manageable.

DIFFERENCE BETWEEN TRADITIONAL SHINY AND MOUDLE APPLICATIONS CODE

Modular app.R

```
# Load the modules
source("~/modules_final/mod_subject_input.R")
source("~/modules_final/mod_boxplot.R")
source("~/modules_final/mod_summary_stats.R")
source("~/modules_final/mod_data_table.R")

adsl <- read_xpt("~/adsl.xpt")

ui <- fluidPage(
  titlePanel("ADaM Subject-Level Analysis"),

  fluidRow(
    column(3,  # Sidebar for subject data selection
           mod_subject_input_ui("subject_data")
    ),

    column(9,  # Main area for the plot and tables
           mod_boxplot_ui("boxplot"),
           mod_summary_stats_ui("summary_stats"),
    )
  ))

server <- function(input, output, session) {

  # Call the input module
  selected_data <- mod_subject_input_server("subject_data")
  mod_boxplot_server("boxplot", selected_data, adsl)
  mod_summary_stats_server("summary_stats", selected_data, adsl)

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

Traditional app.R

```
adsl <- read_xpt("~/Phuse Test/adsl.xpt")

# User Interface ------------------------------
ui <- fluidPage(
  titlePanel("ADaM Subject-Level Analysis"),
  tabsetPanel(
    tabPanel("Analysis 1",
             column(3,
                    selectInput(inputId = "subject_data",
                      label = "Select Subject Data:",
                      choices = c("Age" = "AGE","Baseline BMI" = "BMIBL",
                        "Baseline Height" = "HEIGHTBL",
                        "Baseline Weight" = "WEIGHTBL",
                        "Years of Education" = "EDUCLVL"
                      ))),
             column(9, plotlyOutput("boxplot"),
                    tableOutput("summary_stats")#,
             ))
        ))

