Paper SM11

The R Test Kitchen: Cooking Up Quality in Clinical Analytics & Reporting

Emily Yates, Formation Bio, NY, USA

ABSTRACT

In recent years, R has gained popularity in the clinical analytics and reporting space. Ensuring the validity and reliability of R-generated outputs is crucial for making informed, high-impact, clinical decisions based on reports, dashboards, and statistical analyses. This paper demonstrates the powerful ability to automate testing in R using packages like {testthat} and {shinytest}. This versatile framework can be applied to any number of clinical R use cases including R packages, RShiny dashboards, or ad hoc R analyses. We will cover how you can practically incorporate testing into your existing R workflows. Additionally, we will learn how adoption of good testing practices can translate into significant time savings and the delivery of high quality R-based deliverables.

INTRODUCTION

As the volume and complexity of clinical data continue to grow, ensuring the accuracy and reliability of R-generated outputs has become a critical challenge. Traditional manual QC methods, once sufficient for small datasets and static reports, are no longer feasible for modern clinical workflows involving genomics, wearables, and real-time lab data. The sheer volume of information makes exhaustive manual review impractical, while interactive dashboards and multi-step analyses introduce new failure points that manual methods cannot reliably catch.

Beyond inefficiency, reliance on manual testing introduces risks of human error and inconsistency. As R-based applications play a larger role in clinical decision-making, undetected errors in statistical analyses or visualizations can have significant consequences, from misleading conclusions to regulatory compliance issues. The need for rapid iteration further compounds the challenge, as each update requires re-validation to prevent unintended changes.

To address these issues, clinical data teams must adopt structured, automated testing frameworks that scale with data complexity. This paper explores three key solutions: unit testing with R's {testthat} and {shinytest2}, automated testing with CI/CD pipelines, and AI-assisted test generation. By implementing these approaches, teams can improve efficiency, reduce errors, and ensure high-quality results in clinical analytics.

PROGRAMMATIC TESTING IN R

One of the most effective ways to ensure quality in R-based clinical analytics is through programmatic testing. The {testthat} package provides a structured framework for unit testing, allowing developers to validate function outputs, check for edge cases, and confirm expected behaviors in their scripts and packages. Similarly, {shinytest2} enables automated testing of Shiny applications by recording and replaying user interactions to detect unintended changes in UI behavior.

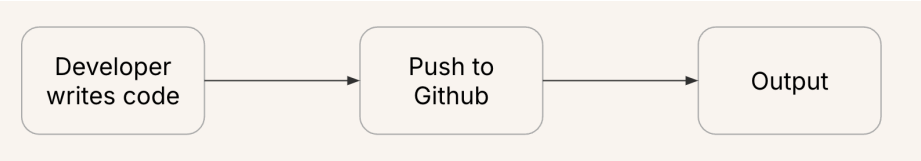

Figure 1: Standard baseline programming workflow where developers informally test their code

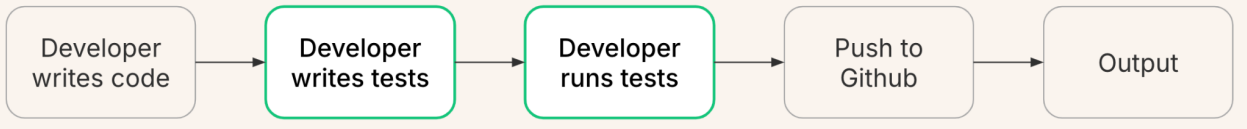

Figure 2: Programmatic testing to improve programmer workflow

```
> add_numbers <- function(x, y) x + y

> test_that("add_numbers correctly adds two numbers", {
+   expect_equal(add_numbers(2, 3), 5)
+   expect_equal(add_numbers(-1, 1), 0)
+ })
Test passed 🥳

> test_that("add_numbers handles edge cases", {
+   expect_equal(add_numbers(0, 0), 0)
+   expect_equal(add_numbers(Inf, 1), Inf)
+ })
Test passed 🌈
```

