PharmaSUG China 2026 - Paper DV-162

Interactive Clinical Trial Data Review Using Non-Shiny R HTML Dashboards

Peng Tan Sanofi Corporation

ABSTRACT

Clinical programming teams are frequently asked to support data review and risk monitoring for studies that span dozens of countries and hundreds of sites, with source data stored in SAS datasets and review decisions tracked across multiple functions. Static tables and listings answer specific questions, but they rarely help teams see the bigger picture — which sites are driving most of the burden, whether the review backlog is growing or shrinking, or how this month's snapshot compares to the last one.

This paper describes a non-Shiny HTML dashboard approach built with R and JavaScript for protocol deviation (PD) data review. The approach uses R Markdown and flexdashboard to render a self-contained HTML file from SAS source data, with all chart rendering and user interaction handled client-side using Plotly.js and vanilla JavaScript modules. The output requires no server, no Shiny runtime, and no special viewer — it opens in any modern browser.

Beyond the basic architecture, the paper discusses how the dashboard handles multi-audience reporting needs: executive KPI views, site-level cross-filtering, country burden comparisons, editable review annotations, and spreadsheet upload against a fixed HTML shell. It also introduces a browser-based local LLM module that can generate natural-language summaries from pre-aggregated dashboard statistics, implemented using the MLC AI WebLLM library and running entirely offline through WebGPU. By executing inference locally on the user's device, this approach simultaneously addresses two persistent challenges in clinical AI adoption: data privacy — no patient-level or study-level information leaves the local machine — and infrastructure dependency — no cloud API keys, no server provisioning, and no network connectivity required at runtime.

The goal of this paper is not to advocate for any particular toolset, but to share a practical design approach that many R programming teams can adapt within their existing clinical workflow without adopting a hosted application platform.

INTRODUCTION

The data review cycle in a clinical trial produces a steady stream of questions that static deliverables handle poorly. A listing of protocol deviations tells you what happened; it does not immediately tell you where the concentration is, whether a site is an outlier relative to regional peers, or whether the current batch of new records represents an improvement over the prior cut. Interactive tools close this gap — but the question of how to build them in a validated, distributable, and practically maintainable way is not straightforward.

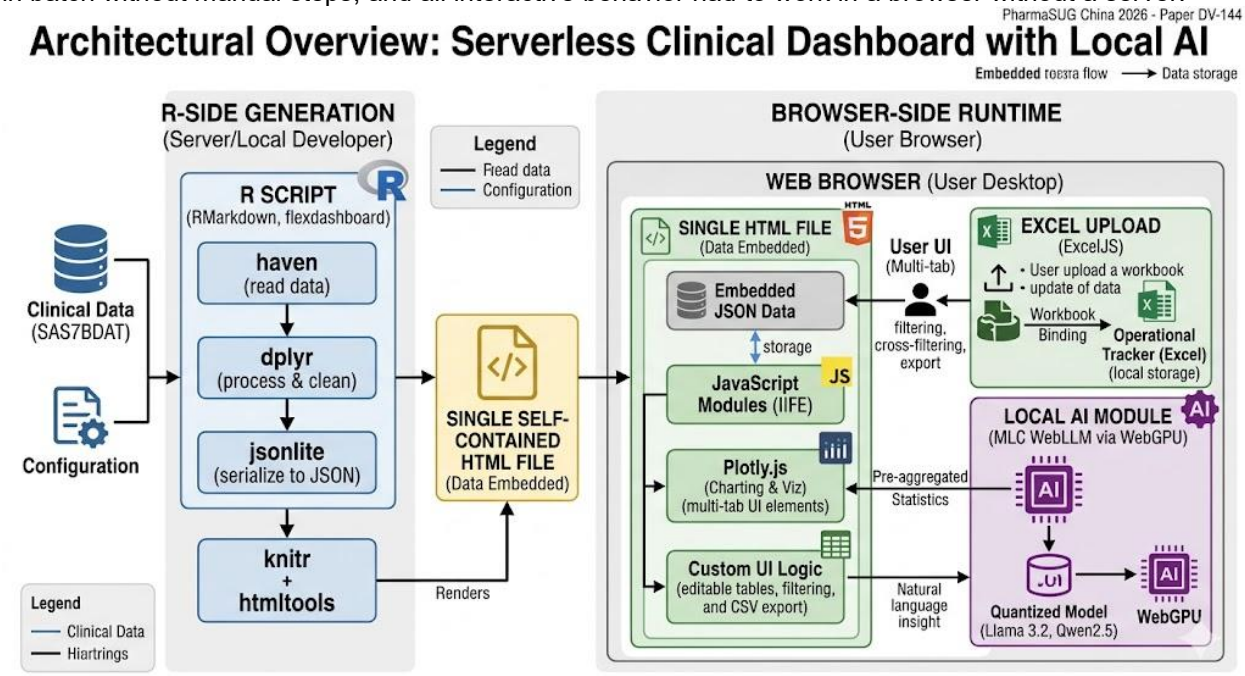
Shiny is the most commonly discussed interactive reporting framework in the R community, and it handles many of these needs well. In clinical programming environments, however, running a Shiny application in production requires a server, ongoing maintenance, IT involvement, and a clear position within the validated system landscape. For operational review tools that need to be shared quickly with study teams, regenerated after each data cut, archived at key milestones, and opened locally on different machines by users with different technical backgrounds, a simpler delivery model is often more practical.

