Clarity: An AI-Powered Shiny Application for Natural Language-Driven Clinical Data Analysis

Jinzhi Zhou, Yusi Liu, Dan Su, Jie Liu
BeOne Medicines

ABSTRACT

Exploratory data analysis in clinical trials often demands both deep statistical expertise and strong programming skills, creating a significant bottleneck for cross-functional teams. Clarity (Clinical Analysis & Reporting Interactive Tool with You) is an AI-powered R Shiny application designed to eliminate this barrier by enabling pharmacometricians, biostatisticians, and clinical programmers to perform complex analyses through plain-language instructions — no code required.

Clarity integrates large language models (LLMs), including GPT and DeepSeek, through an enterprise LLM gateway to translate natural language requests into executable R code. Users upload clinical datasets (SAS7BDAT, CSV, XLSX) and describe their analytical goals in English or Chinese; Clarity generates, displays, and executes R code in real time, returning publication-ready plots, tables, and summaries within a unified chat interface.

Key capabilities include domain-specific skill injection for survival analysis (Kaplan-Meier curves), propensity score matching (PSM/IPTW), efficacy and safety summaries, forest plots, and data cleaning. The selected LLM and token usage are displayed transparently with each response. All generated code is fully visible, editable, and downloadable, supporting reproducibility and auditability. An automated reporting pipeline exports results as HTML or PowerPoint presentations.

Clarity demonstrates that conversational AI can serve as a practical coding assistant in pharmaceutical data science, dramatically accelerating exploratory analysis while preserving complete transparency and reproducibility of the underlying R code.

Keywords: large language models, R Shiny, clinical data analysis, natural language interface, skill library

INTRODUCTION

Clinical programming involves a large volume of repeatable, pattern-driven tasks: constructing Kaplan-Meier curves, building baseline characteristics tables, applying population filters, generating forest plots, adjusting chart formatting. These tasks are essential but largely mechanical — they follow well-established templates and add limited value that requires human judgment. Regardless of programming

experience, these routine tasks consume a disproportionate share of analyst and programmer time that could be redirected toward higher-value scientific work.

Cross-functional teams supporting Medical Affairs (MA) and Biomarker research face this bottleneck acutely. Scientists in these teams understand the analytical question clearly — "What does the OS curve look like stratified by this biomarker subgroup?" — but translating that intent into code, formatting the output to team standards, running quality checks on the data, and packaging results into a report involves many intermediate steps. Clarity (Clinical Analysis & Reporting Interactive Tool with You) is designed to handle those intermediate steps.

The pharmaceutical industry is simultaneously managing a growing R programming ecosystem alongside established SAS workflows. Even for teams proficient in R, re-implementing the same analytical patterns across studies and datasets represents avoidable repetition. The team behind Clarity has accumulated years of coding patterns, parameter conventions, and chart formatting standards from supporting MA and Biomarker analysis. The goal of Clarity is to capture this institutional knowledge in a reusable, AI-accessible form — turning past experience into an intelligent foundation for real-time analysis.

Several PharmaSUG 2026 US papers explore LLM integration in clinical programming. Rowsell et al. (AI-328) demonstrate that LLMs can operate as backend analytical services within R Shiny, using developer-authored prompts to generate TLF code from annotated shells. Pfizer's RWD Output Chat application (AI-339) constrains LLM outputs to a validated function library using retrieval-augmented generation. Merck's framework (AI-332) deploys LLM-assisted ADaM code transformation under human-in-the-loop oversight.

Clarity is distinguished from these systems by its target user and architectural approach. Where AI-328 and AI-339 serve clinical programmers producing deliverable TFLs, Clarity serves a broader team — including MA/Biomarker scientists who need rapid, interactive data exploration and programmers supporting those teams. Rather than RAG or a single large system prompt, Clarity employs a *skill library*: pre-authored, analysis-specific prompt templates that inject the complete analytical specification for each recognized task type. This approach provides tighter behavioral guarantees for the finite, well-defined analysis types that dominate clinical data exploration.

