Paper ML02

Enhancing Real-Time Clinical Reporting with R Shiny and Large Language Models: Automated Code Generation and Dynamic TLG Visualization

Mahesh Divakaran, Amity University, Lucknow, India

ABSTRACT

Clinical reporting demands speed, accuracy, and transparency. Traditional workflows rely on manual programming for Tables, Listings, and Graphs. This approach increases development time and error risk. You often depend on experienced programmers to translate specifications into code. This paper presents a framework combining R Shiny and Large Language Models to automate R code generation for clinical reporting. You upload standard clinical datasets and describe analytical needs using natural language. The system generates executable R code and renders interactive outputs in real time. A prototype application, PharmaTFL Agent, demonstrates the workflow using subject-level analysis data. Results show faster TLG development, clearer traceability, and improved interaction across clinical teams. The framework supports exploratory analysis, review workflows, and internal decision support.

Keywords

Clinical reporting. R Shiny. Large Language Models. TLG automation. Interactive dashboards.

1. INTRODUCTION

Clinical reporting sits at the center of clinical trial operations. You generate Tables, Listings, and Graphs from standardized analysis datasets to support safety review, efficacy evaluation, and regulatory communication. These outputs follow strict conventions. You rely on validated datasets, reproducible code, and traceable results. Despite mature standards, the execution layer remains manual and slow.

You typically begin with ADaM datasets prepared under CDISC guidance. Common examples include ADSL for subject-level data and ADAE for adverse events. You then write R or SAS programs to transform these datasets into outputs defined in TLG specifications. Each output requires repeated steps. You load libraries. You subset data. You summarize variables. You format tables. You define plots. You rerun code after every review comment. This workflow consumes time and increases operational friction.

R provides a rich ecosystem for clinical analytics. Core base R functions support data manipulation and control flow. Packages from the tidyverse, such as dplyr, tidyr, and ggplot2, support readable data pipelines and consistent visualization. ggplot2 enables layered graphics with explicit mappings between variables and aesthetics. Shiny extends R into an interactive environment. You define a user interface. You define server logic. You bind inputs to outputs. Shiny handles reactivity and rendering.

Despite these strengths, you still write all logic by hand. Shiny applications do not remove programming effort. They shift execution into a web interface. You still translate specifications into code. You still debug syntax errors. You still repeat patterns across studies and outputs.

Large Language Models introduce a new execution layer. These models convert natural language instructions into structured code. You describe an analytical task using plain text. The model predicts valid programming statements. In this framework, the model generates R code aligned with tidyverse and Shiny conventions. You review the code. You execute it. You retain control.

The system integrates external LLMs through secure API calls. Each request follows a defined sequence. You submit a prompt containing dataset context and user instructions. The application sends this prompt to the model endpoint. The endpoint returns structured text. The server parses this text as R code. Execution occurs inside the Shiny session. Logs capture success or failure.

The framework supports multiple model providers. OpenAI GPT models serve as primary engines for R code generation. These models handle syntax, function selection, and plotting logic. Gemini models from Google provide an alternative backend. The architecture treats models as interchangeable services. You configure model selection at runtime. This design avoids vendor lock-in and supports comparative evaluation.

Model prompts include domain constraints. You specify approved packages such as ggplot2, dplyr, and tidyr. You restrict file system access. You prevent external data calls. The generated code stays limited to in-memory datasets loaded by the application. This restriction supports governance and audit needs.

Pharmaverse standards guide domain alignment. Pharmaverse provides open-source packages designed for clinical reporting. Examples include admiral for ADaM dataset derivation and pharmaverseadam templates. These resources shape prompt structure and output expectations. You align generated code with community-reviewed patterns rather than ad hoc scripts. This alignment improves consistency across studies.

Training does not modify base model weights. The framework uses prompt engineering and contextual grounding. You inject examples following pharmaverse conventions. You describe variable naming rules. You define expected output structure. The model responds within these constraints. This approach avoids retraining risks and supports controlled behavior.

APIs manage all external communication. Requests include authentication headers and scoped permissions. Responses store no patient data beyond session memory. Execution remains local to the Shiny server. This separation reduces exposure risk and supports deployment in controlled environments.

