Paper OS05

Mastering Production-Grade Shiny Apps with Rhino: Lessons and Insights

André Veríssimo, Appsilon, Lisbon, Portugal
Vedha Viyash, Appsilon, Vellore, India

ABSTRACT

Building a shiny app is a great experience and an easy way to quickly share your work with others. However, when it is time to deploy it to production it is neither fast nor easy.

At Appsilon, we focus on building production-ready shiny apps and have learned a lot of lessons along the way. We created *{rhino}* to standardize the process of building production-grade shiny apps and decided to make it open source.

From the get-go, it introduces the best software development practices like managing the dependencies and setting up different levels of testing for your application.

In this talk, we will discuss the lessons we learned and how to think about apps built to last and easy to maintain.

INTRODUCTION

R, an open-source language widely used for statistical computing, has empowered data scientists, statisticians and software engineers to explore their ideas. The community built around this programming language has grown exponentially across the last decades with more than 25.000 packages in CRAN currently that allow users to use cutting-edge methods in different fields, process datasets, create graphics, create interactive visualizations and much more.

Building on the capabilities of R, Shiny has emerged as an accessible and powerful framework that allows users to quickly prototype interactive web applications. It enables anyone, who may lack extensive software engineering backgrounds, to rapidly create user-friendly applications for visualizing and exploring data in real-time. Shiny applications can be tailored to varying complexity levels, from simple visual dashboards to sophisticated, fully interactive web tools. This flexibility, coupled with a low barrier to entry, makes Shiny an ideal choice for bridging the gap between raw data analysis and interactive user engagement.

As applications grow in complexity and out of their prototype stage, problems start to appear with deteriorating performance, difficulty in introducing new features and more and more bugs affecting the growing user base. As such, it becomes crucial to implement good software development practices early in a project.

Fortunately, the R community has promoted a culture of shared insights and best practices, enabling developers to learn from each other's experiences. Additionally, open-source frameworks designed for R and Shiny have emerged, incorporating these best practices to support clean, modular code, consistency, and thorough documentation. These are essentials for sustainable, scalable applications.

TECHNICAL DEBT IN SHINY APPS

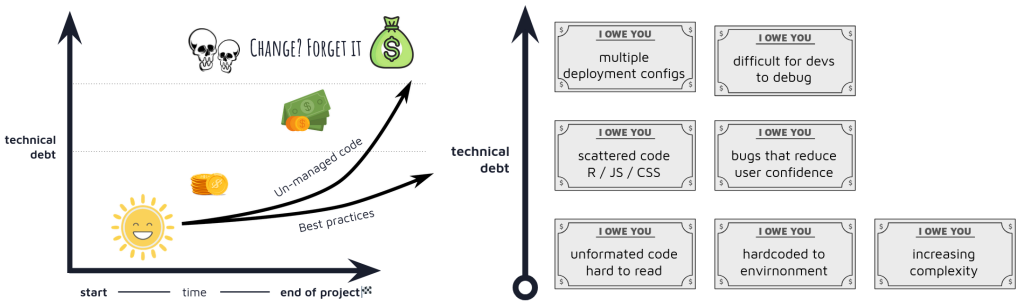

Figure 1: Technical debt during the project lifecycle

Technical debt refers to the cumulative cost of shortcuts or quick solutions taken during the software development process that can create maintenance challenges and inefficiencies over time. Initially, these quick fixes may seem advantageous, allowing teams to deploy features or meet deadlines more swiftly. However, as the codebase grows, these temporary solutions can compound into hidden costs, making the application harder to update, test, or scale. Much like financial debt, technical debt "accrues interest" in the form of increased time and resources required to add features or fix bugs down the line. If not addressed, it can lead to significant refactors, hindered productivity, and reduced stability, especially in complex applications. To manage technical debt effectively, best practices such as modular code, documentation, and periodic refactoring are crucial for maintaining long-term project health.

