Paper DV02

Elevating Data Advocacy: Transforming Dashboards with React.JS and AI-Powered Innovation

Richard Becker, Merck, London, UK
Markus Niederstrasser, Merck, Tokyo, Japan

ABSTRACT

Dashboards play a critical role in delivering real-time data insights and fostering data advocacy within organizations. At Merck, we leverage dashboards to empower decision-making and drive data-driven strategies. In this presentation, we will showcase practical examples of how dashboards are utilized across various functions at Merck, highlighting key lessons learned in their development and adoption. A focus will be placed on the use of React.JS components leveraging Merck's branded Liquid Design System to enhance user experience, streamline functionality, and ensure a consistent look and feel across applications. Additionally, we will investigate the integration of Large Language Models (LLMs) into dashboards to assist users with output generation, thereby improving usability and optimizing standard user interactions.

INTRODUCTION

Digital transformation in clinical development is now essential for organizations striving for rapid, traceable, and interactive reporting. At Merck, the transition from static reports to interactive dashboards has become a cornerstone for supporting data-driven decision making. Traditional approaches, which relied heavily on static tables and delayed reporting cycles, are increasingly being replaced by solutions that offer real-time insights, flexible exploration, and the ability to trace every decision back to its source. By harnessing modern dashboard technologies and a robust design system, we have enabled teams to access standardized outputs in a fraction of the time previously required, support regulatory rigor, and foster a culture of data advocacy.

BACKGROUND AND MOTIVATION

The Analytical Programming and Standards Team (APS) end-to-end pipeline, which aims to streamline the process from Protocol to analysis Output creation, at Merck, illustrates the complex journey from raw SDTM and ADaM data to standardized outputs and comprehensive regulatory reports. The pipeline includes multiple points where dashboards are facilitated. One use case is a dashboard supporting safety signal detection, while another aims to shorten the time to deliver Clinical Study Report (CSR) key statistics. This paper focuses on lessons learned from implementing these dashboards.

The interactive Clinical Study Report (iCSR) R Shiny dashboard application was developed to address several persistent challenges in this pipeline. Historically, the turnaround time for generating key study statistics and outputs was slow, making it difficult for teams to adapt to changing analysis requirements. There was also a lack of transparent traceability, making it challenging to link analysis decisions directly to final reports. Furthermore, the limited interactivity of legacy tools created barriers for subject matter experts (SMEs) who needed to explore data and generate insights independently.

User surveys conducted during the design phase revealed a strong demand for dynamic data visualization and the ability to flexibly define and explore subgroups. Users also emphasized the importance of transparent provenance ensuring that every result could be traced and reproduced. Many SMEs, regardless of their programming background, sought simplified workflows that would allow them to interact with the data and outputs without steep learning curves. Finally, there was a clear need to integrate these solutions with Merck’s corporate design and accessibility standards, ensuring consistency and inclusivity across applications.