This paper describes Clarity's architecture, its skill library design, the code generation and execution loop, its data security and code safety approach, and lessons learned from production use.

SYSTEM ARCHITECTURE

Overview

Clarity is an R Shiny application deployed on Posit Connect. Users interact through a unified chat interface: they upload clinical datasets (XPT, SAS7BDAT, CSV, or XLSX), describe their analytical goal in plain language, and receive executable R code that runs immediately within the server session. Rendered tables (`table_result`) and plots (`plot_result`) appear directly in the browser alongside the code that produced them. Session history is persisted per user, enabling analysts to return to prior conversations.

The architecture has three layers:

1. **Frontend (Shiny UI):** A grouped sidebar (Data Setup, Analysis Tools, Output, Sessions), a model selector, a chat panel, a syntax-highlighted code viewer, an output panel for table and plot preview, and a per-response usage badge. The overall layout is shown in Figure 1.
2. **Agent Core (R):** Message routing, skill selection, system prompt assembly, LLM API dispatch, code extraction, iterative error correction, and output rendering.
3. **LLM Backend:** Two enterprise-hosted models — DeepSeek V4 Pro (default) and GPT-54 — accessed via HTTPS through an internal API gateway. All credentials are stored in server-side environment variables.

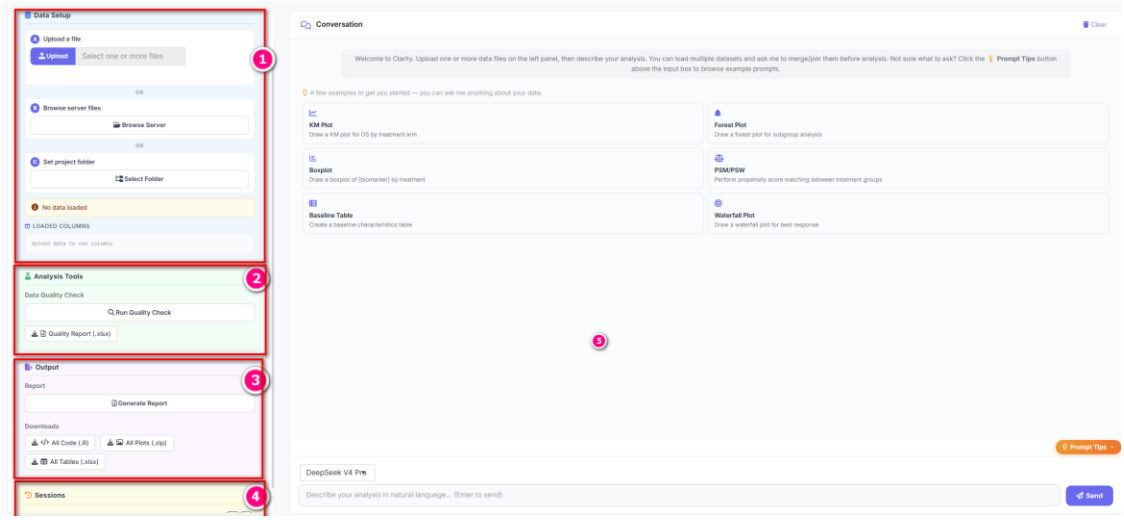

Figure 1. Clarity main interface: the left sidebar groups functions into Data Setup (A/B/C data loading options), Analysis Tools, Output, and Sessions. The right-side conversation panel displays quick-start analysis cards on first load.

Model Selection

Clarity defaults to DeepSeek V4 Pro, which handles both English and Chinese analytical requests and offers a favorable cost profile for the team's usage patterns. Users can switch to GPT-54 at any time via the model selector displayed above the

input box in the chat interface. Both models are accessed through the same enterprise API gateway and support the full range of analysis types in the skill library.