For Shiny applications, this might manifest as:

- Large monolithic application that has R files with thousands of lines, resulting in increased difficulty in development and onboarding new team members;
- Lack of testing, resulting in code that's challenging to maintain and prone to errors;
- Hardcoded parameters and environment that make the deployment process rigid and difficult;
- Missing documentation due to the fast pace of development that may lead to slower reaction times to events and issues.

Rhino, an open-source package developed by Appsilon, aims to combat this issue by embedding best practices into a framework, making it easier to build Shiny apps like a full-stack software engineer.

INTRODUCING RHINO FOR PRODUCTION-GRADE SHINY APPS

Rhino was created to support R developers in building production-ready Shiny apps by providing pre-configured tools that simplify development and enhance code quality. From the beginning, Rhino promotes modularity, dependency management, and testing, ensuring applications are reliable and maintainable. It reduces setup time by bundling essential tools and helps developers adopt industry-standard practices seamlessly.

OVERVIEW OF CORE TOOLS IN RHINO

Figure 2: Open source tools that Rhino comes shipped with

PROJECT ORGANIZATION & MODULARITY

Rhino's modular design and use of the {box} package bring together the expertise and best practices developed by engineers who have built and scaled Shiny applications across hundreds of projects. This structured approach is not only about organizing code but also about embedding lessons learned from real-world applications, where maintainability, scalability, and flexibility are crucial. By breaking down applications into manageable, self-contained components, Rhino ensures that even the most complex applications remain approachable, allowing developers to test, debug, and extend individual parts without impacting the entire system.

With {box}, each R file forms a module with its own namespace, allowing functions and objects to be encapsulated and explicitly exported only when needed outside the module. This encapsulation fosters clean, maintainable code that minimizes unintended interactions and makes the behaviour of each module predictable and self-contained. Such organization is invaluable in collaborative environments where multiple developers contribute, as it prevents overlap and conflicts by keeping each module's functionality contained and clearly defined.

The directory structure in Rhino further reinforces this modular approach, enabling developers to logically separate the application's logic, views, and data handling components. This structure reflects the separation of concerns that seasoned engineers know is critical for large Shiny applications, where the complexity of business logic, visual elements, and backend operations can quickly become overwhelming if not carefully segmented. Rhino's flexible

structure, while not enforcing strict patterns, encourages this separation, allowing teams to focus on specific functionalities without cluttering the codebase or making it difficult to onboard new contributors.

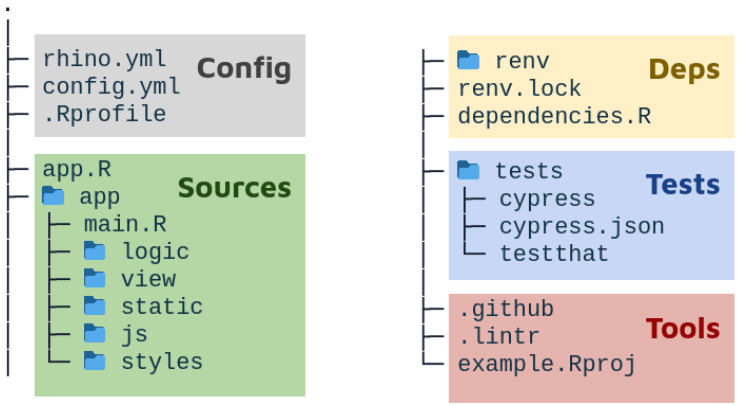

Figure 3: File structure built to host Shiny Apps

By emphasizing the modularity supported by {box} and Rhino's well-organized project structure, development teams can make architectural choices that best fit their application's specific needs. This flexibility, combined with a strong foundation of proven best practices, allows Rhino users to create applications that are scalable, easy to extend, and resilient to future changes. In essence, Rhino's project structure is a toolkit shaped by the cumulative knowledge of Shiny experts, helping teams avoid common pitfalls and accelerate the path from prototype to production-ready application.

TESTING AND VALIDATION

Testing is critical for stability in production applications. Rhino integrates {testthat} R package for unit testing and 2 packages for end-to-end testing, cypress and shinytest2. By covering a wide range of testing options, it ensures both individual functions and entire user workflows work as expected.