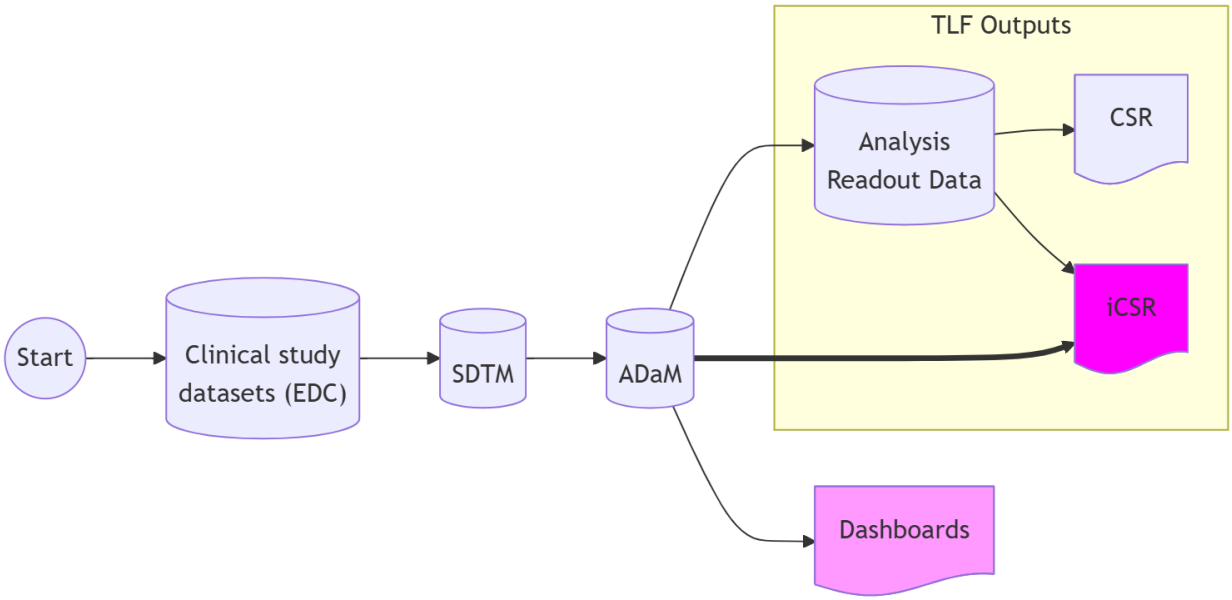

Figure 1: Dashboards integrations within APS end-to-end pipeline.

SYSTEM ARCHITECTURE

The iCSR application is built on a modular R Shiny foundation, with each analysis module encapsulated for maintainability, scalability, and clear ownership. Data is loaded into a shared reactive state to enable seamless transitions between modules, and outputs without redundant loading or state conflicts. A significant UI enhancement comes from adopting Merck's Liquid Design System via the in-house R package {liquid.react}. This ensures a consistent look and feel aligned with the company's UX strategy, improves accessibility, and reduces UI drift and user confusion across applications. Because React components manage their own state and lifecycle, they update predictably in response to Shiny-driven props while remaining isolated from global page context and Bootstrap defaults.

At a high level, raw SDTM and ADaM data are ingested and processed through modular Shiny components. These modules interface with standardized output engines—principally the internally developed {RDOT} package, which focuses on safety and efficacy key stat outputs. RDOT provides commonly required outputs such as Demographic Characteristics, Adverse Events Overviews, Shift Tables, Kaplan–Meier plots, Forest Plots, and more. Under the hood, {RDOT} relies heavily on {Tplyr} for table creation and on {gt} for rendering. Final deliverables can be exported to RTF, or HTML via {gt} and PDF using LaTeX to support standard output formats. Results are rendered in a React-based UI for a responsive, accessible user experience, and all user actions and output parameters are captured via {shinymeta} to ensure full traceability and reproducibility. The overall system is designed to be extensible, including future enhancements such as LLM-assisted narratives or guided workflows.

From a user and developer perspective, responsibilities are cleanly separated: {liquid.react} provides a consistent React-based design layer; Shiny orchestrates reactivity and shared state; {RDOT}/{Tplyr} enforce standardized output creation logic; and {teal.reporter} aggregates artifacts into reports (RTF/PDF/HTML) via {gt}. This separation of concerns simplifies testing, maintenance, and evolution of the stack. Supporting packages include {teal.filter}/{teal.data} for filtering and data panes, {shinymeta} for code capture, and {shinytest2} (with testthat and a headless Chrome engine) for unit and integration tests.

By design, the application keeps dependencies lean. The ground framework is plain R Shiny with modules rather than higher-level frameworks such as {DaVinci}, {teal}, or {Rhino}. During initial development, none of these frameworks fully met the user requirements, and adopting them would have introduced ecosystem lock-in and additional abstraction layers. For future dashboards, a lightweight, in-house framework aligned closely to Shiny modules was therefore preferred to preserve flexibility, minimize overhead, and keep the learning curve low.

Where functionality was stable and closely matched requirements, extensions were either developed in-house or selectively reused from the {teal} ecosystem. For example, integrating {teal.reporter} into an existing Shiny app is straightforward for report "basket" functionality; however, deeper modifications or extensions can be less intuitive and risk diverging from the upstream {teal.reporter} codebase, complicating future updates. At the time of initial

implementation, several {teal} packages were still maturing. Development in {teal} and related packages (e.g., {teal.data}) is advancing quickly and should be reconsidered for adoption in future dashboards where it cleanly satisfies requirements without imposing unnecessary constraints.