The approach described in this paper produces a standalone HTML file. R handles the data processing — reading SAS datasets, deriving reporting fields, stabilizing encodings, and serializing the cleaned data to JSON — and the rendered HTML handles the rest. Once generated, the file can be emailed, placed on a shared drive, or archived at a study milestone with no dependency on the generation environment. The JavaScript layer inside the HTML reads the embedded JSON, builds charts, responds to filter interactions, and exports data on demand. From the end user's perspective, it behaves like a dashboard. From the programmer's perspective, it is a file they can regenerate with a single batch command.

This paper uses protocol deviation review as the motivating use case. PD monitoring is a good example because it involves multiple data dimensions (category, review status, source, geography), multiple audiences (data managers, medical reviewers, study leads, monitors), and a recurring review calendar that puts pressure on communication efficiency. The same design patterns, however, apply to any domain where teams need to move between aggregate signals and source records within a single reporting product.

TECHNICAL FRAMEWORK

Understanding the design requires a brief description of the technology stack. The choice of components was driven by practical constraints: the output had to be self-contained, the generation pipeline had to run in batch without manual steps, and all interactive behavior had to work in a browser without a server.



R-SIDE COMPONENTS

The dashboard is built using R Markdown (rmarkdown 2.30) with the flexdashboard package (version 0.6.2), which provides a tabbed layout framework suitable for multi-panel reporting. The YAML header configures the output as a single self-contained HTML file with scroll-based vertical layout and a Bootstrap theme. The knitr package (version 1.51) handles code chunk execution and HTML assembly during rendering.

Data ingestion uses the haven package (version 2.5.5) to read SAS7BDAT source files directly into R data frames, preserving SAS variable labels as attributes. Because SAS datasets may carry embedded character encodings that cause display problems in HTML, the pipeline applies explicit UTF-8 conversion on all character variables before downstream processing. The dplyr package (version 1.1.4) handles the derivation steps: flag variables for review classification and randomization status, category grouping, geographic assignments derived from country codes embedded in subject identifiers, and text cleaning for filter-compatible string values. The jsonlite package serializes the processed data frames to compact JSON strings that are injected into the HTML as JavaScript global variables at render time using htmltools. Additional utilities include scales for rate and count formatting.

Rendering is coordinated by a batch R script that handles library path configuration, package version checking, input file discovery, output path construction, and the rmarkdown::render() call. Version pinning for all key packages is enforced through remotes::install_version() to support reproducibility across

different generation environments. The rendered file embeds all chart libraries inline via the self_contained: true option, producing a single HTML artifact with no external dependencies at runtime.

JAVASCRIPT-SIDE COMPONENTS

All chart rendering, filtering logic, and user interaction are implemented in JavaScript and bundled into the HTML at render time. The charting library is Plotly.js (version 2.27.0), which handles stacked bar charts, pie and donut charts, indicator gauge charts, and dynamic ranked bar charts. Custom toolbar buttons extend the default Plotly mode bar with study-specific actions: filter popover controls, expand-to-modal for detailed chart review, and formatted CSV export.

The dashboard logic is organized into tab-specific IIFE (immediately invoked function expression) modules. Each module maintains its own filter state, chart references, and table pagination, avoiding variable naming conflicts across the multi-tab layout. Tab 1 manages cross-filtered deviation overview charts and a paginated editable review table. Tab 2 computes KPI gauge metrics replicating the logic of a SAS-derived deviation rate indicator, including projected and confirmed rate calculations over the randomized participant population. Tab 3 handles country and site level visualization with independent filter controls and a separate listing view. Each module listens for a custom pd-data-updated browser event that fires when the user uploads a replacement data file, allowing all charts and tables to redraw from the new dataset without reloading the page.