The result is a layered system. R handles data and execution. Shiny handles interaction. LLMs handle code synthesis. Pharmaverse guidance shapes structure. You gain speed without losing visibility. You inspect every generated line. You decide what enters validated pipelines.

2. Related Work

2.1. Traditional Clinical Reporting

Traditional clinical reporting follows a linear and specification-driven workflow. You start with raw clinical trial data collected in electronic data capture systems. Data management teams clean and standardize these data. Statistical programmers then derive analysis datasets following CDISC ADaM standards. These datasets form the foundation for all downstream reporting.

You define Tables, Listings, and Graphs through detailed TLG specifications. These documents describe population flags, analysis variables, summary statistics, formatting rules, and footnotes. Programmers translate each specification into code. This translation relies on personal experience and institutional conventions. Minor differences in interpretation often lead to rework during review.

Programming environments remain dominated by SAS and R. You write scripts that load datasets, apply filters, calculate summaries, and format outputs. Each output exists as a standalone program or function. Reuse remains limited. Even similar tables across studies require manual adjustment due to study-specific metadata.

Validation represents a critical step. Independent programmers replicate outputs using separate code bases. You compare results cell by cell. Discrepancies trigger investigation and correction. This process improves confidence but doubles effort. Timelines extend as complexity grows.

Change management introduces further cost. Clinical teams frequently request updates during review cycles. You revise population definitions, grouping variables, or display formats. Each request requires code edits, reruns, and revalidation. Interactive exploration remains out of scope. Static outputs dominate communication.

Traditional workflows emphasize stability and compliance. You gain confidence through repetition and strict controls. You lose flexibility and speed. Early insight generation suffers. Exploratory questions wait until formal programming cycles complete.

These constraints motivate interest in systems that preserve transparency and reproducibility while reducing manual effort. The following sections examine how interactive tools and language-based code generation attempt to address these limitations.

2.2. R Shiny in Clinical Analytics

R Shiny provides a framework for building interactive analytical applications using R. You design a user interface using input controls such as dropdowns, sliders, and text fields. You define server logic to respond to user actions. Shiny manages reactivity and rendering. Outputs update when inputs change.

Clinical teams use Shiny for safety dashboards, enrollment tracking, and exploratory data review. You visualize adverse events, laboratory trends, and demographic summaries. Shiny supports rapid internal communication during trial conduct. Review teams interact with data without requesting new static outputs.

Despite these benefits, Shiny does not reduce programming effort. You still implement all transformations, summaries, and visualizations manually. Each new output requires new server logic. Changes in requirements require code modification and redeployment. Shiny improves access to results. It does not automate analytical development.

Governance remains manual. You review code through standard version control processes. You validate outputs outside the application. Shiny serves as a delivery layer rather than a productivity layer for programmers.

2.3. LLM-Based Code Generation

Large Language Models generate programming code from natural language input. You describe tasks such as data filtering, summarization, or plotting. The model predicts syntactically valid code using learned patterns. Data scientists use these models to accelerate exploratory analysis and visualization.

Most applications focus on general analytics and education. Use in regulated clinical settings remains limited. Concerns center on traceability, repeatability, and control over execution. Black-box behavior raises audit and compliance risks.

The framework presented in this paper addresses these concerns directly. You view all generated code before or after execution. The system records execution logs and errors. Generated code runs inside a controlled R environment using predefined packages and datasets. You retain decision authority over downstream use.

3. SYSTEM FRAMEWORK

3.1. OVERVIEW

The system supports automated generation of Tables, Listings, and Graphs from clinical datasets. You upload standardized SDTM or ADaM data. You define reporting intent through a structured promptbox. The system translates this intent into executable R code using Large Language Models. You review results, code, and logs within a single interface.

The system framework focuses on automated generation of Tables, Listings, and Graphs from standardized clinical datasets. You upload SDTM or ADaM data into a controlled session environment. The system reads dataset structure and metadata immediately. You define reporting intent through a promptbox instead of writing code. Each prompt describes population rules, variables, summaries, and display expectations.

After prompt submission, the system applies preset clinical reporting instructions. These instructions follow pharmaverse-aligned conventions and predefined package rules. You select the language model engine, including OpenAI GPT models or Gemini models. The system constructs a final prompt combining user intent, dataset context, and reporting constraints. An API call sends this prompt to the selected model. The model returns executable R code for the requested TFL.