Automated testing allows developers to catch issues early, making updates easier and reducing downtime.

AUTOMATED WORKFLOWS

Rhino simplifies the adoption of Continuous Integration (CI) by providing a template of workflows that integrate with GitHub, automating code checks, testing, and other essential processes described in the other sections. By continuously testing the application with every change, CI reduces the likelihood of bugs, enforces consistency across contributors, and ensures that the application remains stable and functional as it evolves. Automated CI workflows catch issues early in the development cycle, allowing developers to identify and fix potential problems before they make it into the main codebase. This proactive approach minimizes the risk of introducing breaking changes, ultimately enhancing code quality and user experience.

With CI in place, teams benefit from immediate feedback on the impact of their code changes, creating a more agile development process where bugs are addressed promptly, and new features are validated quickly. This also allows for efficient collaboration across multiple contributors, as CI ensures each addition or modification is rigorously tested, reducing integration issues and maintaining a clean, stable codebase. Rhino's pre-configured CI workflows further streamline the process by ensuring that tests, linters, and other quality checks are consistently applied, which is especially helpful for teams with varying levels of experience.

In addition to improving code reliability, CI workflows enhance deployment confidence. As the application undergoes frequent testing and validation, developers can deploy new versions with the assurance that the application is unlikely to break or regress in production. This is particularly valuable for Shiny applications in production environments, where downtime or unexpected behaviour can significantly impact end users.

CODE QUALITY VIA LINTING & STYLING

Maintaining high code quality is essential for creating readable, maintainable, and error-resistant applications. Tools like linters and style guides automate checks to enforce consistent formatting and catch common issues, such as growing complexity in code, problematic coding patterns, unused variables and inconsistent indentation.

Rhino incorporates {lintr} and {styler} to apply these standards, helping teams maintain a uniform codebase that's easier to read and navigate. By ensuring stylistic coherence, these tools improve collaboration, reduce cognitive load, and help even mixed-experience teams work more efficiently within larger applications.

ENVIRONMENT MANAGEMENT

Using the {renv} package in R for dependency management is essential for maintaining a stable, reproducible environment across projects and deployments. By centralizing package dependencies, {renv} eliminates common issues associated with hardcoded paths, manually managed versioning scripts, and uncontrolled environments.

With `{renv}`, all package versions and dependencies are recorded in a lock file, ensuring that any project can be restored with the exact environment in which it was originally developed. This not only avoids errors but also removes the need for custom scripts or manual tracking, streamlining collaboration and saving time during deployment. In data-driven projects where reproducibility is crucial, `{renv}` provides robust control and peace of mind, allowing developers to focus on analysis and development without worrying about the intricacies of environmental setup.

Rhino comes in with a project setup for JavaScript and Sass out of the box. Both are compiled and minified offering a seamless integration of web technologies in the Shiny Application.

LOGGING & CONFIGURATION

The {config} package plays a crucial role in securing sensitive information and managing deployments and environments within Shiny applications. By enabling developers to store environment-specific settings, like database credentials, API keys, and other sensitive information, in a configuration file rather than hardcoding them into the application, {config} helps reduce the risk of accidentally committing sensitive information to a code repository. This approach safeguards the application from potential security breaches, as credentials are never stored in plain text within the codebase, but rather in an encrypted or protected configuration file.

Beyond security, {config} enhances the flexibility and manageability of Shiny applications across multiple environments. With {config}, developers can easily define different configurations for development, testing, and production environments, allowing seamless transitions and reducing manual overhead during deployments.

In addition to secure and manageable configurations, effective logging provided by {logger} is essential for Shiny applications. Logging provides structured, detailed logs that are invaluable for diagnosing issues and improving application stability. Logs offer insights into application behaviour over time, capturing errors, warnings, and user interactions that are crucial for troubleshooting and debugging. For instance, if a user encounters an error, {logger} can help trace back through the application flow to understand the issue's origin, aiding rapid resolution.