DATA SECURITY AND CODE SAFETY

Data security and safe code execution are core design requirements for a clinical analysis tool, not optional features.

Patient Data Protection

Patient-level data is never transmitted to any LLM API. When a user requests an analysis, the prompt contains only: dataset names, column names with R data types, and SAS variable labels. The actual cell values remain in the R server environment throughout the session. This design, consistent with the approach described by Rowsell et al. (AI-328), is compatible with standard pharmaceutical company data governance policies for internal API deployments.

Code Execution Safety

Before executing any LLM-generated R code, Clarity applies a pre-execution safety check against a blacklist of prohibited operations. This check blocks:

- File system operations: deleting, moving, or overwriting files (`file.remove()`, `unlink()`, `file.copy()` to sensitive paths)
- System command execution (`system()`, `shell()`, `system2()`)
- Network access from within generated code (`download.file()`, outbound `httr` calls)
- Process control (`Sys.sleep()` with excessive durations, `quit()`)

If a generated code block contains any blacklisted operation, execution is halted and the user is informed before anything runs. This ensures that even if an LLM produces unexpected output, it cannot take destructive or data-exfiltrating actions against the server environment. The safety check runs deterministically on every code block, independent of which LLM or model version generated it.

Enterprise LLM gateway providers such as Azure OpenAI also apply their own content filtering at the API layer. In practice, these filters occasionally flag clinical terminology — toxicity grade descriptions, adverse event narratives — as policy violations. When this occurs, Clarity retries with a reduced clinical context and falls back to DeepSeek if Azure responses consistently fail. This external filtering is a secondary layer; Clarity's own pre-execution safety check is the primary protection mechanism.

THE SKILL LIBRARY

Motivation

A naive single-system-prompt approach — "You are a clinical data analyst, write R code" — produces results that vary with how the user phrases the request, and is prone to generating plausible-looking but incorrect code when domain-specific conventions (CDISC ADaM dataset structure, PARAMCD filter values, censor coding) are involved. LLM-generated code errors arise from multiple sources: insufficient dataset context, ambiguous user requests, unfamiliar package conventions, and missing domain knowledge about standard analytical parameters.

Clarity addresses the domain knowledge dimension through a **skill library**: a directory of plain-text files, one per recognized analysis type, each containing the complete analytical specification for that task. When the routing layer identifies a match, the corresponding file is injected into the system prompt before the user's request is processed. Combined with dataset metadata and conversation history, this gives the LLM the full context it needs to generate reliable, standards-compliant code.

The skill library encodes the team's institutional knowledge: which packages to use, which parameter defaults match house standards, how to handle common edge cases in clinical data, and what output variable names the downstream rendering pipeline expects.

Skill File Structure

Each skill file follows a consistent format:

```
## Skill: [Name]

Apply when: [routing condition — keywords or patterns]

---

### Approach
[Step-by-step analytical approach following CDISC ADaM dataset conventi
ons]

### Code Template
[Canonical R code skeleton with placeholder column names]

### Rules
[Hard constraints: NA handling, column name policy, output variable nam
es]
```

For example, the `km_analysis` skill specifies that `adtte` is the expected dataset with `PARAMCD`, `AVAL`, `CNSR`, and `TRT01P`; defines the `survfit()` call structure and `ggsurvplot()` parameter defaults aligned with team standards; requires

`plot_result` as the output variable name; and defines edge-case handling for single-arm studies and missing censor indicators.

Routing

Routing is a two-stage process. First, a YAML configuration file maps surface-level patterns — "KM", "Kaplan", "OS", "PFS", "survival" → `km_analysis` — providing fast, deterministic matching for common requests. Second, for ambiguous requests, an LLM classifier receives the list of available skill names and trigger conditions and returns a routing decision, handling paraphrasing and mixed-language input.

