PharmaSUG 2026 – Paper AI-328

Leveraging LLMs in Data Science Web Applications: Beyond the Chat Interface

Tyler Rowsell, Dishank Jani, Nandini Thampi and Hao Xu, AstraZeneca PLC.

ABSTRACT

Large Language Models (LLMs) have become ubiquitous in modern web applications, yet their implementation is often understood to be limited to conversational chat interfaces. This narrow consideration overlooks the transformative potential of LLMs as backend analytical services that can power data science workflows through strategic prompt engineering and response parsing. This paper explores architectural patterns for embedding LLM capabilities into analytical web applications, where the model operates as a backend service rather than solely a user-facing chatbot.

We demonstrate how structured prompt engineering, combining explicit output schemas, delimiter-based serialization, and multi-stage API orchestration, enables LLMs to perform complex subroutines. As representative use cases, we present this approach through two complementary AI capabilities within an R Shiny application. First, we implement semantic annotation detection where an LLM parses TLF shells to extract population definitions, analysis subsets, formatting constraints, transforming unstructured specifications into structured metadata. Second, we leverage this metadata to power interactive code generation with iterative refinement, where the LLM generates executable R code, tests it against live data, and autonomously corrects errors through multi-iteration debugging cycles.

These examples demonstrate that LLMs can serve as programmable reasoning engines when their outputs are constrained through careful prompt design. By treating LLM responses as structured data rather than conversational text, developers can create more powerful, context-aware applications that leverage natural language understanding while maintaining the reliability and predictability required for data science workflows.

INTRODUCTION

Ask most people how they interact with an LLM and they will describe opening a chat window, typing a question, and reading the response. This framing, while intuitive, is profoundly limiting. It casts the model as a helpful conversationalist rather than what it truly is: a programmable reasoning engine capable of transforming unstructured information into structured, actionable outputs.

The consequence of adhering to this mental model is that LLM adoption in data science and clinical programming will remain largely cosmetic, grafting a chatbot interface onto existing tools rather than reimagining what analysis pipelines can look like. Practitioners will write prompts manually, copy-paste responses, and manually integrate outputs, negating much of the efficiency gain the technology could provide.

This paper argues for, and demonstrates, a fundamentally different and well-established approach: treating LLMs as backend analytical services that are called programmatically, receive structured context, return structured outputs, and participate in automated workflows with error handling and retry logic (Yao et al., 2023). The key shift is from user-driven conversational AI to developer-engineered, application-integrated AI.

We ground this discussion in a concrete and relevant implementation: an exemplary R Shiny application (Chang et al., 2023) for basic clinical trial Table, Listing, and Figure (TLF) code generation. The application uses an LLM in two non-conversational ways: to extract structured annotations from free-form table shells, and to iteratively generate and self-correct executable R code. Neither use case involves a chat interface. Neither requires the end user to write a single prompt.

Paper Objectives

This paper has three aims: to reframe how clinical programmers think about LLM integration beyond chat; to provide concrete architectural patterns for embedding LLMs as backend services; and to demonstrate

these patterns through detailed technical examples in the context of clinical programming, with sufficient depth for practitioners to adapt the approach independently.

BACKGROUND AND RELATED WORK

LLM CAPABILITIES RELEVANT TO DATA SCIENCE

Modern LLMs exhibit capabilities that extend well beyond conversational text generation (Brown et al., 2020). Three are especially relevant to analytical workflows: natural language understanding, code generation, and pattern recognition. LLMs can parse loosely structured documents and extract entities, relationships, and constraints, a task that would traditionally require complex rule-based parsers or bespoke NLP pipelines. Their code generation capability has been extensively benchmarked (Chen et al., 2021); models ranging from GPT-4™ to Claude Sonnet 4.5™ achieve high pass rates on standard programming tasks (Bubeck et al., 2023; Chen et al., 2021) and can generate statistically correct data processing code when provided with adequate context about the data schema.

Pattern recognition is perhaps the least appreciated capability in this context. LLMs have absorbed vast amounts of structured data formats, clinical data standards such as CDISC ADaM and SDTM (CDISC, 2022), and domain-specific vocabulary. This pretrained model knowledge allows them to identify variable names, population filters, and statistical conventions in a TLF shell without being explicitly trained on each new table type.