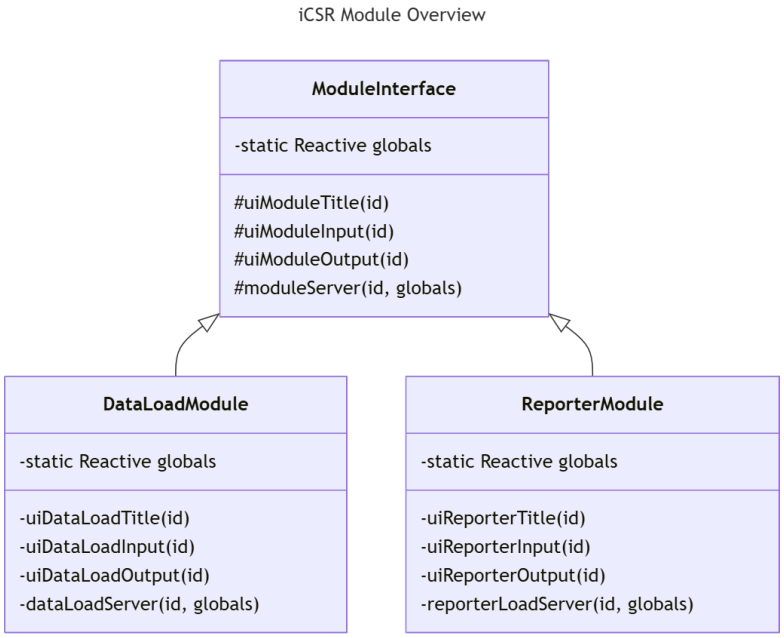

Figure 2: iCSR R Shiny module abstraction overview.

UNIT AND INTEGRATION TESTS

We implemented a layered test strategy combining {testthat} for unit tests and {shinytest2} for end-to-end browser-driven integration tests. Unit tests target pure helpers and module server logic in isolation, while integration tests validate full user flows (including React-based UI via {liquid.react}) in a real headless Chrome environment.

For unit testing, {testthat} exercises module logic deterministically using predefined fixtures (e.g., dummy ADaM/SDTM extracts and settings) stored in helper objects such as test_gen_globals. This keeps tests fast and reproducible, avoiding UI timing concerns.
Example of a module unit test using {testthat} is shown below:

```R
testServer(anasetServer, args = list(globals = test_gen_globals()),{
  session$setInputs(popFl = "SAFFL")
  session$setInputs(trtVar = c("TRT01P", "TRT01PN"))

  # expect_no_error
  expect_no_error(session$setInputs(btnRender = TRUE))

  # output is a data.frame with 1 row and 6 columns
  expect_true(is.data.frame(globals$output_card$data))
  expect_equal(dim(globals$output_card$data), c(1,6))
})
```

Integration testing uses `shinytest2::AppDriver()` to launch the app with a headless Chrome session (via Chromote) and simulate real user interactions. Because every dashboard action triggers reactive bindings (and often initializes React components), explicit synchronization is crucial; `app$wait_for_idle()` (and, when needed, waits for specific values or selectors) is used between interactions to ensure stable state before assertions.

In most environments, this setup runs reliably. However, Chromote-driven tests can be memory intensive and leading to issues depending on R server configuration. Mitigations include reducing viewport and screenshot sizes, limiting parallel test workers, inserting modest wait times for complex renders, explicitly closing drivers between tests.

KEY FEATURES AND CAPABILITIES

The iCSR application serves both technical and non-technical users by pairing standardized study readout outputs with intuitive interactivity. Common “key stats” can be generated quickly, while users refine analyses through filtering and subgroup definition, with options to save and share configurations for future runs.

Transparency is built into the workflow: each output is accompanied by its generating code and parameters captured via {shinymeta}, enabling full provenance and straightforward reproduction. Curated results can be collected in a report “basket” implemented with {teal.reporter} and adapted to our in-house {RDOT} output framework, then exported in RTF, PDF, or HTML.

To meet internal SOPs and quality standards, the solution employs automated unit and integration tests, provenance logging, and golden-file comparisons for layout stability. These controls not only satisfy regulatory review and archiving needs, they also provide reliable audit evidence and streamline change control by detecting regressions early.