Excel upload and workbook binding use the ExcelJS library for in-browser workbook parsing. A validation layer checks for the presence of all 39 required columns before allowing the uploaded data to replace the embedded dataset. An optional File System Access API integration supports binding the dashboard to a specific workbook path on the local file system, enabling refresh and controlled write-back workflows for users who maintain an operational tracker alongside the rendered HTML. This binding capability can be enabled or disabled at deployment time depending on governance requirements.

For the experimental AI insight module described later, the MLC AI WebLLM library is used to run a quantized large language model locally in the browser via WebGPU.

FUNCTIONAL CAPABILITIES

For The dashboard is organized around the natural review workflow: start with the study-wide picture, drill into geographic or categorical concentrations, examine quantitative indicators, and review or annotate individual records.

MULTI-DIMENSIONAL FILTERING AND CROSS-FILTERING

The dashboard provides a rich set of interactive filters — deviation category, review status, confirmatory flag, data source, randomization status, country, site, and free-text search — that apply simultaneously across charts and tables. Charts and listing tables share a cross-filter state: clicking a bar segment or pie slice narrows the table to the records behind that visual signal. This allows users to move from an aggregate pattern to its source records in two clicks, without re-running any code or switching tools.

GEOGRAPHIC BURDEN COMPARISON

A region-country-site hierarchy view shows stacked deviation counts at each geographic level, allowing teams to identify which regions or sites contribute disproportionately to the overall burden. Independent filter states let a reviewer focus on a specific geography without affecting the study-wide view elsewhere in the dashboard.

EDITABLE REVIEW ANNOTATIONS AND CSV EXPORT

Selected fields in the record listing are editable — confirmatory status dropdowns, free-text review notes, tracking flags, and population assignment fields. These edits are local and exploratory, designed to support team coordination during review meetings rather than to serve as a system of record. Users can export the full filtered dataset or the current page to CSV at any time.

HISTORICAL COMPARISON AND TREND MONITORING

Each rendered HTML file from a data cut is a complete, self-contained archive. A comparison mode in the batch render script reads two SAS source files from different dates, derives a delta dataset identifying new records, resolved records, and changed review status, and injects this delta alongside the current snapshot. The dashboard can then highlight changes directly in the listing table or add trend layers to ranked bar charts, giving study teams visibility into whether performance is improving over time — all without adding complexity to the browser-side code.

IN-BROWSER DATA UPLOAD AND WORKBOOK BINDING

Between render cycles, study teams often work from updated operational Excel trackers. The dashboard supports in-browser data replacement: a user uploads a structured workbook, a validation layer checks for all required columns, and on success the embedded dataset is replaced in memory with a full redraw of all charts and tables — no page reload required. An optional File System Access API binding allows the dashboard to hold an open reference to a local workbook file, enabling controlled write-back of review annotations directly from the browser. Governance of this capability — what constitutes an authoritative record versus working notes — should be resolved before deployment.

EXPERIMENTAL BROWSER-BASED AI INSIGHT MODULE

One section of the dashboard implements an AI-assisted insight module that runs entirely in the browser without any server-side component. This capability is currently not exposed in the default production user interface, but it represents a technically meaningful extension of what the non-Shiny HTML approach can support, and it is described here to document the design boundary.

The module has two layers. The first is a rule-based natural language query engine that interprets common review questions — counts by category, top-N terms, comparison by status, filtered subsets — using a pattern-matching approach against the loaded dataset. This layer responds immediately with no model download required and handles the majority of simple lookup-style questions that arise during a review meeting.

The second layer uses the MLC AI WebLLM library to load a quantized language model — options include Llama 3.2 (3B), Qwen2.5 (1.5B or 3B), Phi-3.5 Mini, Gemma 2 (2B), or Mistral 7B — directly in the browser via the WebGPU API. When a user submits a question that the rule-based layer cannot answer, the query is passed to the local model along with a system prompt that contains pre-aggregated dashboard statistics: total deviation counts, category distributions, review status breakdown, country and site burden, top terms, and participant impact metrics. The model generates a natural-language response grounded in these statistics. Because the model runs locally and the statistics are pre-aggregated rather than record-level, no individual participant data is sent to any external service.

The primary practical constraint is model download size. First-use download ranges from approximately 1 GB to 4 GB depending on the selected model, which is impractical in network-constrained environments. For this reason, the module is implemented as a feature that can be enabled or hidden at deployment time depending on local network conditions.