server <- function(input, output, session) {

  output$boxplot <- renderPlotly({
    the_plot <- ggplot(data = adsl, aes(x = TRT01A,
                                        y = .data[[input$subject_data]],
                                        fill = TRT01A)) +
      geom_boxplot() +
      theme_minimal() +
      theme(legend.position = "none",
            text = element_text(size = 15)) +
      labs(title = "ADSL Data",
        subtitle = "Comparing Treatment Groups",
        x = "",y = attributes(adsl[[input$subject_data]])$label
      ) + scale_x_discrete(labels = label_wrap(10))
    ggplotly(the_plot)
  })
  # Summary Statistics
  summary_data <- reactive({adsl %>%
      group_by(TRT01A) %>% summarize(
        Mean = mean(.data[[input$subject_data]], na.rm = TRUE),
        Median = median(.data[[input$subject_data]], na.rm = TRUE),
        SD = sd(.data[[input$subject_data]], na.rm = TRUE)
      )
  })
  output$summary_stats <- renderTable({
    summary_data()
  })
}
```

Purpose: The above source() function loads the individual module scripts. Each module is a self-contained function with its own UI and server logic.

- **fluidPage():** Creates a flexible layout for the app.
- **titlePanel():** Displays the application title at the top of the page.
- **fluidRow() and column():** Organize the layout into two columns:
- **Sidebar (Column 3):** Hosts the mod_subject_input_ui() to let users select subject-level data.

- **Main Area (Column 9)**: Contains the **mod_boxplot_ui()** for visualizing the selected data as a boxplot and **mod_summary_stats_ui()** for showing summary statistics.

This design ensures a responsive layout, with the input section on the left and outputs (boxplot, stats) on the right.

The server function defines the logic for the app's interactivity.

- **Input Module:** mod_subject_input_server() is invoked to handle the user's selection of data.
- **Plot Module:** mod_boxplot_server() generates an interactive boxplot using the user's selected data.
- **Summary Stats Module:** mod_summary_stats_server() calculates and displays summary statistics based on the selected data.

Each module is called with a unique identifier (ID) to ensure isolation and proper communication between the UI and server components.

Wherever on the traditional app.r file we have one ui function contains ui portions of whole app and one server function defines the log for whole app's interactivity

Data: The ADSL dataset is loaded from an XPT file, which is a standard format used in clinical trials.

MODULES BENEFITS

Modules are a kind of small app within a larger app that isolates a particular group of inputs and outputs with a UI and a server part.

- Avoid namespace collisions.
- Allows encapsulate distinct app components.
- Facilities collaboration
- Modules works as functions for Shiny.
    - Helps you break your big, complex app into smaller parts.
    - Makes Shiny code more readable.
    - Makes Shiny code easier to debug.
    - Makes Shiny components reusable.
    - Opens the possibility to apply unit test.

OPEN-SOURCE MODULARIZATION PACKAGE

Each of the following framework's help streamline the development of R Shiny applications, particularly when using modules for scalability and maintainability.

Here's an explanation of each in the context of Shiny modules:

1. golem

Purpose: golem is a framework designed to help structure and industrialize Shiny applications. It emphasizes application robustness by promoting good coding practices and reproducibility.

Shiny Modules: With golem, Shiny modules are a core part of the framework's structure. It encourages modularization to break the app into smaller, reusable components. This makes development scalable and

maintainable for larger apps.

Features: Built-in functions to create and manage modules (golem::add_module).
Focuses on testing, version control, and deployment, which are crucial for applications with several interconnected modules.

2. Rhino (by Appsilon)

Purpose: Rhino by Appsilon is another R Shiny framework that is built to support the development of enterprise-grade Shiny apps.

Shiny Modules: It integrates Shiny modules into its framework by encouraging clear separation of concerns, which makes the application easier to scale, test, and maintain. Modules in Rhino help build reusable and independent components.

Features: It offers support for integration with modern tools like unit tests, TypeScript, and CSS frameworks. Encourages writing modular and clean code, which makes app development more efficient.

3. **Teal**

Purpose: teal is a framework tailored for clinical reporting applications using Shiny. It helps organize complex workflows and create interactive data exploration tools.

Shiny Modules: The teal framework heavily relies on Shiny modules to allow developers to build flexible and interactive analysis tools. Each module in teal represents a specific analytical component that can be reused across different applications.

Features: Simplifies the development of clinical dashboards by using pre-built and custom Shiny modules. Focuses on enabling interactive data exploration and visualization, particularly for the pharmaceutical and clinical industries.

CONCLUSION

Using Shiny modules to develop clinical trial analysis applications not only makes the process simpler and more flexible but also greatly improves time efficiency compared to using standard Shiny development. By organizing the application into smaller, reusable modules, we reduce the overall development time since these modules can be quickly adapted or reused across different projects. This modular approach means developers don't have to rebuild features from scratch, speeding up the process significantly.

In clinical trials, where timely and accurate analysis is critical, these time and cost savings make a big difference. Shiny modules allow us to respond faster to changing requirements, provide quicker insights, and deliver more cost-effective solutions for researchers. This approach not only meets current needs but also helps us to handle future clinical data analysis challenges more efficiently.

REFERENCES

- Chapter 19 Shiny modules | Mastering Shiny (mastering-shiny.org)
- A beginner's guide to Shiny modules | R-bloggers
- Shiny - Modularizing Shiny app code (posit.co)

CONTACT INFORMATION

Contact the author at: **sohan.l225@geninvo.com**
Author Name: Sohan Lal
Company: Geninvo technologies pvt. ltd
Address: 1408 East Empire Street 61701
Bloomington (Illinois), United States.
Work Phone: +1 309-807-5006
Email: sohan.l225@geninvo.com
Website: GENINVO is the go-to partner for life science industry.

Contact the author at: **pinku.h161@geninvo.com**
Author Name: Pinku Hooda
Company: Geninvo technologies pvt. ltd
Address: 1408 East Empire Street 61701
Bloomington (Illinois), United States.
Work Phone: +1 309-807-5006
Email: pinku.h161@geninvo.com
Website: GENINVO is the go-to partner for life science industry.