Figure 3: Simple {testthat} example executed locally

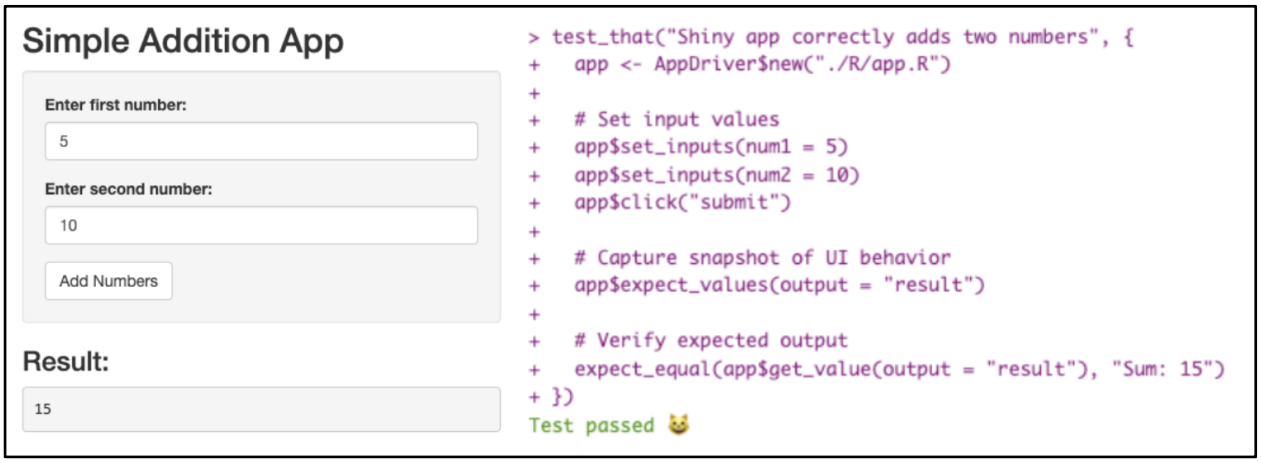

Figure 4: Simple {shinytest2} example executed locally

Implementing these tools improves reproducibility by transforming best practice testing that programmers already perform into documented and reproducible testing artifacts. From this, programmatic tests can be run repeatedly with minimal effort, unlike manual QC, which is time-consuming and prone to oversight. Additionally, programmatic tests help catch errors early in the development process, reducing the risk of incorrect outputs making it into clinical reports or dashboards

Despite these benefits, programmatic testing alone does not fully address scalability challenges. Tests must still be manually written and executed, and there is no built-in mechanism to enforce quality standards before results are shared. While these frameworks are essential for improving the accuracy and reliability of R-generated outputs, they must be complemented by additional automation to handle increasing data complexity and workflow demands.

AUTOMATING TESTING WITH CI/CD

While programmatic testing improves quality and reproducibility, manually running tests remains a bottleneck. To scale testing and enforce quality standards, clinical data teams can integrate continuous integration and continuous deployment (CI/CD) workflows. Tools such as GitHub Actions, GitLab CI, and Jenkins can be used to automate the execution of {testthat} and {shinytest2} tests whenever code is updated, ensuring that errors are caught before results are deployed.

