# Automating Table Generation in Real-World Data Programming: An AI-Assisted Shiny Application

*Date: February 27, 2026*


## Abstract

Real-world data (RWD) programming in pharmaceutical research involves generating standardized statistical tables for outcomes research (OR). The Pfizer RWD programming team developed a framework of R functions for creating consistently formatted tables. While this framework standardizes workflows, programmers still manually construct complex function calls for each table. This paper describes RWD OR Output Chat, a Shiny web application that uses large language models (LLMs) and retrieval-augmented generation (RAG) to generate R code from natural language specifications. Generated code is organized into editable code boxes and validated through independent LLM chat sessions that check each function call against its source definition. An in-app table preview allows visual verification. By accepting natural language input, the application also lowers the barrier to entry for programmers building R proficiency alongside SAS. Within the application, programmers can review and refine the code, then download the final R script and execute it in their R IDE to produce formatted Excel tables.


## Introduction

Pfizer Real-world data (RWD) programmers generate statistical tables summarizing patient demographics, clinical characteristics, and outcomes across cohorts and subgroups. While the statistical operations are often repetitive, manual coding introduces opportunities for inconsistency and error.

The oroutput suite of R functions was developed to standardize table generation workflows within RWD programming teams. It provides predefined functions for data preparation, statistical module execution, table compilation, and formatted output generation.

## The oroutput suite for Table Generation

The oroutput suite consists of four core functions that transform raw datasets into formatted statistical tables conforming to RWD programming standards.

### `oroutput_prepfile`: Dataset Preparation

The `oroutput_prepfile()` function initializes a preparation object that references the source dataset and defines the cohort structure, including any subgroups for stratified analysis. It validates subgroup specifications and calculates sample sizes for table headers.

### `oroutput_module`: Statistical Calculations

The `oroutput_module()` function executes statistical calculations using one of six module types, each designed for specific data types and analysis requirements. All modules follow a consistent output structure and handle subgroup stratification.

### `oroutput_compile`: Table Assembly

The `oroutput_compile()` function assembles all executed modules into a formatted data frame ready for Excel output. It creates headers with subgroup identifiers, optional sample sizes, and indentation for nested categories.

### `oroutput_makeFormatWorksheet`: Excel Formatting

The `oroutput_makeFormatWorksheet()` function writes the compiled table to an Excel workbook with standardized RWD formatting including organizational color schemes, borders, fonts, and metadata footers.

## AI-Assisted Code Generation with oroutput

While oroutput standardizes table generation, programmers must still manually construct function calls with appropriate arguments. For complex tables with multiple modules and subgroups, this is time-consuming and error-prone.

Recent advances in large language models (LLMs) present opportunities to reduce manual coding effort. However, applying generative AI to code production raises concerns about hallucination (generation of non-existent functions), inconsistency with organizational standards, and lack of transparency. This paper describes an approach that constrains LLM outputs to validated functions while providing a conversational interface for iterative code generation and review. By accepting natural language input, the application also lowers the barrier to entry for programmers building R proficiency alongside SAS, allowing them to produce standards-compliant oroutput code while learning R patterns.

### Retrieval-Augmented Generation Approach

The application uses retrieval-augmented generation (RAG) to constrain LLM outputs to documented oroutput functions. Function documentation, source code, and usage examples are embedded in a vector database using semantic embeddings. When users provide natural language table specifications through the chat interface, the system:

1. Retrieves the top-k most relevant documentation chunks based on semantic similarity
2. Provides these chunks as context to the LLM
3. Instructs the LLM to generate code using only the provided functions
4. Returns R code organized into code boxes conforming to organizational templates

This restricts the LLM to a validated function library, reducing the risk of hallucinated function calls.

### Advantages and Assumptions

Advantages:

- **Constrained Generation:** The LLM can only use functions present in the retrieval store
- **Consistency:** Generated code follows organizational style guides and project templates
- **Transparency:** Output is standard R code using functions programmers already know
- **Auditability:** Natural language queries are preserved as comments in generated scripts
- **Iterative Refinement:** Follow-up messages can modify generated code without starting over
- **Visual Verification:** An in-app table preview shows the compiled output before downloading
- **Automated Validation:** Generated code is checked against function source definitions

### Considerations

- **Ambiguity:** While the LLM handles most ambiguous queries well — particularly when descriptive column names provide sufficient context — edge cases with highly ambiguous specifications may still require follow-up clarification.