Moreover, {logger} enables real-time monitoring of application performance, usage patterns, and user activity, which can highlight areas for optimization or reveal unexpected behaviour before it affects end-users. In production environments, logging supports accountability, as it provides a comprehensive activity history that is useful for auditing and security compliance. Together, {config} and {logger} bring enhanced security, flexibility, and operational transparency to Shiny applications, making them better equipped for deployment in secure, production-grade settings.

CASE STUDIES AND PRACTICAL APPLICATIONS

In real-world projects, applications often evolve from a focused prototype into a larger, more complex solution, presenting new challenges as they grow to meet expanded requirements. The following case studies illustrate how the best practices presented in this paper can be applied to tackle common issues faced by Shiny developers: from refactoring monolithic codebases to enhancing modularity and scalability in large applications.

In **Case Study 1**, we explore a scenario where a Shiny application evolved rapidly, leading to a bloated codebase and performance issues. The original app's monolithic structure and tight coupling of components made it challenging to introduce new features, maintain, and optimize. Here, Rhino's modular architecture and tool integrations enabled the team to decouple the logic, streamline data processing, and ultimately reduce the app's load times.

Case Study 2 presents a different challenge: a complex Shiny app with a codebase that was difficult to modify or extend due to inconsistencies in how components were structured. In this case, Rhino's structured approach, using the {box} package for modularity and directory organization, was instrumental in reorganizing the application. This restructuring not only clarified the separation between business logic and user interface components but also made it easier to scale the app with new features and support diverse data sources.

CASE STUDY 1: REFACTORING A LONG APP.R FILE

A very common pattern when analysing existing Shiny applications is the presence of large R files with thousands of lines of code. In this case study, we are looking into an application that grew too fast and is facing serious performance issues, taking up to 10 seconds to process the raw data, filter the results and display the results in an interactive plot.

Our business goal was to create a proof-of-concept based on the existing application to help decide on the future of that product. Whether to eliminate the product and start over, or clean the code base and improve the performance of the application and therefore, satisfy the users. In technical terms, we needed to reduce the loading times and avoid locking the application with every filter change.

The process started with an analysis of the code, benchmarking the application's performance with quantitative data and creating an extensive action plan with the cost and impact of each item. With this information in hand, we started addressing the critical architectural issues first, so that we could build upon this new foundation.

The project was possible to succeed without Rhino, however, we determined that it would make the process smoother and faster if we used the tools available in the framework. The first set of actions can be grouped into a "separating logic from view" category, leveraging Rhino project organization and modularity capabilities. As such, we:

- Decoupled the data retrieval from AWS and the processing pipeline outside of the Shiny application with an independent API to help simplify communication and increase confidence in the data;
- Broke down the monolithic Shiny implementation of the different elements into separate modules, further separating the logic from the view;
- Created a set of functions that generate interactive visualizations. This gave us the flexibility to delegate the computationally heavy visualizations to a new API outside the Shiny application.

During this initial phase, we were continually improving on code quality, by moving the codebase to smaller files, linting this new code and cleaning repetitive code/process.

We developed early on a set of end-to-end specifications that would look into a sample of interactive visualizations and filters. This would be the basis for the application benchmark using {shiny.benchmark} R package from the Rhinoverse family.

At the end of this phase, we converted a file with 5000+ lines of code into a manageable structure with 6 logic modules that perform the bulk of data and backend operations and 9 views containing the Shiny modules managing all the visual elements (filters, plots and tables).

In the second phase of the project, the data operations and expensive plots were offloaded from the Shiny application and packaged in a Plumber API. The reduction in computation requirements allows the application to be very responsive, mitigating some of the limitations of using a Shiny application with many users. The API was replicated behind a load balancer, which can be scaled up depending on the number of concurrent users.