COMMON IMPLEMENTATION PATTERNS

Current LLM integration in enterprise applications falls into three broad patterns. The first and most prevalent is the chat overlay (White et al., 2023) : an LLM chatbot is embedded in an existing application as an assistant (e.g., ChatGPT™ user chat interface). The model answers questions about the application or its data but does not participate in core workflows. The second pattern is code copilot integration (e.g., GitHub Copilot™), where the model suggests code completions, but a human developer makes all final decisions. The third, rarer pattern is agentic pipelines, where the model performs multi-step tasks with minimal human intervention (e.g., OpenClaw™).

The approach described in this paper sits between patterns two and three. The model is tightly integrated into a specific workflow, produces structured outputs that feed directly into subsequent processing steps, and self-corrects on failure, but does so within a bounded, developer-defined task scope rather than open-ended agency, which is particularly relevant given the regulatory demands of the pharmaceutical industry.

THE BACKEND SERVICE PARADIGM

The core architectural insight is that an LLM API call is functionally equivalent to any other microservice call. It receives input, performs processing, and returns outputs. The difference is that the processing is neural inference rather than deterministic computation. This framing enables standard software engineering practices: input validation, output parsing, error handling, retry logic, logging, and testing.

When developers adopt this equivalence in practice, LLM integration shifts to a systems design problem. The relevant questions become: What is the input schema? What output schema is required? What are acceptable failure modes? How many retries are appropriate?

ARCHITECTURE OVERVIEW

CONVERSATIONAL VS. BACKEND LLM ARCHITECTURE

The distinction between conversational and backend LLM architecture manifests in every layer of the software stack. Table 1 summarizes the key differences with examples.

Dimension	Conversational LLM	Backend LLM Service
Primary Interaction	User chat messages	Programmatic API calls
Output Format	Free-form text	Structured JSON / code / arrays
Error Handling	User corrects manually	Automated retry loops
Context Source	Conversation history	Application state + data
Typical Use Case	Q&A, writing assistance, info retrieval	Data processing, code gen
Integration Depth	Shallow (UI overlay)	Deep (core logic)
Prompt Control	User-driven	Developer-engineered

Table 1. Conversational vs. Backend LLM Integration: Key Architectural Differences

In conversational architecture, the prompt is owned by the user and varies unpredictably. In a backend architecture, the prompt is owned by the developer and is engineered for reliability. This shift from user-authored to developer-authored prompts is the fundamental enabler of backend integration.

EXAMPLE APPLICATION ARCHITECTURE

The reference implementation is an R Shiny application framework with two distinct layers. The frontend provides a multi-step workflow interface: users upload ADaM data sets, paste a TLF shell copied from Excel, review extracted annotations, inspect and edit generated code, and view the rendered output table. The backend processing layer, hidden entirely from the user, handles three responsibilities: (1) parsing uploaded data files across multiple formats; (2) constructing and dispatching LLM API calls; and (3) parsing, validating, and executing LLM responses.

The LLM integration layer is encapsulated in a single function, chatGPT_API() (Appendix A), which handles authentication, request construction, HTTP dispatch, status code handling, and response extraction. The full implementation is provided in Appendix A. This encapsulation means the rest of the application treats LLM calls as simple function calls that return a text string, which is then parsed according to the expected output format for each use case.

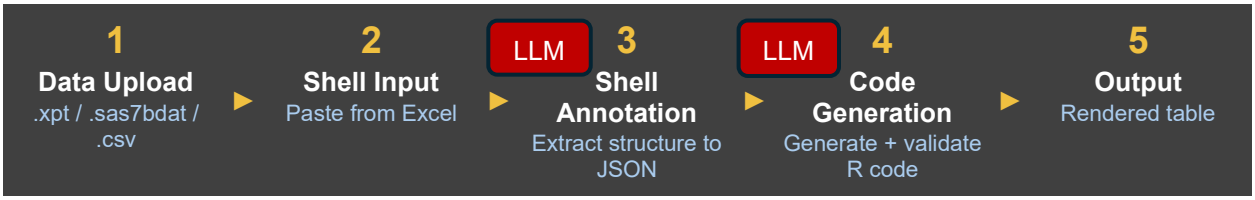


Figure 1. End-to-End Application Workflow: User Actions and LLM Processing Steps

DATA SECURITY AND PRIVACY CONSIDERATIONS

In clinical programming contexts, data privacy is non-negotiable. The reference implementation addresses this through several mechanisms. The API endpoint URL is stored as an environment variable and validated at runtime to enforce HTTPS. The API key is retrieved from environment variables, never hardcoded. Clinical data from data sets are never transmitted to the LLM; only metadata (variable names, types, and labels) are included in prompts. The data sets themselves remain in the R server environment and are never serialized into API requests. These design choices allow the system to operate within typical company security policies for internal API deployments.