## The RWD OR Output Chat Application

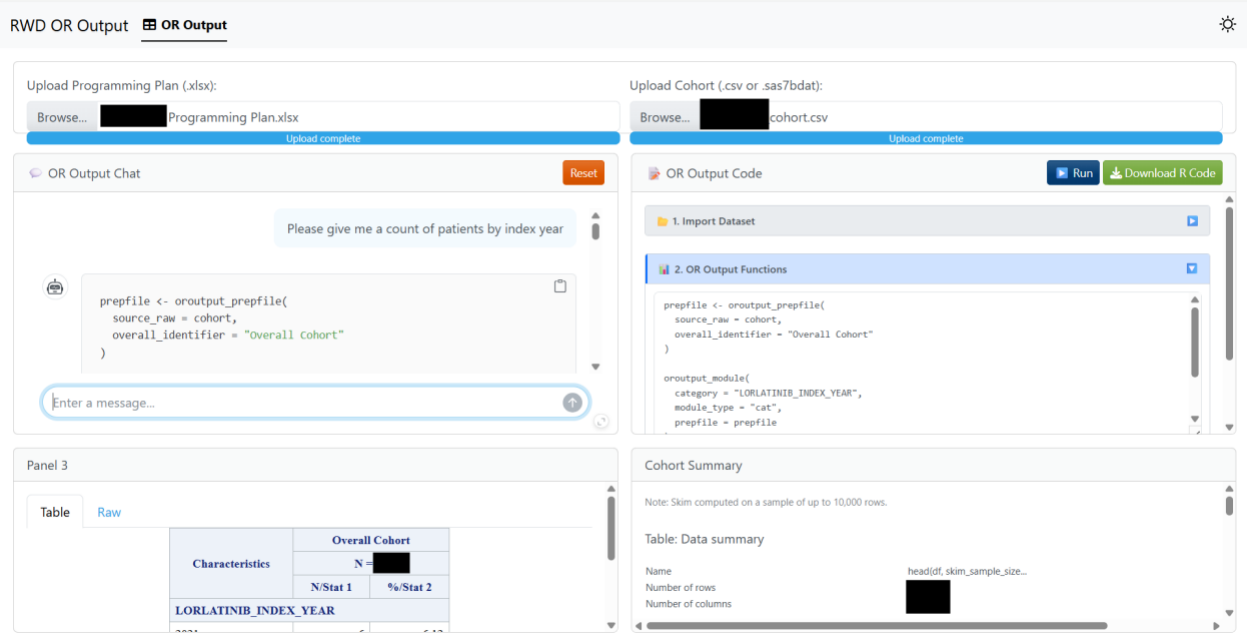
The application is a Shiny web application that integrates LLM-based code generation with organizational templates, an in-app table preview, and automated code validation. The application generates R code; programmers download the final script and execute it in their R IDE to produce the formatted Excel tables.

### Application Architecture

The application uses R Shiny with bslib for Bootstrap 5 theming. Key dependencies include shinychat for the chat interface, ellmer for LLM integration, ragnar for RAG-based documentation retrieval, flextable for the table preview, and skimr for dataset diagnostics. LLM access is provided through VoxR, an internal R package that wraps ellmer to connect to the Pfizer Vox GenAI platform — Pfizer's internally hosted LLM infrastructure. VoxR handles authentication and endpoint configuration so that the application can use ellmer's chat interface with the organization's LLM without exposing platform-specific details.

The interface has the following components:

- A top row for file uploads (programming plan and cohort dataset)
- A chat panel for natural language interaction with the LLM
- A code panel with three editable code boxes
- A table preview panel showing the compiled output with organizational styling
- A cohort summary panel with dataset diagnostics




## Project Configuration and Dataset Integration

Programmers upload two files directly through the application interface: a programming plan (`.xlsx`) and a cohort dataset (`.csv` or `.sas7bdat`). When the programming plan is uploaded, the application reads the programming plan's Setup sheet and auto-loads the setup variables (e.g., project directory, study ID) into the execution environment in the background. When the cohort dataset is uploaded, the application reads its column names and passes them to the LLM as context so the model can reference exact column names in generated code.

## The Three Code Boxes

Generated code is organized into three editable code boxes:

**Box 1 – Import Dataset:** Auto-generated when a dataset is uploaded. Contains the project header with programmer metadata, package loading, and the data import statement. The import method is determined by file extension: `read_csv()` for CSV files or `haven::read_sas()` for SAS files. Setup variables from the programming plan's Setup sheet are auto-loaded in the background execution environment.

**Box 2 – OR Output Functions:** Contains the `oroutput_prepfile()`, `oroutput_module()`, and `oroutput_compile()` calls generated by the LLM.