The routing layer selects exactly one skill per request. Injecting a single, precisely matched skill template avoids context conflicts that arise when multiple analytical specifications are combined in a single prompt — for instance, simultaneously providing KM and longitudinal plot conventions for a request mentioning "time" would introduce contradictory guidance. Single-skill injection ensures the LLM receives unambiguous, task-specific instructions.

Coverage

The current skill library covers: KM survival analysis, waterfall plot (tumor response), swimmer plot (treatment duration), spider plot (tumor size change from baseline), forest plot (subgroup hazard ratio and odds ratio), boxplot (biomarker distribution), longitudinal line/error-bar plot, PRO domain bar chart, PRO MMRM analysis, propensity score matching and IPTW, Sankey/alluvial diagram, PK concentration-time analysis, baseline characteristics table (Table 1), and a general analysis fallback.

CODE GENERATION AND EXECUTION LOOP

Generation

When the agent core dispatches a request, the LLM receives a system prompt comprising: global role definition and output variable naming conventions; the injected skill template for the matched analysis type; dataset metadata (all column names with R types and SAS variable labels); and the conversation history from the current session. The LLM is instructed to return a single R code block with no surrounding prose. Figure 2 shows a complete end-to-end interaction, from the user's request through skill detection, prompt assembly, code generation, and the rendered result.

Figure 2. Example interaction: the user submits a KM analysis request (①); Clarity detects the KM Analysis skill and displays it as a badge; the system prompt sent to the LLM is shown in a collapsible panel (②); the generated R code is displayed with model attribution and token/cost badge (③); the rendered Kaplan-Meier curve appears below with a real-time styling panel for font and size adjustments (④).

Iterative Error Correction

Generated code is immediately executed in an isolated R environment. If execution fails, the exact error message is captured and returned to the LLM along with the failing code, requesting a targeted fix. Clarity repeats this automatic correction cycle until the code runs successfully or the issue is determined to require user input. The majority of runtime failures — type mismatches, incorrect filter values, missing package calls — are resolved automatically without the user needing to intervene. When a failure cannot be resolved automatically, Clarity surfaces the unresolved error and requests clarification, keeping the analyst informed throughout.

Multi-Step Context Retention

Conversation history is maintained across turns within a session, enabling analysts to build analyses incrementally: *"Draw a KM plot for OS by treatment arm"* → *"Add the log-rank p-value"* → *"Now restrict to ECOG 0–1 patients"* → *"Generate a report with this result."* Each instruction updates only the relevant portion of the code; the full analytical context carries forward automatically.

Truncation Detection

Long responses occasionally hit output token limits mid-generation. Clarity detects truncation by checking for unclosed code fences and absent terminal punctuation, then requests the remainder via a continuation prompt. This is transparent to the user.

ADDITIONAL CAPABILITIES

Automated Data Quality Check

Before analysis, users can run a six-check automated data quality assessment covering: missing values per column, outlier detection, mixed-language variable labels, type mismatches, duplicate rows, and duplicate columns. Results are exported as a color-coded Excel workbook. This is particularly useful for external datasets received from partner institutions or hospitals, where data quality and format cannot be assumed.

Prompt Dictionary

Users uncertain how to phrase a request can access the built-in **Prompt Dictionary**: a structured catalog of example prompts organized by analysis category (Survival, Efficacy, Forest, Boxplot, PRO, PSM, Tables, and more), with English and Chinese variants and explicit column name placeholders such as `[TRT01P]`, `[AVAL]`, and `[PARAMCD]='OS'`. The dictionary is accessible via a floating panel above the input box and requires no navigation away from the chat.

Every complex analysis type includes a template with the minimum necessary parameter specification. For example, rather than *"Draw a forest plot"* — which reliably produces hallucinated subgroup column names — the template reads *"Draw HR forest plot for OS by [SEX], [AGEGR1], [ECOG01] using adtte where PARAMCD='OS'"*, providing the LLM with the column names and dataset context it needs to generate correct code (Figure 3).