PROMPT ENGINEERING FUNDAMENTALS

THE SYSTEM PROMPT AS A CONTRACT

Every API call in the application includes both a system prompt and a user prompt. The system prompt establishes the model's role, capabilities, and output constraints before any task-specific content is provided as the user prompt. A well-designed system prompt dramatically reduces output variance (Wei et al., 2022), functioning as a contract that defines the rules of the interaction. In a backend architecture, both the system prompt and the user prompt are developer-authored and programmatically constructed, neither originates from the end user of the application.

The annotation extraction system prompt, for example, specifies the model's role as a CDISC expert (Program 5), requires output to be valid JSON, and pre-empts common failure modes by explicitly prohibiting annotation of descriptive text labels and requiring exact variable name matching. These negative constraints are as important as positive instructions.

STRUCTURED OUTPUT SCHEMAS

The most important prompt engineering technique for backend integration is explicit output schema specification. Rather than asking the model to 'extract information', the prompt specifies the exact JSON structure the application expects, including field names, data types, and nesting depth. The application then parses this JSON deterministically. Code Example 1 shows the schema example embedded directly in the annotation extraction prompt.

```
# Code Example 1: JSON output schema embedded in the annotation extraction prompt.
# The model is shown this exact structure so its output can be parsed deterministically.
# This schema is passed as a literal string block inside the user prompt (see Code
Example 5).

EXAMPLE_OUTPUT_SCHEMA <- '
{
  "metadata": {
    "title": "[output title]",
    "datasets": ["[dataset(s)"],
    "population": { "filter": "[pop var] == \'Y\'", "variable": "[pop var]" },
    "table_structure": { "column_by": "[treatment var]", "include_total": true },
    "analysis_variables": [
      { "name": "[var]",    "label": "[label]",       "type": "continuous",
        "statistics": ["n","mean","sd","median","min","max"], "parent": null },
      { "name": "[var]", "label": "[label]", "type": "categorical",
        "statistics": ["n","percent"], "parent": null }
    ]
  },
  "cell_annotations": [
    { "row": 1, "column": 1, "content": "[dataset]",
      "annotation_type": "dataset", "variable": "[dataset]",   "parent": null },
    { "row": 2, "column": 2, "content": "[pop var]=\'Y\'",
      "annotation_type": "population_filter",   "variable": "[pop var]","parent": null },
    { "row": 3, "column": 1, "content": "[treatment var]",
      "annotation_type": "by_variable", "variable": "[treatment var]", "parent": null },
    { "row": 5, "column": 1, "content": "[var]",
      "annotation_type": "continuous_variable",  "variable": "[var]", "parent": null },
    { "row": 10, "column": 1, "content": "[var]",
      "annotation_type": "categorical_variable", "variable": "[var]", "parent": null }
  ]
}'
```

Program 1. JSON Output Schema Embedded in Annotation Extraction Prompt: Model Output Is Parsed Against This Structure

CONTEXT INJECTION

Backend LLMs require richer context than conversational LLMs because there is no human in the loop to clarify ambiguities. The annotation prompt injects three types of contextual information: the parsed shell

structure (a JSON representation of the table's cell coordinates and content), the available data set names, and the complete list of variable names and types for each data set. This allows the model to perform entity matching, identifying which cell contents correspond to real variable names, without access to actual data values.

CONSTRAINT LAYERING

Complex tasks benefit from layered constraints: broad behavioral constraints in the system prompt, task-specific instructions in the primary prompt, and explicit negative examples to prevent common errors. The code generation prompt illustrates this pattern by specifying what code to write, what not to include (data loading code, since data sets are pre-loaded), helper function signatures to use, and how to handle edge cases such as missing treatment group values. These constraints allow a single prompt to reliably produce code that executes correctly in the application's runtime environment.