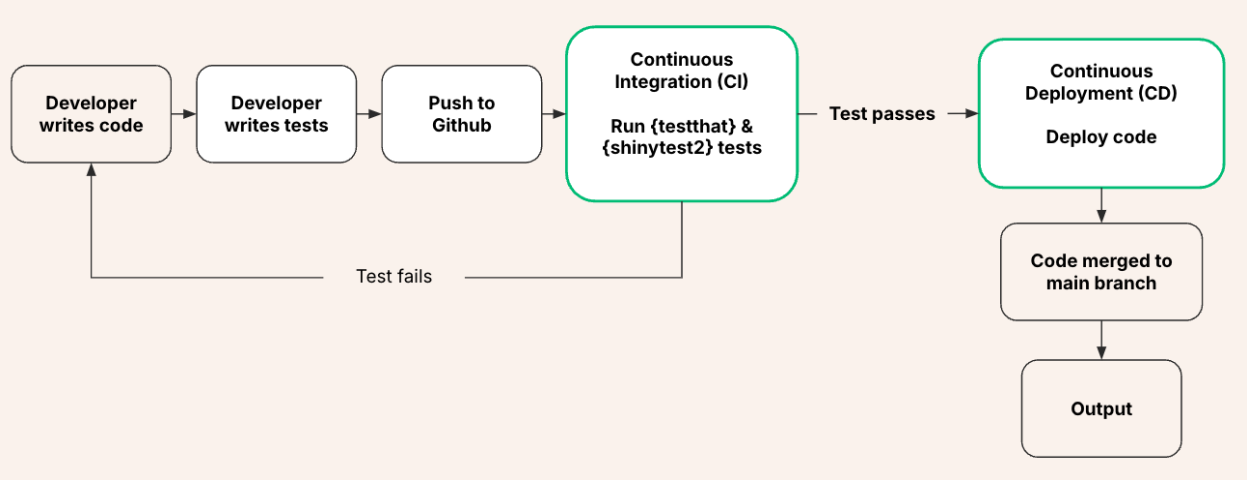

Figure 5: Addition of CI/CD into proposed workflow to automate & control test execution and deployment

CI/CD workflows automate the process of executing tests, reducing manual effort. Unlike manual execution of tests, CI/CD enforces quality by blocking code changes that fail tests, preventing faulty outputs from reaching stakeholders. Additionally, CI/CD maintains consistency across environments, as it ensures code behaves on local machines the same way it does in production environments.

Despite these advantages, CI/CD automation still relies on subject matter experts (SMEs) to write and maintain test cases. Setting up these pipelines requires technical knowledge, a very specific file folder structure, and manually creating comprehensive test coverage - which remains a time-consuming process. While CI/CD increases scalability and ensures consistency, it does not fully eliminate the effort required to design and maintain tests—an issue that AI-assisted test generation seeks to address.

AI-ASSISTED TEST GENERATION

While CI/CD automates test execution, the process of writing tests remains a bottleneck. Creating comprehensive test cases requires specialized expertise, making it time-intensive and difficult to scale across large datasets and complex clinical workflows. AI-assisted test generation offers a solution by leveraging large language models (LLMs) to automatically generate {testthat} unit tests and {shinytest2} snapshot tests, significantly reducing the manual effort required for test creation.

