Prototyping a Shiny App with ChatGPT

Ronald G. Thomas

2025-05-15

Figure 1: A napkin sketch of a simple app layout, a pen resting on top, beside a closed laptop: a rough idea sketched out before being built.

Iterative prototyping with conversational AI produces working Shiny applications faster than coding from scratch.

Introduction

I did not really know how effective ChatGPT could be as a prototyping partner until I sat down with a clear goal (build a modular Shiny app for Palmer Penguin data exploration) and tried to get there through iterative prompting. Three prompts and roughly twenty minutes later, I had a working application with boxplots, scatterplots, a correlation matrix, tooltips, and a modular architecture.

The key insight was not that ChatGPT writes perfect code on the first attempt (it does not), but that conversational iteration (starting broad and progressively adding specificity) mirrors how an experienced developer prototypes. Each prompt built on the previous output, refining the application in stages rather than attempting to specify everything upfront.

This post presents the three prompts I used, the complete Shiny application code that resulted, and a discussion of the modular architecture that ChatGPT produced. The code is presented as-is, with annotation explaining each component.

Motivations

- I wanted to explore how conversational AI performs at a practical prototyping task rather than generating toy examples.
- Shiny module architecture is notoriously difficult to get right, and I was curious whether ChatGPT could scaffold a modular app without extensive correction.
- The Palmer Penguins dataset is well-suited for interactive exploration and provides a concrete, familiar target for evaluating the generated code.
- I needed a working prototype quickly for a workshop and wanted to test whether AI assistance could compress the development timeline.
- I was skeptical of claims about AI-assisted coding and wanted first-hand experience to form an informed opinion.

Objectives

1. Build a complete, modular Shiny application for Palmer Penguin data exploration using iterative ChatGPT prompts.
2. Document the three-prompt strategy that progressively adds features: basic app, grouped boxplots, and scatterplots.
3. Present the complete application code with annotations explaining the module architecture, helper functions, and UI layout.
4. Evaluate the strengths and limitations of AI-assisted prototyping for Shiny development.

This learning process is documented here. Errors spotted or better approaches are always welcome.

Iterative prompting mirrors iterative design: start broad, refine progressively.

Prerequisites and Setup

To run the Shiny application produced in this post, one will need:

```
install.packages(
  c("shiny", "bslib", "bsicons",
    "palmerpenguins", "ggplot2",
    "corrgram", "dplyr")
)
```

Load the required libraries:

```
library(shiny)
library(bslib)
library(bsicons)
library(palmerpenguins)
library(ggplot2)
library(corrgram)
library(dplyr)
```

Background assumed: Familiarity with basic Shiny concepts (UI, server, reactive expressions) and the Palmer Penguins dataset. No prior experience with Shiny modules or bslib is required.

What is AI-Assisted Prototyping?

AI-assisted prototyping uses conversational AI tools like ChatGPT to generate initial code scaffolds that a developer then refines. Unlike traditional coding from scratch, the developer acts as a director rather than a typist: specifying what the application should do in natural language and then evaluating, correcting, and extending the generated output.

Think of it as pair programming with an AI partner that has broad but shallow knowledge. The AI produces syntactically correct code quickly, but the human provides domain expertise, architectural judgment, and quality standards. The combination tends to be faster than either approach alone, particularly for prototypes where speed matters more than perfection.

The three-prompt strategy demonstrated here reflects a general pattern: (1) establish the foundation, (2) add the primary feature, (3) extend with secondary features. Each prompt builds on the generated output rather than starting over.

Getting Started

The Three-Prompt Strategy

The iterative approach used three progressively specific prompts:

Prompt 1: Foundation:

> "I want to use the Palmer Penguin dataset to create a Shiny app for data exploration."

This produced a basic application with the dataset loaded, missing values removed, and a simple plotting interface.

Prompt 2: Primary feature:

> "Update shiny app. Add a dropdown menu to select categorical variables, sex, species or island. Also add a dropdown menu to select continuous variables. Use selected categorical variable as a grouping variable in side-by-side boxplots of selected continuous variables."

This added the interactive controls and the boxplot visualization that forms the core of the application.

Prompt 3: Secondary feature:

> "Add a second interactive plot to app.R code to provide scatterplots of 2 selected continuous variables. The two continuous vars are selected from drop down menus."

This added the scatterplot panel with independent axis selection and trend lines.