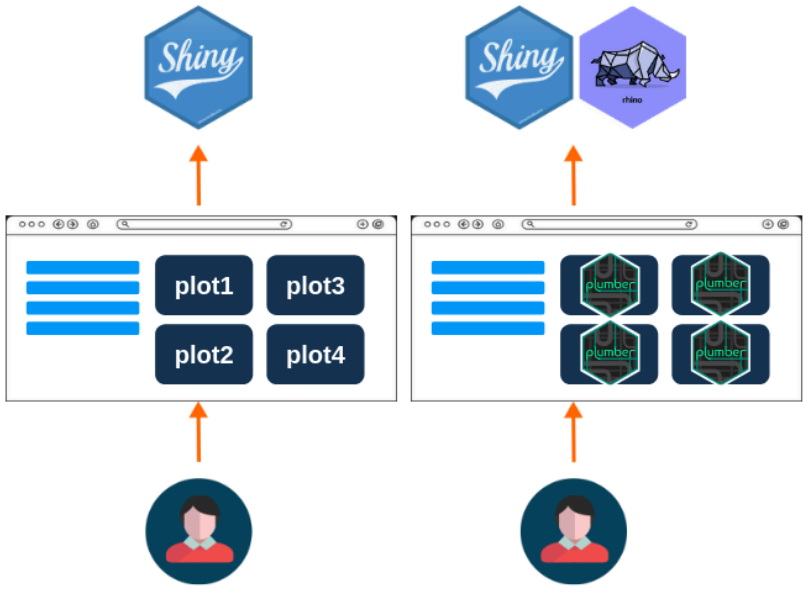

Figure 4: Change in architecture in phase 2 delegating computational heavy operations to Plumber REST API

Lastly, we also used the configuration and resource management capabilities of Rhino to allow for production and development environments, as well as an explicit dependency version management that ensures an identical behaviour of the application, regardless of where it is deployed. This also addressed some security concerns as credentials and access information were being stored in plain text.

In conclusion, this project required a large amount of resources and effort to tackle the technical debt accrued over time. While it is impossible to have a project without any issues, some steps can be taken to reduce the risks, such as having a team with extensive software engineering knowledge and experience, as well as understanding the tools available to ensure code quality and best practices.

CASE STUDY 2: ADDRESSING INEFFICIENCIES IN A COMPLEX CODEBASE

Building a large Shiny app with multiple moving parts requires careful design and planning. Common design patterns can help solve recurring challenges. One significant issue in developing large Shiny apps is determining how to effectively split individual modules. This split can often be subjective, leading to a wide range of implementations for the same app. Rhino addresses this issue by encouraging developers to separate business logic from Shiny module content into dedicated directories. This simple organization enables developers to make design choices that are more maintainable.

In this case study, we examine a large Shiny app with an architecture that made it difficult to modify or extend. Although the app was modularized and divided into multiple files, maintenance remained challenging. The business goal was to update all static visualizations in the app to interactive visualizations and create similar plots using different data sources.