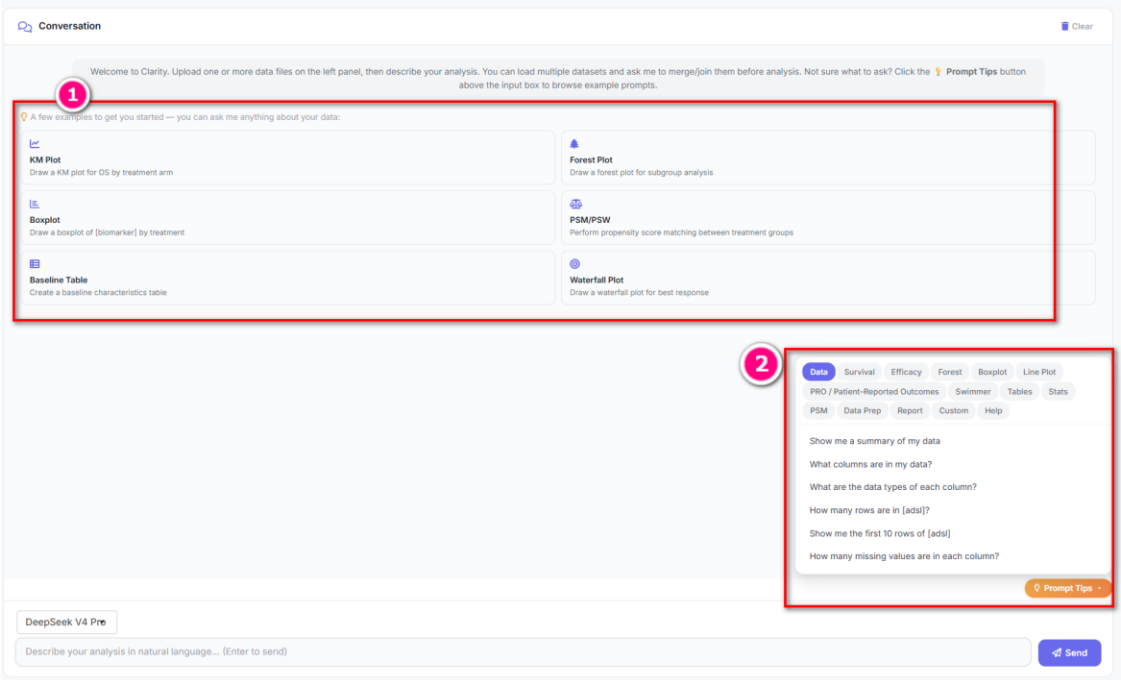

Figure 3. The Prompt Tips floating panel provides category-organized example prompts (Data, Survival, Efficacy, Forest, etc.) with ready-to-use templates that include explicit column name placeholders, reducing the need for users to recall variable names.

Downloadable Outputs

Every analysis produced in a session can be exported for offline use. The Output panel collects all executed code, rendered plots, and tables, and provides one-click downloads: all generated R code as a single `.R` script, all plots as a `.zip` archive, and all tables as an `.xlsx` workbook (Figure 4). Because the full R code is downloadable and runnable outside Clarity, every result remains fully reproducible and auditable independent of the application.

Figure 4. After analysis, the Output panel provides one-click downloads of all generated R code (①), all plots as a zip archive (②), and all tables as an Excel workbook (③). The example shows a swimmer plot with a real-time color and styling panel on the right.

Automated Reporting

At any point during a session, the user can request a report. Clarity packages all executed code blocks, rendered plots, and tables into an HTML document and a PowerPoint slide deck, providing a reproducible record of the exploratory session for sharing with clinical teams or inclusion in internal documentation. The generated report can be opened in a new browser tab or downloaded directly (Figure 5).