The browser capability constraint, by contrast, has largely been resolved. As of early 2026, WebGPU is enabled by default in Chrome 113+, Edge 113+, and Firefox 128+ (covering over 85% of enterprise desktop browser installations according to StatCounter and Can I Use data). The W3C WebGPU specification reached Candidate Recommendation status in 2024, and all major browser vendors have committed to long-term support. In practice, this means that for most enterprise users on managed Windows desktops with discrete or integrated GPUs manufactured after 2020, WebGPU is available without any special configuration. The remaining gaps are primarily older Linux distributions and certain Citrix/VDI environments where GPU passthrough is not configured — environments that should be identified during deployment planning but that no longer represent the majority case.

The practical value, where it can be enabled, is that it extends the dashboard's insight layer beyond pre-programmed chart logic. A reviewer facing an unusual pattern in a specific region can ask the dashboard to describe what the statistics show, without requiring the programmer to have anticipated that specific question in the chart design. This is modest assistance, not a replacement for clinical judgment, but in a

fast-moving review cycle even a concise summary of what the data show in a given filter context can meaningfully reduce time to consensus.

ADVANTAGES AND LIMITATIONS

Compared to static TLF packages, the dashboard approach has clear practical advantages for operational review: faster access to geographic and categorical breakdowns, immediate drill-down without re-running code, and a single deliverable that serves multiple audience levels. Compared to Shiny, the lack of a server requirement simplifies distribution considerably and makes milestone archiving straightforward — the HTML file from a given data cut is a complete and self-contained record.

The limitations are real, however, and should be stated plainly.

Client-side data scale becomes a practical issue when the embedded JSON exceeds several hundred thousand records. For large global studies, it may be necessary to pre-aggregate some views server-side or limit the listing view to a filtered subset rather than the full dataset.

Governance of local edits requires explicit clarification before deployment. The upload and binding features described in this paper are designed to support collaborative review, not to create a new system of record. Organizations adopting these features should document clearly what constitutes an authoritative data source and ensure that dashboard annotations are treated as working notes rather than formal review decisions.

The dashboard is an operational review tool, not an audit trail system. This distinction matters in regulated contexts and should be stated explicitly in any deployment documentation. The dashboard does not log user actions, does not maintain a version history of edits, does not enforce access control at the record level, and does not constitute an electronic record under 21 CFR Part 11 or equivalent frameworks. Its purpose is to support human review decisions by presenting data in a navigable and filterable format — the authoritative record of those decisions sits in the downstream data management system, not in the dashboard itself. Teams that deploy this tool in a GCP context should include this scope statement in their applicable SOPs or user guidance to avoid ambiguity during audits or inspections.

Browser environment variability affects advanced features. The File System Access API is not available in all browsers or in enterprise environments where security policies restrict file system access from web pages. The WebGPU requirement for the AI module is not yet universally available. Deployment planning should include explicit browser and environment compatibility checks for any features that go beyond standard HTML and JavaScript.

There is also no direct equivalent of a validated SAS macro library for the JavaScript layer. Teams that extend the dashboard logic significantly will need to establish their own testing and review procedures for the browser-side code, which is a non-trivial investment in a regulated context.

FUTURE DIRECTIONS

Several extensions of this approach are worth pursuing for future development cycles

A structured snapshot comparison mode with delta annotations would make recurring review significantly more efficient. Rather than reading two dashboards in parallel, reviewers would see a single view that surfaces what changed.

Template configurations for different clinical domains — safety event monitoring, data query tracking, site performance review — would reduce the setup time for new studies. The underlying architecture is not PD-specific; the most study-specific components are the variable mapping and the category grouping logic, both of which can be externalized to a configuration file.

A multi-study aggregation view represents a natural extension of the single-study model. Rather than generating one HTML file per study, the batch render script could iterate across a portfolio of active studies, pre-aggregate each study's KPI metrics — total deviation count, confirmed important rate, unresolved backlog, randomized participant count — and inject this cross-study summary alongside the per-study detail data. The resulting dashboard would present a portfolio-level overview tab showing which studies are above threshold on key indicators, with drill-down navigation into individual study detail views.

The R-side logic for this pattern is straightforward: `purrr::map` over a study list, apply the same derivation pipeline to each, and bind summary rows into a single aggregated frame. The JavaScript layer requires only one additional tab module managing the portfolio table and its link behavior to per-study sections. For oversight functions such as data management leadership or clinical operations leads who monitor multiple concurrent studies, this view reduces the overhead of opening and interpreting a separate dashboard per study.