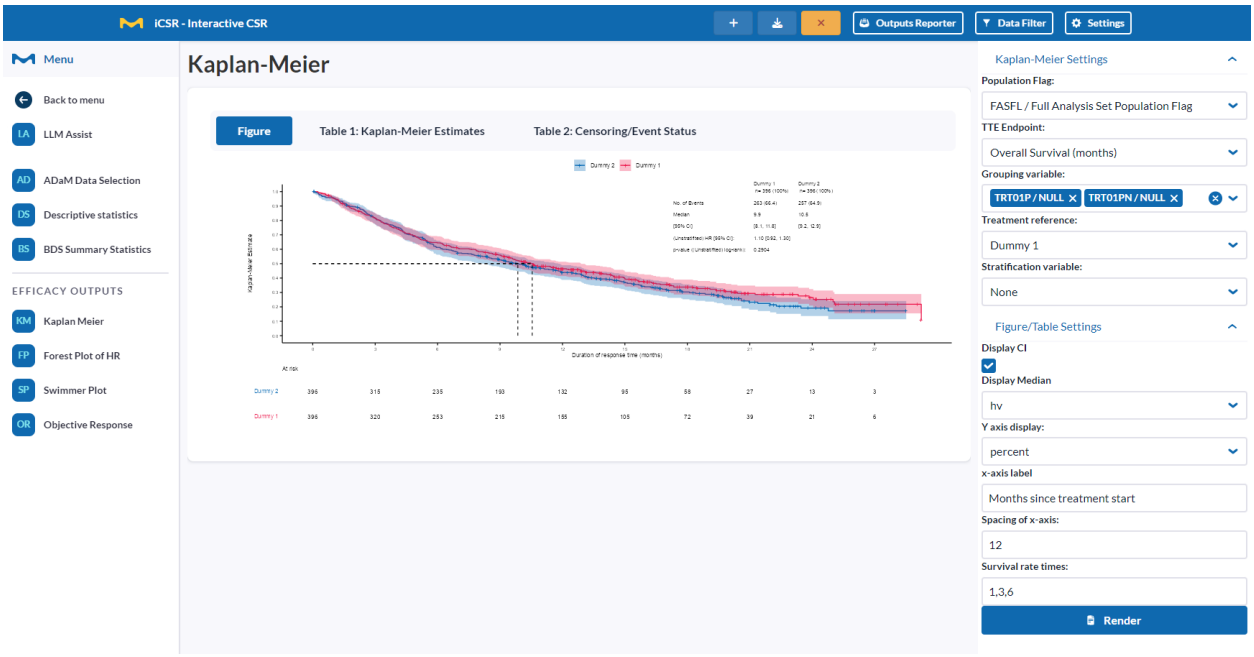

Figure 3: iCSR dashboard layout displaying KM outputs with {liquid.react} UI.

PRACTICAL LESSONS LEARNED

Adoption of the iCSR dashboard underscored the value of empowering SMEs with interactive tools, supported by clear onboarding and in-context guidance. Iterative development with frequent user feedback was essential; early prototypes exposed navigation and filtering friction that was resolved through focused UX improvements.

Consistency in look and feel—delivered through the Liquid Design system—reduced cognitive load and training overhead across teams. Equally important was end-to-end traceability: linking each result to its inputs and logic, and producing deterministic, submission-ready artifacts with seeded runs and rich metadata. Comprehensive testing (e.g., {shinytest2} and {testthat}) kept output behavior stable over time, while the emphasis on reproducible workflows ensured that enhancements could be made without compromising compliance.

REACT.JS INTEGRATION IN R SHINY

Integrating React.JS with Shiny has enabled the development team to deliver a more modern, responsive, and accessible user interface. Merck has developed an inhouse UI library names Liquid Oxygen link which is used as a unified UI framework across the company. Using Liquid Oxygen across projects ensures a consistent look and feel across related applications/dashboards. Liquid Oxygen make heavy use of web components to build framework-agnostic and future-proof UI components. The framework-agnostic nature of Web Components allows you to use Liquid Oxygen in any framework (e.g. React, Vue, Solid or Svelte) or even without a framework.

The natural choice to integrate Liquid Oxygen into existing R Shiny apps was via its React.JS framework, which brings accessible, consistent components into dashboards, improves performance and reuse, and coexists cleanly with Bootstrap CSS/JS because React's virtual DOM isolates component updates from global styles and scripts—making it feasible to use React components alongside existing or hard-coded Bootstrap elements that underpin many Shiny defaults.

The inhouse developed R package {liquid.react} capsules those Merck Liquid Oxygen components as Ld*() functions and makes them available inside the UI or Server part of a Shiny App. Example code :

```R
library(shiny)
library(liquid.react)

shinyApp(
  ui = tagList(
    LdHeader(siteName = "Liquid Oxygen"),
    LdButton("Test Button"),
    LdLoading(),
    LdIcon(name="placeholder")
  ),
  server = function(input, output) {}
)
```