The system displays generated code in a dedicated tab before execution. You trigger execution explicitly. The R engine runs code within the session using in-memory data only. The Result Workspace renders tables, listings, or graphs. You export outputs as PDF when needed. The log tab captures messages and errors. You revise the prompt and rerender outputs within the same session. This framework reduces manual programming while preserving transparency, traceability, and user control.

3.2. SYSTEM FLOW

Step 1. Data upload

- ⇒ You upload SDTM or ADaM datasets.
- ⇒ The system loads data into the session workspace.
- ⇒ Dataset metadata and variable names become available for prompt context.

Step 2. Promptbox definition

- ⇒ You write prompts in a structured promptbox.
- ⇒ Each prompt describes one TFL requirement.
- ⇒ Examples include population filters, summary statistics, and display format.
- ⇒ Prompts stay reusable across sessions.

Step 3. Prompt validation and enrichment

- ⇒ The system checks prompt structure.
- ⇒ Preset instructions add reporting rules and package constraints.
- ⇒ Pharmaverse-aligned conventions guide variable usage and output structure.

Step 4. Model selection and API call

- ⇒ You select the model engine.
- ⇒ Supported engines include OpenAI GPT models and Gemini models.
- ⇒ The system sends the enriched prompt through secured API calls.
- ⇒ No dataset leaves the execution environment.

Step 5. Code generation

- ⇒ The selected model generates R code.
- ⇒ The code follows tidyverse and clinical reporting conventions.
- ⇒ The system stores the code without execution.

Step 6. Code execution

- ⇒ You trigger execution from the interface.
- ⇒ The R engine runs the generated code in a controlled session.
- ⇒ Execution uses in-memory datasets only.

Step 7. Result rendering

⇒ The Result Workspace tab displays tables, listings, or graphs.
⇒ You interact with outputs immediately.
⇒ You download outputs in PDF format when required.

Step 8. Code inspection

⇒ The Code tab displays full generated R code.
⇒ You review logic and formatting.
⇒ You copy or export code for validated pipelines.

Step 9. Logging and rerendering

⇒ The Log tab captures messages, warnings, and errors.
⇒ You adjust the prompt directly when issues appear.
⇒ You rerender outputs without restarting the session.