Launching the Application

Save the code below as `app.R` and launch from within R:

```
shiny::runApp("app.R", launch.browser = TRUE)
```

Or from the shell:

```
R -e "shiny::runApp('app.R', launch.browser=T)"
```

Figure 2: A speech-bubble-shaped paper cutout pinned above a sketch of an app screen on a corkboard: a conversational interface shaping a design

The complete application after three iterative prompts.

The Complete Application

The code below is the full `app.R` file produced through the three-prompt iteration. Annotations explain the architectural decisions that ChatGPT made.

Data Preparation and Helper Functions

```
library(shiny)
library(bslib)
library(bsicons)
library(palmerpenguins)
library(ggplot2)
library(corrgram)
library(dplyr)

data <- na.omit(penguins)

add_tooltip <- function(input_ui, tooltip_text) {
  input_ui$children[[1]] <- div(
    input_ui$children[[1]],
```

```
    span(
      bs_icon("info-circle-fill"),
      class = "tooltip-icon ms-2",
      `data-bs-toggle` = "tooltip",
      `data-bs-placement` = "right",
      title = tooltip_text
    ),
    style = paste0(
      "display: flex; ",
      "align-items: center;"
    )
  )
  input_ui
}
```

The `add_tooltip` function modifies a Shiny input widget to include a Bootstrap tooltip icon beside the label. This is a UI enhancement that ChatGPT generated unprompted (it was not part of any of the three prompts).

Input Module

```
inputsUI <- function(id) {
  ns <- NS(id)
  tagList(
    h4("Boxplot Controls"),
    add_tooltip(
      selectInput(
        ns("xvar"),
        "Continuous Variable for Boxplot:",
        choices = names(data)[3:6]
      ),
      paste0(
        "Select a continuous variable to ",
        "display on the Y-axis of the boxplot."
      )
    ),
    add_tooltip(
      selectInput(
        ns("groupvar"),
        "Group by (Categorical Variable):",
        choices = c("species", "sex", "island"),
        selected = "species"
      ),
      paste0(
        "Select a categorical variable to ",
```

```
        "group data in the boxplot."
      )
    ),
    hr(),
    h4("Scatterplot Controls"),
    add_tooltip(
      selectInput(
        ns("scatter_x"),
        "X-axis for Scatterplot:",
        choices = names(data)[3:6]
      ),
      paste0(
        "Select a variable for the X-axis ",
        "of the scatterplot."
      )
    ),
    add_tooltip(
      selectInput(
        ns("scatter_y"),
        "Y-axis for Scatterplot:",
        choices = names(data)[3:6]
      ),
      paste0(
        "Select a variable for the Y-axis ",
        "of the scatterplot."
      )
    ),
    add_tooltip(
      selectInput(
        ns("groupvar_scatter"),
        "Group by (Scatterplot):",
        choices = c("species", "sex", "island"),
        selected = "species"
      ),
      paste0(
        "Select a variable to group points ",
        "in the scatterplot by color."
      )
    )
  )
}

inputsServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    reactive(input)
  })
}
```

The input module consolidates all user controls into a single namespace. The `inputsServer` function returns a reactive reference to the entire input object, which other modules consume.

Boxplot Module

```
boxplotUI <- function(id) {
  ns <- NS(id)
  plotOutput(ns("boxPlot"), height = "400px")
}

boxplotServer <- function(id, data, inputs) {
  moduleServer(
    id, function(input, output, session) {
    output$boxPlot <- renderPlot({
      req(data(), inputs())
      ggplot(
        data(),
        aes(
          x = .data[[inputs()$groupvar]],
          y = .data[[inputs()$xvar]],
          fill = .data[[inputs()$groupvar]]
        )
      ) +
        geom_boxplot(alpha = 0.7) +
        theme_minimal() +
        labs(
          x = inputs()$groupvar,
          y = inputs()$xvar,
          fill = inputs()$groupvar
        ) +
        theme(legend.position = "bottom")
    })
  })
}
```

The boxplot module uses `.data[[]]` pronoun syntax for tidy evaluation, which is the correct approach for programmatic column selection in ggplot2. This is one area where ChatGPT produced idiomatic code without correction.