USE CASE 1: SHELL ANNOTATION PROCESSING

THE PROBLEM

TLF shells are specification-rich documents that clinical programmers understand at a glance. A programmer can look at a shell and immediately identify which data set to use, what population to filter, which variables to analyze, how to group results, and what statistics to calculate. This implicit understanding happens because programmers recognize patterns: ADSL is clearly a data set name, SAFFL='Y' is obviously a filter condition, TRT01P in column headers indicates treatment grouping. Take this basic annotated table shell as an example.

ADSL	**FASFL = Y**			
Demographics (Full Analysis Set)		**TRT01P**		
		Arm A	Arm B	Total
AGE	Statistic	N=xxx	N=xxx	N=xxx
Age (Years)	n	xxx	xxx	xxx
	Mean	xx.x	xx.x	xx.x
	SD	xx.x	xx.x	xx.x
	Median	xx.x	xx.x	xx.x
	Min	xx	xx	xx
AGEGR1	Max	xx	xx	xx
Age group (Years)				
< 50	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
>= 50 - < 65	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
>= 65 - < 75	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
>= 75	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
Missing	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
SEX				
Sex				
Female	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
Male	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
Missing	n (%)	xxx (xx.x)	xxx (xx.x)	xxx (xx.x)
...	...	...	...	...

Table 2. Example Annotated Shell for a Standard Demographics Table Output.

However, this human-readable richness becomes a barrier when attempting to streamline code generation with LLMs. The central challenge is not extracting information from shells, that is straightforward for humans, but translating the implicit, visually-encoded specifications into explicit, machine-readable metadata that can drive code generation workflows.

FROM EXCEL SHELL TO STRUCTURED JSON: THE FULL PIPELINE

TLF shells in clinical programming practice are commonly stored in Excel™ files. The shell processing pipeline must therefore bridge the gap from an Excel-formatted specification to the cell-coordinate JSON that the LLM receives. Three deterministic processing stages run before the LLM is ever invoked (Programs 2-4).

Stage 1: Paste Capture. The user copies a shell directly from Excel and pastes it into a contenteditable HTML div in the Shiny application. Modern browsers preserve Excel's table structure as HTML during paste operations. A JavaScript listener, injected into the browser at session start via Shiny's runjs(), captures this HTML immediately after paste and forwards it to the R server as a reactive input value via Shiny.setInputValue(). Code Example 2 shows this listener and its R-side wrapper.

```
# Code Example 2: Stage 1 of the shell pipeline - capturing pasted Excel content.
# The observe() block runs once when the Shiny session starts, registering the
# JavaScript listener in the user's browser via runjs().

observe({
  runjs("
    $('#shellInput').on('paste', function(e) {
      // A 100ms delay lets the browser finish rendering the pasted content
      // before we read it. Without the delay, html() returns the pre-paste value.
      setTimeout(function() {
        // When content is pasted from Excel, the browser converts the
        // spreadsheet table into an HTML <table> element automatically.
        var htmlContent = $('#shellInput').html();
        // Shiny.setInputValue() writes to input$shellContent on the R server,
        // triggering any observeEvent() or reactive() that depends on it.
        Shiny.setInputValue('shellContent', htmlContent, {priority: 'event'});
      }, 100);
    });
  ")
})

# On the R server, input$shellContent is now an HTML string containing
# the pasted Excel table, ready for Stage 2 (parse_shell_to_table).
observeEvent(input$shellContent, {
  parsed_table <- parse_shell_to_table(input$shellContent)  # -> Stage 2
  shell_table(parsed_table)   # shell_table() is a reactiveVal
})
```

Program 2. Stage 1: Browser-Side Paste Capture; Excel Table Structure Preserved as HTML and Forwarded to R Server