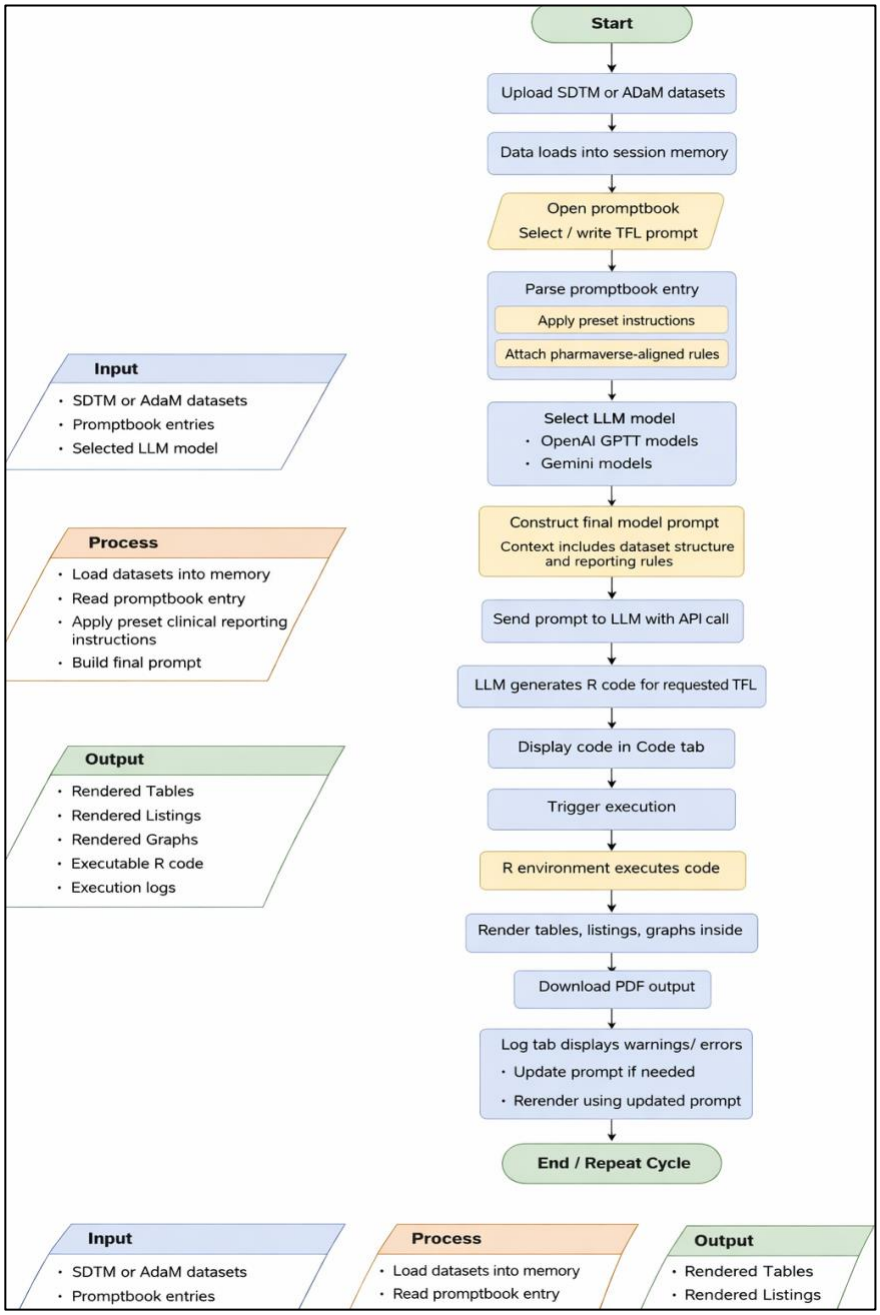

Figure 1: System Flow of the proposed Model

3.3. DATA INPUT

You upload analysis-ready clinical datasets such as ADSL or other ADaM or SDTM domains. The system loads these datasets into the Shiny session memory. Variable names, labels, and data structure become available immediately for downstream processing. Basic checks verify presence of required fields referenced in the promptbox. The dataset remains active across multiple prompt executions within the same session.

This approach avoids repeated data loading and reduces execution overhead. You work on a consistent data snapshot. All generated outputs trace back to the same in-memory dataset. This design supports repeatability during review and re-render cycles.

3.4. NATURAL LANGUAGE INSTRUCTIONS

You describe required outputs using clear natural language inside the promptbox. Each prompt defines one Table, Listing, or Graph. Example. Create a boxplot of age by treatment arm with minimal theme. You focus on intent rather than implementation details. The system interprets population rules, variables, summaries, and formatting cues from the prompt.

The selected language model converts the prompt into R code. The generated code includes library loading, data manipulation steps, plotting logic, axis labels, and themes. The system exposes the full code in the Code tab. You review logic before or after execution. This visibility supports audit review and controlled reuse.

3.5. EXECUTION AND OUTPUT

You trigger execution explicitly from the interface. The R engine evaluates generated code within the session environment. Execution uses only loaded datasets and approved packages. No external data access occurs during runtime.

Rendered outputs appear in the Result Workspace tab. Tables, listings, or graphs display immediately after execution. The Log tab captures messages, warnings, and errors. You assess execution status without leaving the application. When issues arise, you update the prompt and rerender outputs directly. This workflow shortens iteration cycles while preserving full transparency.

4. PROTOTYPE APPLICATION. PHARMATFL AGENT

4.1. CORE FEATURES

The PharmaTFL Agent serves as a working prototype for the proposed framework. You interact with a single Shiny-based interface designed for clinical reporting tasks. The application targets generation of Tables, Listings, and Graphs from standardized clinical datasets using prompt-driven automation.

You upload SDTM or ADaM datasets through the interface. The system validates file structure and loads data into session memory. Variable names and dataset context become available immediately for prompt execution. This step ensures consistency across repeated renders within the same session.