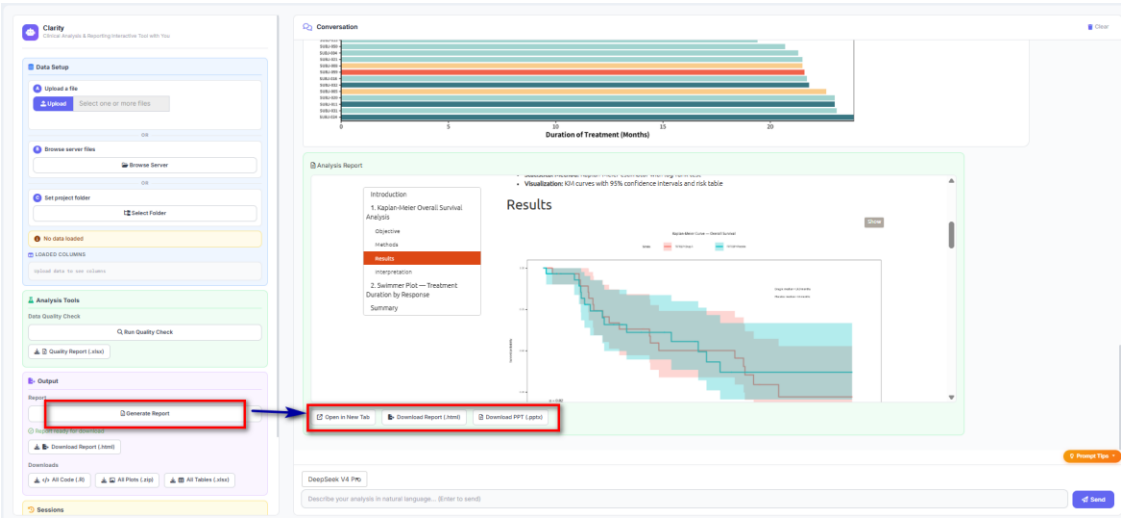

Figure 5. Clicking "Generate Report" assembles all session results into a structured analysis report — with introduction, objective, methods, results, and interpretation

sections — that can be opened in a new tab, downloaded as HTML, or exported as a PowerPoint deck.

Usage Transparency

Each LLM response displays the model used, estimated token count, and cost. For teams deploying Clarity across multiple users, this provides immediate visibility into query complexity and supports internal cost allocation.

LESSONS LEARNED

Skill injection improves code reliability for domain-specific analyses. Providing the LLM with the complete analytical specification for a recognized task — which dataset structure to expect, which R functions and parameters to use, which edge cases to handle — consistently produces more reliable code than relying on a general prompt. Code generation failures arise from multiple sources: insufficient dataset context, ambiguous user requests, and missing domain knowledge. The skill library addresses the domain knowledge dimension; the prompt dictionary and dataset metadata address the others.

Single-skill injection produces cleaner results than multi-skill contexts. When a user request could match multiple analysis types, resolving the ambiguity at the routing stage and injecting exactly one skill produces cleaner, more focused code than injecting multiple partially-matching templates. Providing contradictory conventions simultaneously — for instance, KM and longitudinal plot specifications — leads to confused outputs that combine elements from both. The routing layer's single-skill constraint is a deliberate design choice that reflects the principle of progressive disclosure: surface the most specific, relevant context rather than all potentially applicable knowledge.

Automatic error correction covers the majority of runtime failures. The iterative error correction loop — which feeds the exact error message and failing code back to the LLM for a targeted fix — handles most common failures transparently, without the user needing to intervene. Failures that cannot be resolved automatically indicate an ambiguous or structurally inconsistent request that benefits from user clarification.

Prompt dictionary specificity directly determines user success rate. The prompt dictionary's most important design property is that every template for a complex analysis includes the minimum necessary parameter specification: dataset name, key variable names, population filter, and grouping variable. Templates lacking this specificity lead users to submit underspecified requests that the LLM must guess at, producing unreliable results. Specificity in example prompts translates directly into specificity in user requests.

FUTURE DIRECTIONS

The current deployed version of Clarity focuses on conversational exploratory analysis supporting MA, Biomarker, and programming teams. Several directions are under early exploration.