Scatterplot Module

```
scatterplotUI <- function(id) {
  ns <- NS(id)
  plotOutput(
```

```
    ns("scatterPlot"), height = "400px"
  )
}

scatterplotServer <- function(id, data, inputs) {
  moduleServer(
    id, function(input, output, session) {
    output$scatterPlot <- renderPlot({
      req(data(), inputs())
      ggplot(
        data(),
        aes(
          x = .data[[inputs()$scatter_x]],
          y = .data[[inputs()$scatter_y]],
          color = .data[[
            inputs()$groupvar_scatter
          ]]
        )
      ) +
        geom_point(alpha = 0.7, size = 2) +
        geom_smooth(
          method = "lm",
          se = FALSE,
          aes(group = 1),
          color = "black",
          linetype = "dashed"
        ) +
        geom_smooth(
          method = "lm", se = FALSE
        ) +
        theme_minimal() +
        labs(
          x = inputs()$scatter_x,
          y = inputs()$scatter_y,
          color = inputs()$groupvar_scatter
        ) +
        theme(legend.position = "bottom")
    })
  })
}
```

The scatterplot module includes two trend lines: a dashed black line for the overall relationship and colored lines for each group. This dual-trend approach was generated by ChatGPT in response to the third prompt.

Correlation Module

```
correlationUI <- function(id) {
  ns <- NS(id)
  plotOutput(
    ns("correlationMatrix"), height = "400px"
  )
}

correlationServer <- function(id, data) {
  moduleServer(
    id, function(input, output, session) {
    output$correlationMatrix <- renderPlot({
      req(data())
      numericData <- data()[
        , sapply(data(), is.numeric)
      ]
      corrgram(
        numericData,
        order = TRUE,
        lower.panel = panel.shade,
        upper.panel = panel.pie,
        text.panel = panel.txt,
        main = "Correlation Matrix"
      )
    })
  })
}
```

The correlation module was generated as part of the first prompt response. It uses the corrgram package for visual correlation display. Note that this module does not take the `inputs` argument because the correlation matrix displays all numeric variables regardless of user selection.

Main Application Assembly

```
ui <- fluidPage(
  theme = bs_theme(bootswatch = "litera"),
  titlePanel("Palmer Penguins Explorer"),
  fluidRow(
    column(
      3,
      inputsUI("inputs"),
      style = paste0(
        "overflow-y: auto; ",
        "max-height: 600px;"
```

```
      )
    ),
    column(
      6,
      fluidRow(
        column(
          12,
          h4("Boxplot"),
          boxplotUI("boxplot"),
          style = "height: 50%;"
        ),
        column(
          12,
          h4("Scatterplot"),
          scatterplotUI("scatterplot"),
          style = "height: 50%;"
        )
      ),
      style = "overflow-y: auto;"
    ),
    column(
      3,
      h4("Correlation Matrix"),
      correlationUI("correlation")
    )
  )
)

server <- function(input, output, session) {
  filteredData <- reactive({ data })
  inputs <- inputsServer("inputs")
  boxplotServer("boxplot", filteredData, inputs)
  scatterplotServer(
    "scatterplot", filteredData, inputs
  )
  correlationServer("correlation", filteredData)
}

shinyApp(ui, server)
```

The main application uses a three-column layout: controls on the left, plots in the center, and the correlation matrix on the right. The `bslib` theme (Bootswatch "litera") provides clean typography without custom CSS.

Things to Watch Out For

1. **Reactive input forwarding is fragile.** The pattern of returning `reactive(input)` from the inputs module works but creates tight coupling. If new inputs are added, all consuming modules must be aware of the new field names.
2. **`na.omit(penguins)` drops observations aggressively.** The Palmer Penguins dataset has missing values in different columns. Dropping all rows with any missing value reduces the dataset from 344 to 333 observations. Consider column-specific filtering instead.
3. **The correlation module ignores user selections.** It always shows all numeric variables, which may confuse users who expect the correlation to reflect their current filter selections.
4. **ChatGPT does not add input validation.** The generated code uses `req()` but does not validate that selected column names actually exist in the data. Adding `validate(need(...))` calls would improve robustness.
5. **The tooltip helper manipulates widget internals.** The `add_tooltip` function directly accesses `input_ui$children[[1]]`, which depends on Shiny's internal widget structure. This may break with future Shiny updates.

Figure 3: A pair programming session with notes and reference documentation visible

AI-assisted prototyping works best when the human provides architectural judgment and the AI provides code scaffolding.

Lessons Learned

Conceptual Understanding

- AI-assisted prototyping works best as a conversation, not a single prompt. The three-prompt strategy produced better results than attempting to specify everything upfront.
- ChatGPT generates architecturally reasonable code (modules, separation of concerns) when the problem is well-defined and the target framework has extensive training data.
- The generated code is a starting point, not a finished product. Module interfaces, error handling, and edge cases all require human review.
- The Palmer Penguins dataset is small enough that performance is not a concern, but the patterns ChatGPT produced (loading data once, reactive filtering) would scale to larger datasets.