You submit analysis requests using text-based prompts stored in a promptbook. Each prompt defines one reporting task. The system processes the prompt, applies preset reporting instructions, and sends the enriched request to the selected language model through an API call. The model returns executable R code aligned with clinical reporting conventions.

The application executes generated code in a controlled R environment. The Result Workspace displays tables, listings, or graphs with interactive behavior where applicable. A dedicated Code tab exposes all generated R statements. A Log tab records execution messages and errors. You review results, inspect code, and diagnose issues without leaving the application.

The application also offers, you control every step. You define intent through prompts. You inspect generated code. You validate results through logs. This workflow replaces manual coding with prompt-driven execution while preserving full transparency and review control.

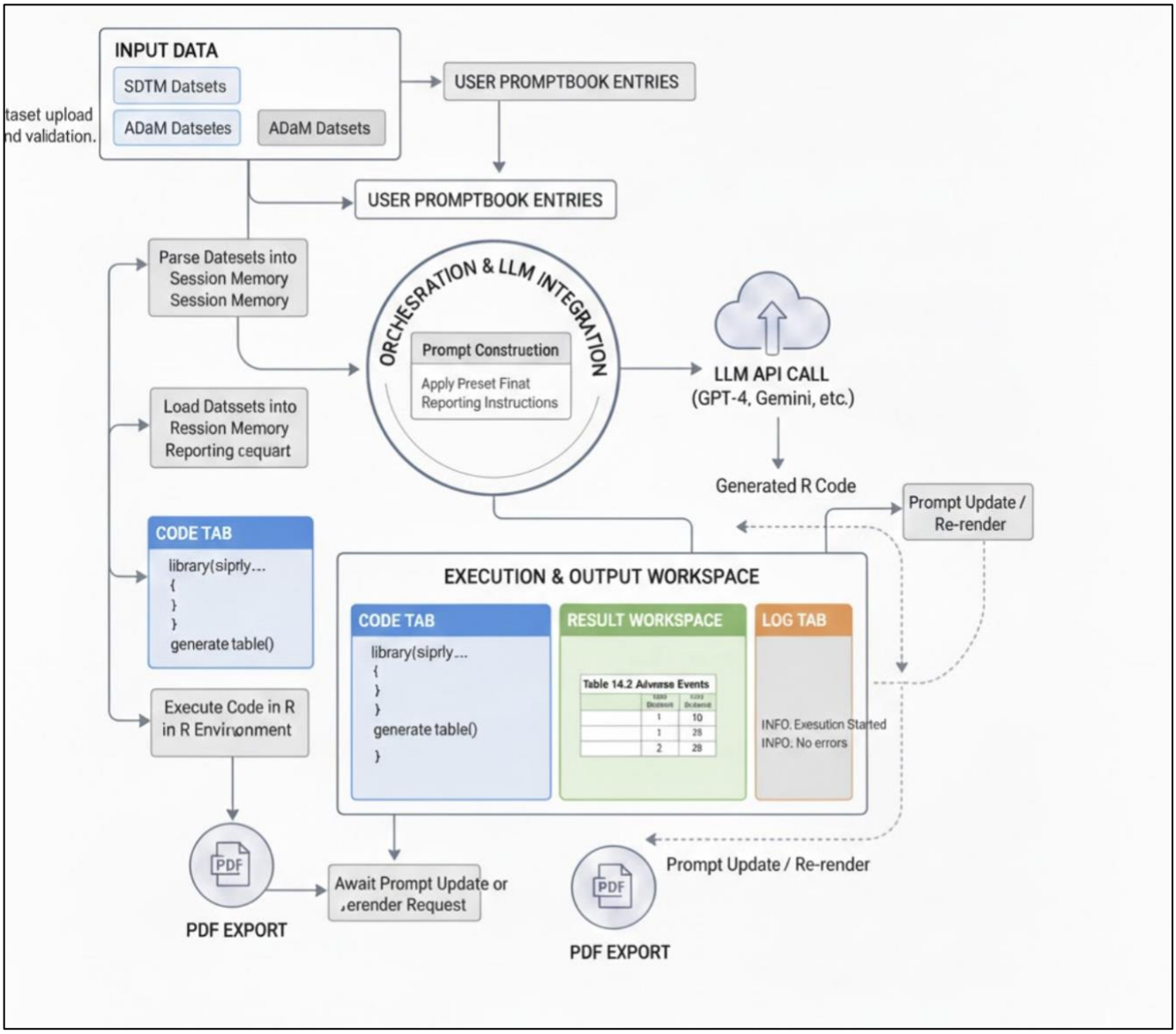

Figure 2: Core Features workflow