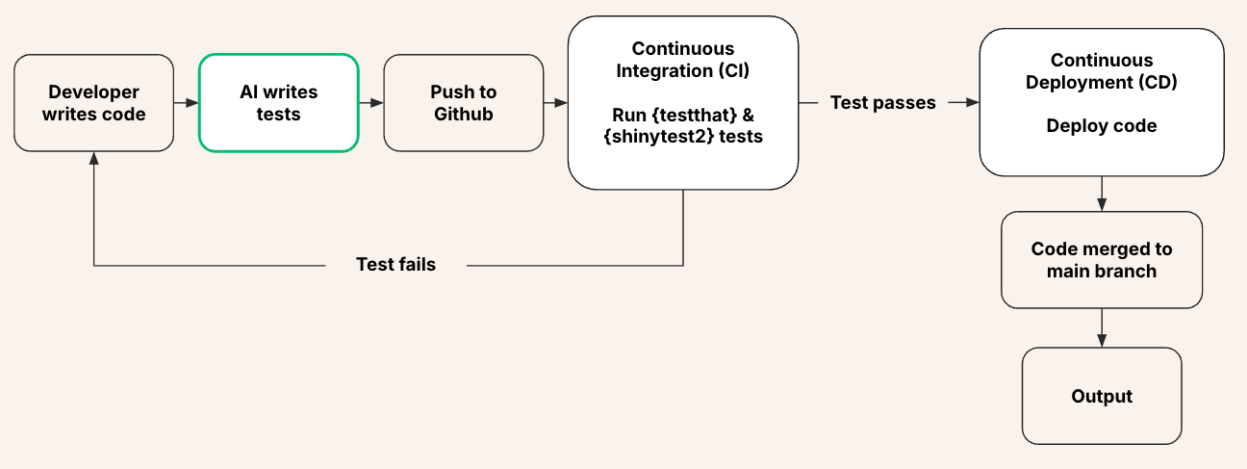

Figure 6: Addition of AI-Assisted test generation into proposed workflow to automate test writing

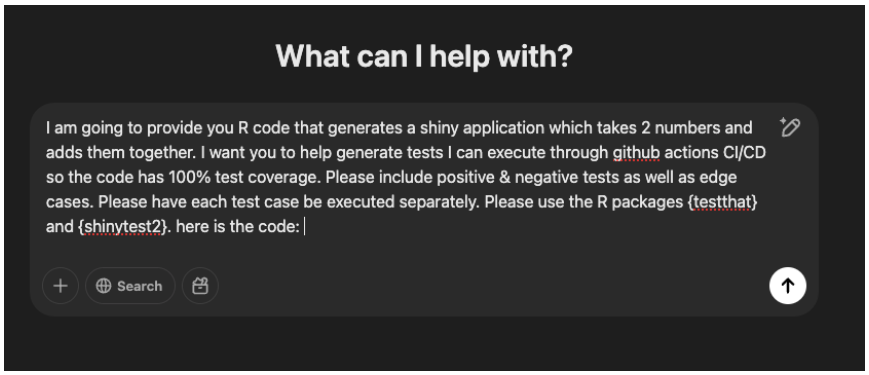

Figure 7: Specific and clear prompt to generate tests for R code

```r
# Define UI for the application
ui <- fluidPage(
  titlePanel("Simple Addition App"),
  sidebarLayout(
    sidebarPanel(
      numericInput("num1", "Enter first number:", value = 0),
      numericInput("num2", "Enter second number:", value = 0),
      actionButton("add_btn", "Add Numbers")
    ),
    mainPanel(
      h3("Result:"),
      verbatimTextOutput("sum")
    )
  )
)

# Define server logic
server <- function(input, output) {
  sum_result <- reactive({
    input$add_btn  # React when button is clicked
    isolate(input$num1 + input$num2)
  })

  output$sum <- renderText({
    sum_result()
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Figure 8: Example input code passed to LLM

```r
test_that("Addition works with large numbers", {
  app <- AppDriver$new(variant = platform_variant(), name = "large_numbers_test")
  app$set_inputs(num1 = 1e6, num2 = 2e6)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "3000000")
  app$stop()
})

test_that("Addition works with negative and positive number", {
  app <- AppDriver$new(variant = platform_variant(), name = "mixed_sign_test")
  app$set_inputs(num1 = -5, num2 = 10)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "5")
  app$stop()
})

test_that("Addition handles missing values correctly", {
  app <- AppDriver$new(variant = platform_variant(), name = "missing_value_test")
  app$set_inputs(num1 = NA, num2 = 5)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "NA")
  app$stop()
})

test_that("Addition handles NULL values correctly", {
  app <- AppDriver$new(variant = platform_variant(), name = "null_value_test")
  app$set_inputs(num1 = NULL, num2 = 5)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "NA")
  app$stop()
})

test_that("Addition handles non-numeric input correctly", {
  app <- AppDriver$new(variant = platform_variant(), name = "non_numeric_test")
  app$set_inputs(num1 = "abc", num2 = 5)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "NA")
  app$stop()
})
```