**Box 3 – Notes & Export:** Contains `openxlsx::createWorkbook()`, `oroutput_makeFormatWorksheet()`, and `openxlsx::saveWorkbook()` calls for Excel output, plus the original natural language query preserved as comments.

The system prompt instructs the LLM to respect these boundaries, and automated continuity checks verify that functions appear in the correct box.

## System Prompt Configuration

The system prompt constrains the LLM to generate only valid oroutput code and respect the three-box organization. It directs the model to return only R code without explanatory text, use only functions from the documentation store, assume data are ready for analysis, format each argument on its own line, and place functions in the correct code box. The prompt also explicitly prohibits unnecessary boilerplate — for example, `library()` calls are excluded because package loading is handled automatically in Box 1, and the LLM is restricted to only the oroutput and openxlsx functions needed for table generation. This prevents the LLM from generating extraneous setup code that would clutter the output or conflict with the application's own initialization.

The system prompt includes a function reference with parameters, types, and usage notes for `oroutput_prepfile()`, `oroutput_module()`, `oroutput_compile()`, and `oroutput_makeFormatWorksheet()`. When a dataset is loaded, the application reads the data file, extracts its column names, and appends them to the system prompt. This is important because oroutput functions require exact, case-sensitive column names. By providing the actual column names, the LLM can map user descriptions to the correct columns (e.g., a user who writes "age" can be matched to the column `AGE_N`). The prompt also instructs the LLM to ask the user when it cannot determine which column to use rather than guessing.

## Initializing the LLM Chat with RAG

The LLM chat session is lazily initialized when the programmer sends their first message. At initialization, the application constructs the full system prompt by combining the base instructions with any dataset column names currently loaded, then creates a chat session via the Vox GenAI platform. The application also connects to the RAG vector store and registers it as a retrieval tool on the chat, configured to return the 10 most semantically similar documentation chunks for each query. This gives the LLM access to oroutput function documentation during code generation. The session is cached and reused for subsequent messages in the same conversation.

## Conversational Code Generation

The application supports iterative code generation through a persistent chat session. When a programmer sends a message, the application determines whether this is a new request or a follow-up edit:

```
message_to_vox <- if (nzchar(box2_current) || nzchar(box3_current)) {
  paste0(
    user_input,
    "\n\nCurrent code in the three boxes (for context and editing):",
    "\nBox 1 (Import Dataset):\n```r\n", input$or_code_box_1, "\n```",
    "\nBox 2 (OR Output Functions):\n```r\n", input$or_code_box_2, "\n```",
    "\nBox 3 (Notes & Export):\n```r\n", input$or_code_box_3, "\n```"
  )
```

```
} else {
  user_input
}

stream <- chat$stream_async(message_to_vox)
```

For follow-up requests, the current contents of all three code boxes are sent to the LLM as context. The LLM evaluates which code boxes need changes based on the request and returns labeled code blocks for only those boxes, leaving the rest untouched. This means programmers can instruct the LLM to add new worksheets, create separate workbooks, reorder modules, or adjust formatting — and the LLM will determine which code boxes to modify accordingly. Responses are streamed to provide feedback in the chat panel.

## Automated Code Validation

After code is generated, the application performs two types of validation:

**Function-level validation** checks each oroutput function call against its source code definition. For each function found in the generated code, the application initializes a fresh, independent LLM chat session with that function's source code loaded as context. This means each validation check is a separate call to the Vox GenAI platform — a new chat instance with its own system prompt containing the specific function's source code.

```
or_check_code <- function(code) {
  check_items <- list(
    list(func = "oroutput_prepfile",            file = "oroutput_prepfile.R")
,
    list(func = "oroutput_module",              file = "oroutput_module.R"),
    list(func = "oroutput_compile",             file = "oroutput_compile.R"),
    list(func = "oroutput_makeFormatWorksheet", file = "oroutput_makeFormatWo
rksheet.R")
  )
  ...
}
```

**Box continuity validation** checks that function calls appear in the correct code box. When any of these checks detect an issue, the application posts a warning message in the chat panel describing the problem.

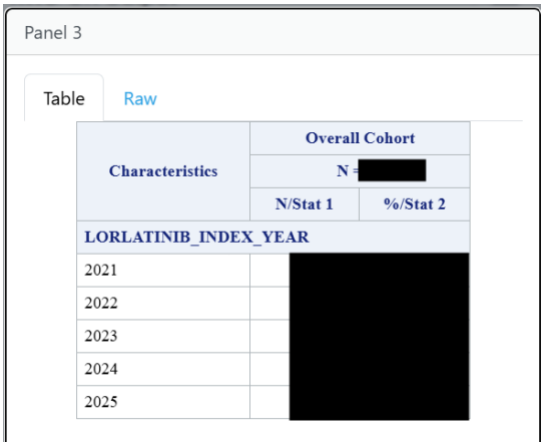
Additionally, if R coding errors occur during execution, the user can copy the error message into the chat box. Because the LLM has full context of the current code across all three boxes, it can interpret the error, identify the issue, and generate corrected code in response.