4.2. EXAMPLE WORKFLOW WITH TABLE AND FIGURE GENERATION

You start by uploading an ADSL dataset into the application. The system loads the dataset into session memory and scans variable names, types, and labels. Key subject-level variables such as AGE, ARM, SEX, and RACE become available for reporting. The dataset remains unchanged across multiple prompt executions, which ensures consistency during review and rerender cycles.

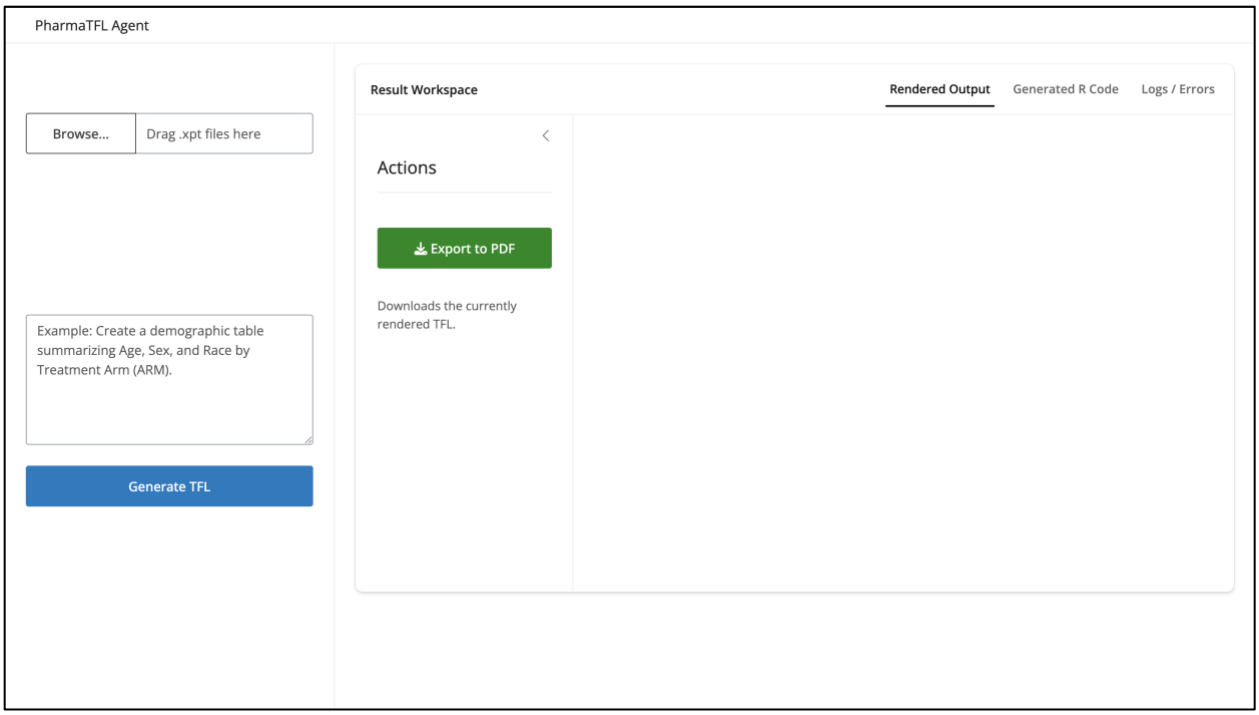

Figure 3: Basic App Interface

Example 1. Demographic Summary Table

You enter a structured prompt in the promptbook requesting a demographic summary table. The prompt defines column structure, row analysis, titles, and output assignment. You do not specify R syntax. You specify reporting intent.

The system parses the prompt and applies preset clinical reporting instructions. These instructions restrict package usage to approved libraries such as rtables and dplyr. The system enriches the prompt with dataset context, including variable availability and population assumptions such as ITT.