A packaged reporting mode that generates a short narrative PDF or Word summary alongside the HTML — drawing from the same pre-aggregated statistics used by the AI insight module — would close the gap between the interactive operational tool and the formal summary deliverables that study teams still need for milestone documentation.

The browser-based AI insight module, as browser hardware support for WebGPU becomes more consistent, may become a practical default rather than an optional feature. Smaller quantized models running in under 2 GB of GPU memory are already capable of summarizing structured statistical context with reasonable coherence.

CONCLUSION

The central practical argument of this paper is that a non-Shiny HTML dashboard represents a viable and underused option in the clinical programming toolkit. It does not require a server, it can be generated from a single batch script, it produces a portable file that any study team member can open, and it supports the kind of interactive filtering and drill-down that makes large deviation datasets genuinely reviewable rather than merely reportable.

The implementation described here — flexdashboard and R Markdown for rendering, haven and dplyr for data preparation, Plotly.js and custom JavaScript modules for interactivity, ExcelJS for upload handling, and WebLLM for optional AI assistance — reflects one working combination of tools. Other combinations are possible. The architectural choices that matter most are simpler: embed the data, keep the logic in the browser, make the generation reproducible, and design the tabs around how people actually use the data rather than around what is easy to implement.

Protocol deviation review was the motivating example here, but the same patterns are applicable across most recurring operational review workflows in clinical development. As the expectation for real-time, accessible, and cross-functional data communication grows in pharmaceutical and biotech settings, approaches that combine the reproducibility of batch programming with the accessibility of browser-based interaction will become increasingly relevant.

There is a broader implication worth stating explicitly. The skill set required to build this kind of deliverable — R data processing, JavaScript interaction design, HTML/CSS layout, browser API integration, and an understanding of clinical data governance — represents a meaningful expansion of what clinical programmers traditionally do. The R programmer who can produce this dashboard is not merely writing statistical reports; they are functioning as a full-stack clinical developer, bridging the gap between data engineering, front-end interaction design, and domain-specific regulatory understanding. This convergence is not accidental. As clinical development organizations expect faster turnaround, richer communication, and more self-service capability from their data products, the programmer who can deliver a complete interactive artifact — from SAS source to validated browser output — occupies an increasingly strategic position. For R programming teams considering their professional development trajectory, investing in JavaScript literacy, browser platform knowledge, and client-side architecture thinking is not a diversion from clinical programming — it is an extension of it into the territory where the highest-leverage operational impact now lies.

REFERENCES

Wickham H, Francois R, Henry L, Muller K, Vaughan D. dplyr: A Grammar of Data Manipulation. R package version 1.1.4. https://CRAN.R-project.org/package=dplyr

Xie Y, Allaire JJ, Grolemund G. R Markdown: The Definitive Guide. Chapman and Hall/CRC, 2018. https://bookdown.org/yihui/rmarkdown/

Iannone R, Allaire J, Borges B. flexdashboard: R Markdown Format for Flexible Dashboards. R package version 0.6.2. https://CRAN.R-project.org/package=flexdashboard

Wickham H, Hester J, Bryan J. haven: Import and Export SPSS, Stata and SAS Files. R package version 2.5.5. https://CRAN.R-project.org/package=haven

Ooms J. jsonlite: A Simple and Robust JSON Parser and Generator for R. R package version 1.8.9. https://CRAN.R-project.org/package=jsonlite

Sievert C. Interactive Web-Based Data Visualization with R, plotly, and shiny. Chapman and Hall/CRC, 2020. https://plotly-r.com

Plotly Technologies Inc. Plotly JavaScript Open Source Graphing Library, version 2.27.0. https://plotly.com/javascript/

ExcelJS Contributors. ExcelJS: Excel Workbook Manager. https://github.com/exceljs/exceljs

MLC AI. WebLLM: High-performance In-browser LLM Inference Engine. https://webllm.mlc.ai/

Mozilla Developer Network. File System Access API. https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API

SAS Institute Inc. SAS 9.4 Language Reference. https://support.sas.com/en/documentation.html

ACKNOWLEDGMENTS

I'd like to thank colleagues in clinical data management, medical review, and statistical programming for their engagement with operational review workflows and for the practical feedback that shaped the design described.

CONTACT INFORMATION

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

Peng Tan
Sanofi
39F, Building 3, No. 1199 North Section of Tianfu Avenue Chengdu High-tech District, Chengdu City, Sichuan, China
Matthew.tan@sanofi.com