Technical Skills

- Shiny module architecture with `NS()`, `moduleServer`, and reactive input forwarding is idiomatic in the generated code.
- The `.data[[]]` pronoun for programmatic ggplot2 column selection is the correct tidy evaluation pattern and was generated without prompting.
- The `bslib` theme integration with `bs_theme()` provides professional styling with a single line.
- The `corrgram` package produces useful correlation visualizations with minimal configuration.

Gotchas and Pitfalls

- ChatGPT occasionally generates deprecated patterns (e.g., `aes_string()` instead of `.data[[]]`); review generated code against current best practices.
- The tooltip helper function accesses Shiny widget internals that may change between versions.
- The generated correlation module is disconnected from user selections, which could be confusing in a dashboard context.
- `na.omit()` is aggressive; column-specific filtering preserves more data and should replace the blanket removal.

Limitations

- This post demonstrates prototyping a simple exploration tool. Production applications require input validation, error handling, logging, and testing that AI-generated code typically omits.
- The evaluation is based on a single dataset (Palmer Penguins) and may not generalize to more complex data structures or domain-specific requirements.
- ChatGPT's output varies between sessions; the same prompts may produce different code at different times.
- The modular architecture, while reasonable, uses a pattern (reactive input forwarding) that creates tight coupling between modules.
- No automated tests were generated. A production application would need testthat tests for each module.

- The application does not handle large datasets, concurrent users, or deployment considerations.

Opportunities for Improvement

1. Add a data filtering panel that lets users subset by species, island, or sex before visualization.
2. Replace `na.omit()` with column-specific filtering using `tidyr::drop_na()` on only the columns needed for each plot.
3. Add `shinytest2` tests for each module to ensure the application behaves correctly after future modifications.
4. Implement a download button for each plot, allowing users to export visualizations as PNG or PDF.
5. Replace the `corrgram` correlation matrix with an interactive `plotly` heatmap that responds to user selections.
6. Add a data table tab using `DT::datatable()` so users can inspect the underlying data alongside the visualizations.

Wrapping Up

Prototyping a Shiny application with ChatGPT proved to be an effective approach for generating a working scaffold quickly. Three iterative prompts produced a modular application with boxplots, scatterplots, a correlation matrix, Bootstrap tooltips, and a clean theme (all in roughly twenty minutes).

What this exercise revealed is that AI-assisted prototyping shifts the developer's role from typist to director. The value is not in the generated code per se but in the speed of iteration. Starting broad and refining through conversation mirrors the natural prototyping process, compressed into a fraction of the usual time.

The generated code is not production-ready. It lacks input validation, automated tests, error handling, and documentation. But as a starting point for further development, it is substantially better than a blank file.

In conclusion, four points merit emphasis. First, three iterative prompts were sufficient to produce a working, modular Shiny application in under twenty minutes. Second, ChatGPT generates idiomatic Shiny code including modules and tidy evaluation patterns without explicit instruction. Third, the human role is to provide architectural judgment, review for correctness, and add production-quality features that the AI systematically omits. Fourth, AI-assisted prototyping functions as a speed multiplier, not a replacement for developer expertise.

See Also

Related posts:

- Combining Observable JS and Shiny: Mixing client-side and server-side reactivity in Quarto

Key resources:

- Shiny Modules: Official guide to Shiny module architecture
- Mastering Shiny (Wickham): Comprehensive Shiny textbook
- Palmer Penguins Dataset: Dataset documentation and examples
- bslib Package: Bootstrap theming for Shiny
- Shiny Gallery: Examples of production Shiny applications

Reproducibility

The complete application code is presented inline above. To reproduce:

1. Save the code blocks into a single `app.R` file.
2. Install the required packages.
3. Run `shiny::runApp("app.R")`.

```
Rscript -e "shiny::runApp('app.R', port=3838)"
```

Project files:

```
simpleshinyappwithchatgpt/
  analysis/report/index.qmd  (this post)
  app.R                       (complete app)
  analysis/media/images/      (hero, ambiance)
```

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from readers who:

- Spot an error or a better approach to any of the code in this post.
- Have suggestions for topics to cover.
- Want to discuss R programming, data science, or reproducible research.
- Have questions about anything in this tutorial.
- Simply want to say hello and connect.

Related posts in this cluster

This post is part of the *Shiny and Interactive Visualization* series. Recommended reading order:

1. Post 90: Combining Observable JS and Shiny in a Quarto Document
2. **Post 91: Prototyping a Shiny App with ChatGPT** (this post)