Technically, {liquid.react} builds on {shiny.react} and ships a webpack-built bundle of Liquid Oxygen React components and styles. The webpack entry file (index.js) imports the Merck Liquid Oxygen React component library alongside {shiny.react} adapters (e.g., InputAdapter and ButtonAdapter). Using these adapters, we expose Ld* components for R Shiny by wrapping the original React components and wiring up value/prop synchronization, event handling (e.g., onChange), and serialization. These wrappers provide the hooks Shiny needs to set/get values and propagate reactivity without leaking side effects into the global page context. For further reference, see the {shiny.react} vignette link. Below is an extract of the index.js file.

```JS
const Liquid = require('@emdgroup-liquid/liquid/dist/react');
const { InputAdapter, ButtonAdapter } = require('@/shiny.react')

require('@emdgroup-liquid/liquid/dist/css/liquid.global_no_fonts.css');

export const LdButton = ButtonAdapter(Liquid.LdButton);

export const LdSlider = InputAdapter(Liquid.LdSlider, (myValue, setValue) => ({
  onLdchange: (event) => {
  setValue(event.detail);
  },
  value : myValue
}));

window.jsmodule = {
  ...window.jsmodule,
  '@emdgroup-liquid/liquid': Liquid,
  '@/shiny.liquid': { LdButton, LdSlider }
};

```

LLM-ASSISTED DASHBOARDS: OPPORTUNITES AND RISKS

Integrating Large Language Models (LLMs) into dashboards is a non-trivial endeavor. Beyond simply adding a chatbot (e.g., a sidebar assistant), value comes from well-defined, domain-specific use cases, robust guardrails, and tight coupling to the dashboard's data and provenance. In practice, LLMs can generate narrative summaries for outputs, propose R code for data manipulation, and provide contextual help and documentation. For example, a user might request a narrative summarizing treatment-emergent adverse events; the LLM can produce a context-aware, human-readable summary consistent with the displayed analysis.

Because LLMs can hallucinate plausible but incorrect content, we apply human-in-the-loop review and log all interactions for traceability. Security is addressed by restricting the context window, using prompt templates to constrain scope, and validating any generated code before execution. Determinism and reproducibility remain the default for final outputs.

Potential use cases we evaluated for generative AI dashboards within iCSR (using {shinychat} for the UI and {ellmer} for connectivity to the Merck internal LLM API) include:

- Chat assistant for navigation and in-context documentation, tailored to the current module and data view.
- AI-generated summaries or translations of figures/tables to accelerate drafting of CSR-ready narratives.
- Dashboard control via tool calling (e.g., apply filters, switch modules, preconfigure inputs) when workflows are complex or repeated.
- R code generation leveraging an internal snippet repository to scaffold reproducible filters, subgroup definitions, or output customizations.

Technical implementation follows a simple pattern: a sidebar chat interface built with {shinychat} and a backend bridge via {ellmer} to the internal LLM API. Tool calling extends the assistant's capabilities by invoking well-scoped Shiny functions (e.g., update filters, navigate modules, refresh outputs). In practice, we found that full UI control by the LLM adds limited net benefit—(experienced) users are often faster at direct interaction—so we position the assistant primarily as a side helper for R code generation, subgroup definition, and narrative drafting rather than automated UI driving.

To ensure responsible and transparent AI integration, governance and guardrails are essential, with LLM oversight strengthened in several key areas:

- Prompt discipline: curated templates, role guidance, and minimal necessary context to reduce drift.
- Provenance: capture prompts, parameters, and generated code; link to outputs via {shinymeta}.
- Validation: execute generated code in a safe sandbox with unit/integration tests; prefer deterministic outputs for archival.
- Security and privacy: limit access to sensitive data; enforce rate limits and session scoping; redact volatile identifiers in prompts. No patient-level information is shared with the LLM; only the minimal data-frame schema required for code generation (variable names, labels, and types) is provided.

We found the most value by targeting specific, high-impact use cases with strong guardrails rather than adding a generic chatbot. Two proven implementations in iCSR directly address common user needs while preserving governance and reproducibility.

Proven use cases implemented in iCSR

- R code generator for dynamic subgroups
    - Problem: SMEs often need ad-hoc subgroup variables (e.g., a new subgroup variable based on different biomarker cut-off) but may not write R fluently.
    - Solution: An LLM assistant that turns plain English into executable R code (e.g., dplyr syntax) based on the known data schema.
    - Implementation highlights:
        - User describes the subgroup in natural language; the LLM generates R code.
        - Code is shown for review prior to execution; type/column checks run against the schema.
        - Execution runs in an isolated R environment; any changes to the loaded data frames are returned to the app and immediately available to all modules.
        - No patient-level data is shared; only minimal schema (variable names, labels, types) is sent for code generation.
    - Impact: Creation of adhoc variables/modifications reduces the need to modify ADaM for exploratory outputs leading to reduced time to output for common adhoc updates.
- Documentation helper and navigation assistant
    - Problem: New users need guidance to navigate modules and understand available analyses.
    - Solution: A context-aware chat experience that surfaces relevant how-tos and definitions based on the current module view.
    - Example interactions: “How do I create a Kaplan–Meier plot?”, “What does SAFFL mean?”, “Where do I export this table?”
    - Impact: Easier navigation for end user and improving accessibility.