The enriched prompt is sent to the selected language model through an API call. The model generates R code using the rtables framework. The code defines column splits by treatment arm, adds an overall column, and calculates subject counts for column headers. Continuous summaries compute mean, standard deviation, median, and range for AGE. Categorical summaries compute counts and percentages for SEX and RACE. Titles and footers attach exactly as defined in the prompt. The final table object assigns to final_output.

You trigger execution manually. The R engine runs the code inside the session. The Result Workspace displays the formatted demographic table. The table lists treatment arms as columns and demographic variables as rows. The Code tab shows the full rtables pipeline. The Log tab confirms successful execution or highlights issues if present. You export the table to PDF for sharing or archive.

Example 2. Age Distribution Figure

You submit a second prompt requesting a graphical output. The prompt specifies a box plot of AGE by ARM with overlaid subject-level points. Formatting instructions define title text, axis labels, and minimal theme usage.

The system follows the same enrichment and API call process. The language model generates ggplot2 code. The code maps ARM to the x-axis and AGE to the y-axis. It defines boxplot geometry and adds jittered points to show individual subjects. Labels and theme settings apply as requested. The final plot object assigns to final_output.

You trigger execution. The Result Workspace displays the box plot grouped by treatment arm. Each box summarizes age distribution while jittered points show variability across subjects. The Code tab exposes the ggplot2 commands used for data mapping and styling. The Log tab confirms clean execution. You review the figure and download it as

a PDF or rerender after updating the prompt.

Figure 4: Example output

Across both examples, you control every step. You define intent through prompts. You inspect generated code. You validate results through logs. This workflow replaces manual coding with prompt-driven execution while preserving full transparency and review control.

4.3. TRANSPARENCY AND CONTROL

You maintain complete visibility of the analytical process at every stage. The application displays the full R code generated by the language model before and after execution. Each data transformation, summary calculation, and visualization step appears explicitly. No implicit logic runs outside the visible code path. This structure supports audit review and internal validation activities.

Execution requires your direct action. The system does not run generated code automatically. All messages, warnings, and errors appear in the Log tab with execution timestamps. You associate each output with the exact prompt, model selection, and code version used during generation. This linkage improves traceability during review discussions.

You control how outputs progress through the reporting lifecycle. Exploratory results remain within the application. Approved outputs and reviewed code move into validated production pipelines when required. This separation allows rapid iteration during analysis while preserving compliance expectations for final clinical reporting.

5. EVALUATION

5.1. DEVELOPMENT TIME

Evaluation focused on exploratory Tables, Listings, and Graphs created during internal review cycles. You compared manual programming workflows with the prompt-driven approach implemented in the prototype. In a traditional setup, you write R scripts for each output. You load data, define filters, calculate summaries, format results, and debug errors. This process typically requires several hours for a single demographic table or figure, especially when specifications change during review.

Using the proposed system, you complete the same tasks within minutes. You describe the output in the promptbook and trigger execution. When review comments arise, you update the prompt and rerender immediately. This rapid iteration supports live review meetings. You no longer wait for code updates between discussion rounds. The reduction in development time improves responsiveness and shortens feedback loops.

5.2. Error Handling

Manual clinical programming often introduces syntax and logic errors during repeated edits. Common issues include mismatched variable names, incorrect grouping, and formatting inconsistencies. In the proposed framework, standardized prompt enrichment and preset instructions reduce these risks. Generated code follows consistent patterns aligned with approved packages and reporting rules.

The application captures all execution messages in the Log tab. You see errors, warnings, and status messages immediately after execution. You diagnose issues without leaving the interface. When errors occur, you modify the prompt directly and rerender outputs. This immediate feedback shortens debugging cycles and reduces dependency on external tools or repeated script runs.

5.3. User Interaction

The interactive interface changes how clinical teams engage with outputs. Statisticians, clinicians, and reviewers view tables and figures directly in the application. You explore distributions, treatment comparisons, and formatting options in real time. You request alternate summaries or display changes without waiting for reprogramming.

This interaction supports collaborative review. You align interpretation and presentation during the same session. Exploratory questions receive immediate answers. The system encourages earlier insight generation while maintaining clear separation between exploratory outputs and validated production deliverables.