Real-time TFL Quality Control. A future capability of Clarity could be interactive, on-demand TFL quality control — independently reproducing a production table within the chat interface and comparing it against the reference output in real time.

Flexible TFL Generation from Earlier Data Sources. Traditional TFL programming starts from ADaM datasets. Clarity's conversational, skill-driven architecture is inherently flexible about its starting point, and we expect future versions to generate TFLs from more upstream data sources — guiding the path from raw or SDTM-level data through derivation to the final TFL within a single session. This complements industry work such as Merck's SDTM Engine (Zhang et al., DS-440) and the SAP-reading agent described by Kimura et al. (AI-206).

Ad-hoc Table Generation. Analysts frequently need formatted tables not defined by approved shells — subgroup explorations, internal summaries, slide-ready outputs. Extending the skill library with `flextable`-based output templates would serve this need without conflating exploratory outputs with validated production deliverables.

CONCLUSION

Clarity demonstrates that a constrained, skill-library architecture enables reliable LLM-driven clinical analysis in a production R Shiny deployment. By pre-defining the complete analytical approach for each recognized task type and injecting it at routing time, the system achieves a practical balance between flexibility and reproducibility. All generated code remains visible, editable, and downloadable, ensuring that the analyst retains full control and the analysis remains reproducible and auditable.

The system is deployed and actively used by MA, Biomarker, and programming teams. The key technical contributions — skill library routing, pre-execution code safety checks, automatic error correction, and a structured prompt dictionary that guides users toward well-specified requests — are transferable patterns for other clinical programming teams building production LLM integrations in R.

Clinical programming teams need tools that reduce the time between analytical intent and executable result, without sacrificing transparency, reproducibility, or data security. Clarity's approach — using accumulated team expertise encoded in a skill library, executed against the analyst's own clinical dataset (ADaM, CSV, or external partner data) in a secure server environment — addresses these requirements in a form ready for production use.

ACKNOWLEDGMENTS

The authors thank colleagues in the MA, Biomarker, and Statistical Programming teams who contributed use cases, tested early versions of Clarity, and provided the domain expertise encoded in the skill library.

REFERENCES

1. Rowsell T, Jani D, Thampi N, Xu H. "Leveraging LLMs in Data Science Web Applications: Beyond the Chat Interface." *PharmaSUG 2026 US*, Paper AI-328. AstraZeneca PLC. 2026.

2. "Automating Table Generation in Real-World Data Programming: An AI-Assisted Shiny Application." *PharmaSUG 2026 US*, Paper AI-339. Pfizer Inc. 2026.

3. Xia C, Du F. "A Human-in-the-Loop AI-Assisted Framework for ADaM Standardization." *PharmaSUG 2026 US*, Paper AI-332. Merck & Co., Inc. 2026.

4. Kimura T, Wang S, Du W, Xie S. "AI Reading and Understanding the SAP to Generate the TFL TOC." *PharmaSUG 2026 US*, Paper AI-206. Arcsine Analytics / Alnylam / Regeneron. 2026.

5. Zhang LX, Lanoue J, Nielsen U. "From Specification to SDTM at Speed: Deploying the SDTM Engine in Production." *PharmaSUG 2026 US*, Paper DS-440. Merck & Co., Inc. 2026.

6. CDISC. "Analysis Data Model (ADaM) Implementation Guide v1.3." Clinical Data Interchange Standards Consortium. 2022.

7. Wickham H, et al. "Welcome to the tidyverse." *Journal of Open Source Software*, 4(43), 1686. 2019.

8. Chang R, et al. "shiny: Web Application Framework for R." R package version 1.7.5. 2023.

CONTACT INFORMATION

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

Jinzhi Zhou: jinzhi.zhou@beonemed.com

Yusi Liu: yusi.liu@beonemed.com

Dan Su: dan1.su@beonemed.com

Jie Liu: jie1.liu@beonemed.com