ROADMAP AND FUTURE VISION

Looking ahead, the iCSR application is poised for meaningful enhancements that deepen compliance, accelerate workflows, and improve user experience. Migration of data access to a new Analytical Platform (AP) will centralize storage, strengthen role-based access controls, and provide auditable data lineage across projects. {RDOT} output coverage will be broadened to include additional analysis domains—such as flexible subgroup table layout, exposure and dosing, and time-to-event endpoints—so teams can standardize more of the study readout within one environment. The built-in sandbox environment will let SMEs safely experiment—on in-memory copies of the globally loaded data—with variable selection, on-the-fly subgroup definitions, ad-hoc variable creation, and exploratory visualizations. All changes are non-destructive and do not modify the source SDTM/ADaM datasets; when ready, users can export the transformation steps as reusable code with clear lineage and modified datasets back to the dashboard modules.

A key focus is advanced traceability and reproducibility. We aim to provide full lineage from CSR outputs back to ADaM and SDTM sources, including explicit mappings and metadata (e.g., derivations, codelists, session/environment details), code provenance via shinymeta, and versioning of inputs and outputs. This will enable faster audits, easier re-analysis across cohorts, and deterministic, submission-ready artifacts.

LLM governance will be strengthened with curated prompt libraries, formal review paths, tighter context scoping, and automated provenance logging to ensure responsible and transparent AI integration.

We also plan to introduce bookmarks and state persistence to save/restore dashboard sessions. Users will be able to generate a shareable link or saved state that captures filters, subgroup definitions, module selections, and output

parameters. This improves collaboration, supports reproducibility, and shortens the path from exploration to standardized reporting.

Furthermore, we are evaluating LLM agents to support agentic workflows that automate routine tasks:

- Ingest a Programming Plan and extract analysis requirements and output mappings.
- Locate and load the necessary data, metadata, and codelists from AP with appropriate access controls.
- Enhance integration and automation into the APS end-to-end pipeline, by exposing iCSR functionality via API to a MCP server.

These enhancements are designed to balance speed and usability with rigor and auditability, ensuring iCSR continues to deliver compliant, reproducible insights while adapting to evolving user needs and regulatory expectations.

CONCLUSION

The transformation from static to interactive, standardized dashboards has had a profound impact on clinical reporting at Merck. Success has been driven by close collaboration between SMEs, statisticians, and developers, as well as by a commitment to strong design and validation standards. The willingness to iterate and adapt based on user feedback has ensured that the solution remains relevant and user-friendly. As AI-powered features become more prevalent, proactive governance and continuous validation will be essential to maintain trust and regulatory compliance.

The iCSR application stands as a testament to how modern web technologies and AI can elevate data advocacy in regulated environments. By combining modular Shiny architecture, React.JS components, and LLM assistance, the solution delivers faster insights, improved compliance, and a superior user experience. Ongoing investment in validation, accessibility, and governance ensures that the application will remain robust and adaptable to future needs.

Initial deployment demonstrated the potential to shorten time to deliver key statistics—contingent on full end-to-end pipeline implementation—and enabled non-technical users to generate outputs independently, reducing bottlenecks in the analysis pipeline.

ACKNOWLEDGMENTS

We thank the teams contributing to iCSR development, testing, and user feedback.

With special thanks to Yan Hao, Wayne Yang, Sukie Gao and Kyle Husmann.

CONTACT INFORMATION

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

Author Name: Richard Becker
Company: Merck Serono Ltd., an affiliate of Merck KGaA
Address: Feltham, Middlesex, TW14 8HA, UK
Email: richard.becker@merckgroup.com

Author Name: Markus Niederstrasser
Company: Merck Biopharma Co., Ltd., an affiliate of Merck KGaA
Address: 1-3-1 Azabudai, Minato-ku, 106-0041 Tokyo, Japan
Email: markus.niederstrasser@merckgroup.com