6. Discussion

6.1. Use in Clinical Workflows

The proposed framework fits naturally into existing clinical reporting workflows. You use the system during exploratory analysis, internal review, and pre-production stages. The framework does not replace validated production programming environments. It operates upstream of final submission deliverables.

You use the application to explore data, test display formats, and confirm interpretation with clinical teams. Prompts replace ad hoc exploratory scripts. Outputs remain traceable and reproducible within the session. Once review teams agree on content and presentation, you transfer approved logic and code into validated pipelines. This approach reduces rework and stabilizes downstream programming.

6.2. Limitations

The quality of generated outputs depends on the clarity and structure of prompts. Ambiguous instructions lead to inconsistent code generation. Prompt governance and standardized promptbooks remain necessary. Review of generated code remains mandatory.

The current prototype focuses on descriptive tables and visualizations. Complex inferential analyses, multipage tables, and advanced statistical outputs require additional safeguards and validation logic. Model behavior remains probabilistic. Controlled execution environments and explicit user approval mitigate risk but do not eliminate the need for oversight.

6.3. Future Work

Future development will extend support to full TLF suites across multiple domains. Integration with CDISC metadata repositories will enable automated variable discovery and rule enforcement. Version tracking will link prompts, code, and outputs across sessions. Enhanced audit logs will support inspection and review.

Deployment in secure enterprise environments remains a priority. Planned work includes role-based access, controlled API routing, and on-premise or private cloud deployment. These enhancements will strengthen governance while

preserving the efficiency gains demonstrated in this study.

7. CONCLUSION

This paper describes a Shiny-based system designed to support automated generation of Tables, Listings, and Graphs from standardized clinical datasets using Large Language Models. You upload SDTM or ADaM data, define reporting intent through a structured promptbook, and select a language model for code generation. The system translates these inputs into executable R code, executes the code in a controlled session, and renders outputs within an interactive workspace.

The PharmaTFL Agent prototype demonstrates clear practical value during exploratory analysis and internal review. You reduce development time by removing repetitive manual coding steps. You iterate on table layouts and figures during live review discussions. The system exposes all generated code and execution logs, which supports traceability, review, and internal quality checks.

The framework strengthens collaboration across statisticians, clinicians, and reviewers. You interact directly with outputs and respond to questions immediately. Exploratory results remain separate from validated production deliverables. Approved logic moves forward with confidence into formal programming pipelines. This approach improves upstream efficiency while preserving the rigor required for clinical reporting.

REFERENCES

1. Chang, W., Cheng, J., Allaire, J. J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., & Borges, B. (2024). Shiny: Web application framework for R. R Foundation for Statistical Computing. https://shiny.posit.co
2. CDISC. (2023). Analysis Data Model (ADaM) implementation guide, version 2.1. Clinical Data Interchange Standards Consortium. https://www.cdisc.org
3. CDISC. (2023). Study Data Tabulation Model (SDTM) implementation guide, version 3.4. Clinical Data Interchange Standards Consortium. https://www.cdisc.org
4. Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., & Yutani, H. (2019). Welcome to the tidyverse. Journal of Open Source Software, 4(43), 1686. https://doi.org/10.21105/joss.01686
5. Wickham, H., & Grolemund, G. (2017). R for data science: Import, tidy, transform, visualize, and model data. O'Reilly Media.
6. Morgan, M., & Anderson, J. (2023). rtables: Reporting tables for clinical trials in R. R Foundation for Statistical Computing. https://cran.r-project.org/package=rtables
7. Pharmaverse. (2024). Pharmaverse: Open-source clinical reporting ecosystem. https://pharmaverse.org
8. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., … Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.
9. OpenAI. (2024). GPT-4 technical report. https://openai.com/research
10. Google DeepMind. (2024). Gemini: A family of multimodal language models. https://deepmind.google/technologies/gemini/
11. Food and Drug Administration. (2023). Study data technical conformance guide. U.S. Department of Health and Human Services. https://www.fda.gov
12. OpenAI. (2025). ChatGPT (GPT-4o, 2 July version) Large language model. (https://chat.openai.com/)

CONTACT INFORMATION

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

Author Name: Mahesh Divakaran
Company: Amity University, Lucknow.
Address:
Work Phone: +91 8089686911
Email: mahesh.d@s.amity.edu
Website: amity.edu