```r
test_that("Addition works for positive numbers", {
  app <- AppDriver$new(variant = platform_variant(), name = "addition_test")
  app$set_inputs(num1 = 3, num2 = 5)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "8")
  app$stop()
})

test_that("Addition works for negative numbers", {
  app <- AppDriver$new(variant = platform_variant(), name = "negative_numbers_test")
  app$set_inputs(num1 = -3, num2 = -7)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "-10")
  app$stop()
})

test_that("Addition works when one number is zero", {
  app <- AppDriver$new(variant = platform_variant(), name = "zero_test")
  app$set_inputs(num1 = 0, num2 = 10)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "10")
  app$stop()
})

test_that("Addition works when both numbers are zero", {
  app <- AppDriver$new(variant = platform_variant(), name = "both_zero_test")
  app$set_inputs(num1 = 0, num2 = 0)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "0")
  app$stop()
})

test_that("Addition works with decimal values", {
  app <- AppDriver$new(variant = platform_variant(), name = "decimal_test")
  app$set_inputs(num1 = 2.5, num2 = 3.5)
  app$click("add_btn")
  expect_equal(app$get_value(output = "sum"), "6")
  app$stop()
})
```

Figure 9: LLM generated tests

With AI-powered tools, developers can pass R code to an LLM, which then suggests appropriate test cases based on function logic, expected outputs, and edge cases. These AI-generated tests can be reviewed and refined by human experts, ensuring accuracy while minimizing the burden on SMEs. When integrated into CI/CD workflows, this approach enables scalable, high-quality testing without requiring extensive manual intervention.

CONCLUSION

As clinical datasets continue to expand in volume and complexity, ensuring quality in R-based analytics requires a shift from manual QC to automated testing. Programmatic testing with {testthat} and {shinytest2} improves reproducibility and catches errors early, while CI/CD pipelines enhance scalability by enforcing quality standards before deployment. However, both approaches still rely on manual test creation, which remains a bottleneck. AI-assisted test generation addresses this challenge by automating test creation, allowing clinical data teams to scale testing efforts efficiently while maintaining human oversight.

By adopting a combination of programmatic testing, CI/CD automation, and AI-driven test generation, organizations can improve efficiency, reduce human error, and ensure high-quality, reproducible outputs. These strategies are essential as clinical data science increasingly relies on large-scale, dynamic datasets, where manual validation is no longer feasible. Looking ahead, further advancements in AI-powered testing tools could streamline testing even further, enabling more adaptive, intelligent validation processes tailored to complex clinical workflows.

With these innovations, clinical data teams can confidently produce reliable, high-impact insights, ensuring that R-based analytics remain a trusted foundation for decision-making in the evolving landscape of clinical research.

REFERENCES

Wickham H (2011). “testthat: Get Started with Testing.” The R Journal, 3, 5–10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Wickham H, Bryan J, Barrett M, Teucher A (2024). usethis: Automate Package and Project Setup. R package version 3.1.0, https://github.com/r-lib/usethis, https://usethis.r-lib.org.

Schloerke B (2025). shinytest2: Testing for Shiny Applications. R package version 0.3.2.9000, https://github.com/rstudio/shinytest2, https://rstudio.github.io/shinytest2/.

Chang W, Cheng J, Allaire J, Sievert C, Schloerke B, Xie Y, Allen J, McPherson J, Dipert A, Borges B (2025). *shiny: Web Application Framework for R*. R package version 1.10.0.9000, https://github.com/rstudio/shiny.

Xie Y, Allaire J, Horner J (2023). _markdown: Render Markdown with 'commonmark'_. R package version

1.12, https://CRAN.R-project.org/package=markdown.

Allaire J, Dervieux C (2024). _quarto: R Interface to 'Quarto' Markdown Publishing System_. R package version 1.4.4, https://CRAN.R-project.org/package=quarto.

OpenAI. (2024). GPT-4 Technical Report. Retrieved from https://openai.com/research.

RECOMMENDED READING

"R Packages" by Hadley Wickham and Jennifer Bryan.

CONTACT INFORMATION

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

Emily Yates
Formation Bio
eyates@formation.bio