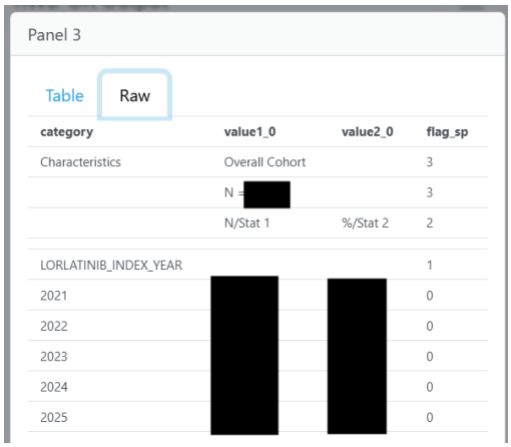
## Table Preview

When code is generated or the Run button is clicked, the application executes Box 1 and Box 2 code in an isolated environment and renders a preview of the compiled table:

```
or_execute_boxes <- function(box1_code, box2_code) {
  tryCatch({
    env <- new.env(parent = globalenv())
    eval(parse(text = paste(box1_code, box2_code, sep = "\n")), envir = env)
    ...
  }, error = function(e) {
    or_run_error(e$message)
    NULL
  })
}
```

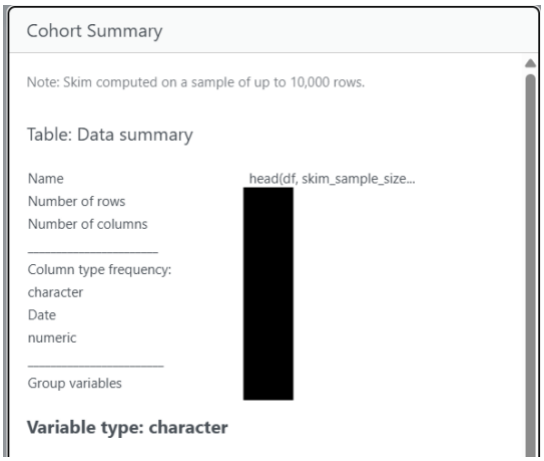
The preview uses flextable to approximate the formatting from `oroutput_makeFormatWorksheet()`, including color schemes, header styling, cell merges, and section grouping. A raw data tab shows the underlying data frame. This preview is for verification only. To produce the final formatted Excel workbook, programmers download the R script and run it in their R IDE.




## Dataset Diagnostics

When a dataset is loaded, the application generates a summary using `skimr::skim()` on a sample of the data. This shows column types, missing values, and distributions to help programmers formulate queries and review code.



## Human-in-the-Loop Design

The application keeps programmers in control through several mechanisms:

- **Visual Code Review:** Generated code is displayed in editable text areas where programmers can read, modify, or correct it
- **Table Preview:** The in-app preview shows the output structure before downloading
- **Iterative Refinement:** Follow-up messages can modify specific parts of the generated code
- **Automated Checks:** Function-level and box continuity validations flag potential issues
- **Download and Execute:** Programmers download the R script and execute it in their R IDE, maintaining a clear separation between code generation and code execution

The application generates code but does not produce the final deliverables. The download-and-execute step ensures programmers take ownership of the code before it runs.

## Example Usage

A typical workflow:

1. Upload the programming plan (`.xlsx`) and cohort dataset (e.g., a CSV file)
2. Review the auto-generated import code in Box 1
3. Type a specification in the chat — the application automatically detects all column names and provides them as context, so there is no need to wrap column names in braces or any special format. The table can also be built iteratively through follow-up messages. For example, start with a simple request:

   "Create a table with summary stats for age and number of patients by age group. Use subgroups for each year from 2021 to 2023."

   Then refine with follow-up messages:

   "Group diabetes and hypertension under a category called 'Comorbidity'."

   "Add a footnote that the cohort is only among patients from 2021 to 2023."
4. Review the generated code in the three code boxes; the table preview renders for verification
5. Send further follow-up messages to adjust (e.g., "Change the rounding to 1 decimal place")
6. Click Download R Code to export the script
7. Open the script in an R IDE, review, and execute to produce the Excel table

## Why a Shiny Application?

Delivering the workflow as a Shiny application has several practical benefits over a standalone R function.