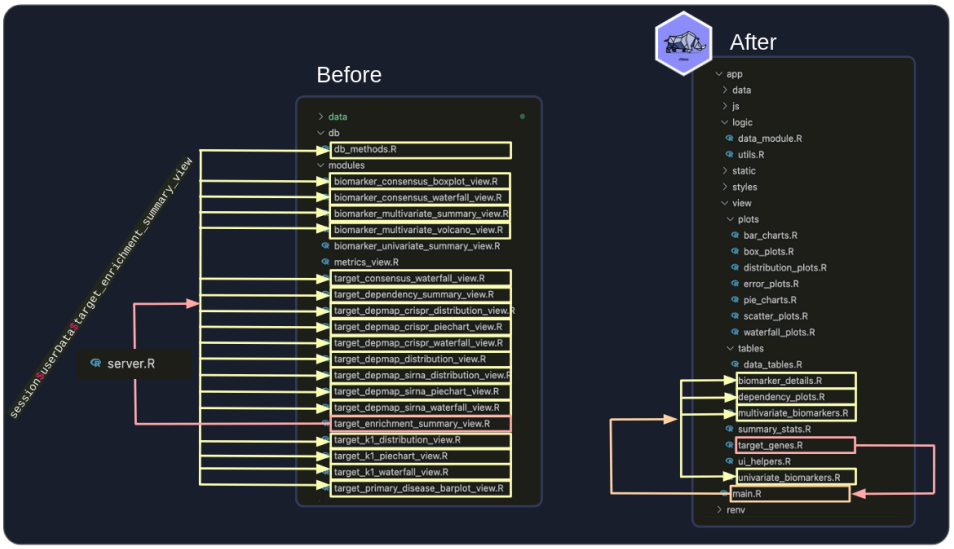

Figure 5: Before and after the rhino migration and re-modularization

Our first step was to re-modularize the app into meaningful components. Migrating the app to Rhino facilitated this process by enforcing a separation between logic and Shiny code. This structure clarified the distinction between the two, allowing us to identify that the plot/table generation code was non-Shiny-related and consisted solely of business logic. We confidently moved these components into the logic directory, simplifying the structure of the Shiny modules and enhancing the overall reactive flow of the app, which significantly improved future maintainability.

Following the migration to Rhino, we were able to implement the necessary changes requested by app users. We applied a similar file and logic structures to other Shiny apps. By the end of developing two additional Shiny apps, the team had gained confidence in making modifications independently.

CONCLUSION

In conclusion, this paper highlights the significance of adopting structured frameworks, such as Rhino, for developing robust, production-ready Shiny applications. As Shiny apps evolve from simple prototypes to complex tools, technical debt and code maintenance challenges can quickly accumulate.

The packages and workflows present in the Rhino framework address these concerns by promoting modularity, testing, and best practices in environment management, allowing developers to scale applications efficiently while maintaining code quality and performance.

The case studies illustrate how Rhino's structured approach can significantly improve application responsiveness, modularity, and maintainability, ultimately supporting seamless growth and adaptation of Shiny applications in diverse, production-grade environments. By implementing tools and best practices from the outset, developers can avoid many pitfalls associated with technical debt, streamline their workflows, and deliver high-quality applications that meet both current and future needs in data-driven industries.

REFERENCES

Tom, E., Aurum, A., Vidgen, R. (2013) An exploration of technical debt. https://doi.org/10.1016/j.jss.2012.12.052.

Hornik, K. (2012). The comprehensive R archive network. *Wiley interdisciplinary reviews: Computational statistics*, *4*(4), 394-398. https://doi.org/10.1002/wics.1212.

Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., Borges, B. (2024). *shiny: Web Application Framework for R*, <https://CRAN.R-project.org/package=shiny>.

Fowler, M., & Foemmel, M. (2006, May). *Continuous integration*. https://martinfowler.com/articles/continuousIntegration.html.

Żyła, K., Nowicki, J., Siemiński, L., Rogala, M., Vibal, R., Makowski, T., Basa, R. (2024). *rhino: A Framework for Enterprise Shiny Applications*. https://CRAN.R-project.org/package=rhino.

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.

Ushey, K., Wickham, H. (2024). *renv: Project Environments*. https://rstudio.github.io/renv/.

Rudolph, K. (2024). box: Write Reusable, Composable and Modular R Code. https://CRAN.R-project.org/package=box.

Schloerke, B. (2024). *shinytest2: Testing for Shiny Applications*. https://CRAN.R-project.org/package=shinytest2.

Allaire, J. (2023). *config: Manage Environment Specific Configuration Values*. https://CRAN.R-project.org/package=config.

Hester, J., Angly, F., Hyde, R., Chirico, M., Ren, K., Rosenstock, A., Patil, I. (2024). *lintr: A 'Linter' for R Code*. https://CRAN.R-project.org/package=lintr.

Müller, K, Walthert, L. (2024). *styler: Non-Invasive Pretty Printing of R Code*. https://styler.r-lib.org.

Daróczi, G., Wickham, H. (2024). *logger: A Lightweight, Modern and Flexible Logging Utility*. https://CRAN.R-project.org/package=logger.

Masse, M. (2011). *REST API design rulebook*. O'Reilly Media, Inc. ISBN: 978-1-449-31050-9

CONTACT INFORMATION

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

Author Name: André Veríssimo

Company: Appsilon

Email: andre.verissimo@appsilon.com

Website: appsilon.com