### Interactive and Iterative Workflow

The application supports iterative refinement through conversation. Programmers can start with a basic specification and progressively build complexity through follow-up messages — adding modules, creating additional worksheets or workbooks, reordering table sections, adjusting formatting, or correcting errors. The application works best when programmers send simple, focused instructions across multiple messages rather than constructing a single complex specification. Each message can target a specific aspect of the table, and the LLM determines which code boxes to update based on the request.

### Integrated Interface

The application combines project configuration, code generation, code editing, table preview, and dataset diagnostics in one interface. This reduces context switching during the code generation phase. The three code boxes provide a structured view of the script. Once code is finalized, programmers download it for execution in their R IDE.

### Accessibility and Building Polyglot Programmers

The web interface is accessible regardless of experience with LLM APIs, RAG systems, or the R console. The application handles the technical complexity of connecting to the LLM platform, querying the documentation store, parsing responses, and organizing code.

This accessibility is particularly valuable for programmers adding R to their skillset. RWD programming teams aim to be polyglots proficient in both SAS and R, but programmers new to R face a learning curve with R syntax, package ecosystems, and coding conventions. The application lowers this barrier by allowing programmers to describe tables in natural language rather than writing R code from scratch. The generated code uses the same oroutput functions that experienced R programmers use, so programmers can learn R patterns by reviewing and modifying the output. Over time, this builds familiarity with R syntax and the oroutput framework while still producing correct, standards-compliant code from the start.

### Centralized Deployment

The application is deployed on Posit Connect, providing consistent configuration, version management, and access controls. All programmers use the same documentation store, system prompt, and LLM model. Updates are available to all users without requiring individual package updates.

## Discussion and Future Directions

The application addresses several challenges in AI-assisted code generation. This section discusses how the system handles key scenarios and identifies areas for future development.

### Multi-Sheet and Multi-Workbook Support

Programmers can instruct the LLM to create additional worksheets or separate workbooks through follow-up messages, and the LLM adjusts the relevant code boxes accordingly. Future versions could further streamline this with dedicated UI controls for managing multiple output targets within a session.

### Data Transformations

Although the application is designed primarily for oroutput table generation, the LLM can handle data transformations and complex conditional logic when guided through incremental prompting. Programmers can instruct the LLM to filter, subset, derive variables, or apply conditional logic as part of the code generation workflow. Each transformation can be specified in a separate follow-up message, and the LLM adjusts the relevant code boxes accordingly. More complex data preparation pipelines may benefit from dedicated documentation stores and validation frameworks in future versions.

### Ambiguity Handling

Because the system prompt includes the dataset's actual column names, the LLM can reason about user intent even when queries are imprecise. For example, if a user requests "age stats" and the dataset contains `AGE_N` and `AGE_GRP`, the LLM can infer which column to use based on the requested analysis type. Descriptive column names provide substantial context that reduces ambiguity without requiring the user to be explicit. When column names alone are insufficient to resolve intent, the conversational interface allows programmers to clarify through follow-up messages.

### Deployment Considerations

The application is deployed on Posit Connect. Several practical considerations apply:

- **Documentation maintenance** requires updating the vector store as functions change. Version control for documentation should be synchronized with code releases.
- **Programmer training** should cover effective query formulation and code review, including what the application can and cannot do.
- **Quality monitoring** should track how often manual corrections are needed, what types of errors occur, and compare AI-generated code quality to manually written code.

- **Version control** applies to both AI-generated and manually written code. The Download R Code feature exports scripts with embedded natural language queries. Code produced by AI should be labeled with metadata, and manual edits should be tracked.

## Conclusion

This application integrates AI-assisted code generation with the oroutput framework through a Shiny web interface. It constrains LLM outputs through retrieval-augmented generation, provides a table preview for verification, performs automated code validation, and supports iterative refinement through conversation.

The application generates code but does not produce final deliverables. Programmers review the generated code in editable code boxes, verify the output through the in-app preview, download the R script, and execute it in their R IDE to produce formatted Excel tables. This separation keeps programmers in control of final output. By accepting natural language input, the application also serves as an accessible entry point for programmers building R proficiency alongside SAS.

Edge cases with highly ambiguous specifications may require follow-up clarification. When R coding errors occur, programmers can paste the error into the chat for the LLM to diagnose and correct.

Future development areas include:

1. Dedicated UI controls for multi-sheet and multi-workbook workflows
2. Dedicated documentation stores for data preparation tasks
3. Enhanced ambiguity handling for edge cases with non-descriptive column names
4. Quality monitoring to track and improve code generation accuracy