Stage 2 - HTML Parsing. The R server receives the HTML string from input$shellContent and calls parse_shell_to_table(). The function uses the rvest library to extract the table into a data frame with known row and column coordinates (Wickham et al., 2019). A tab-separated plain-text fallback handles cases where Excel content arrives without HTML markup. Code Example 3 shows the complete function.

```
# Code Example 3: Stage 2 of the shell pipeline - parse_shell_to_table().
# INPUT:  shell_html  - HTML string from input$shellContent (Stage 1 output).
# OUTPUT: a data frame where row i / column j corresponds to table cell (i, j).
#         This data frame is passed directly to Stage 3 (create_shell_structure_json).

parse_shell_to_table <- function(shell_html) {
  tryCatch({
    # Primary path: Excel paste arrives as an HTML <table> element.
    if (grepl('<table', shell_html, ignore.case = TRUE)) {
      doc    <- read_html(shell_html)          # rvest: parse HTML string
      tables <- html_table(doc, fill = TRUE)   # extract all <table> elements
      if (length(tables) > 0) return(tables[[1]])
    }
```

```
    # Fallback: plain tab-separated text (some Excel paste modes skip HTML).
    clean <- gsub('<[^>]+>', '', shell_html)   # strip any residual HTML tags
    clean <- gsub(' ',  ' ', clean)       # decode non-breaking spaces
    lines <- strsplit(clean, '\n')[[1]]
    lines <- lines[nchar(trimws(lines)) > 0]   # drop blank lines

    max_cols <- max(sapply(lines, function(l) length(strsplit(l, '\t')[[1]])))
    rows <- lapply(lines, function(line) {
      cells <- trimws(strsplit(line, '\t')[[1]])
      length(cells) <- max_cols  # pad short rows with NA
      cells[is.na(cells)] <- ''
      cells
    })
    as.data.frame(do.call(rbind, rows), stringsAsFactors = FALSE)
  }, error = function(e) NULL)
}
```

Program 3. Stage 2: Shell Parsing: HTML Table Extraction with Tab-Separated Fallback; Returns Data Frame of Cell Values

Stage 3 - Coordinate Serialization. The data frame from Stage 2 is passed to create_shell_structure_json(), which records each non-empty cell's row index, column index, and content as a list. That list is then serialized to a JSON string using jsonlite::toJSON() before being injected into the LLM prompt. The row/column coordinates are critical: they allow the LLM's returned annotations to be mapped back to specific cells in the original table for visual display. Code Example 4 shows both functions and the serialization step together.

```
# Code Example 4: Stage 3 of the shell pipeline — coordinate serialization.
# INPUT:  table_df  — data frame from parse_shell_to_table() (Stage 2 output).
# OUTPUT: shell_structure_json_string - a JSON string ready for prompt injection.

# Step 3a: build an R list recording every non-empty cell with its coordinates.
create_shell_structure_json <- function(table_df) {
  cells <- list()
  for (i in 1:nrow(table_df)) {
    for (j in 1:ncol(table_df)) {
      val <- table_df[i, j]
      if (!is.na(val) && nchar(trimws(val)) > 0) {
        cells[[length(cells) + 1]] <- list(
          row     = i,
          column  = j,
          content = trimws(val)
        )
      }
    }
  }
  list(rows = nrow(table_df), columns = ncol(table_df), cells = cells)
  # Returns an R list such as:
  # $rows: 12  $columns: 6
  # $cells: list of lists, each with $row, $column, $content
}

# Step 3b: convert the R list to a JSON string for prompt injection.
# This string becomes shell_structure_json_string used in Code Example 5.
shell_structure      <- create_shell_structure_json(shell_table())
shell_structure_json_string <- jsonlite::toJSON(
  shell_structure, pretty = TRUE, auto_unbox = TRUE
)
# shell_structure_json_string is now a character string such as:
# '{"rows":12,"columns":6,"cells":[{"row":1,"col":1,"content":"ADSL"},...]}'
```

Program 4. Stage 3: Coordinate Serialization - Data Frame to R List (create_shell_structure_json) Then to JSON String (jsonlite::toJSON)

THE ANNOTATION EXTRACTION LLM CALL

Only after the three deterministic stages above are complete does the LLM enter the workflow. The prompt is assembled by combining the cell-coordinate JSON from Stage 3 with data set metadata (variable names and types) and the output schema defined in Code Example 1. Code Example 5 shows the complete prompt construction and API call.

```
# Code Example 5: Annotation extraction — assembling and dispatching the API call.
# This code runs inside observeEvent(input$processShell, {...}) on the Shiny server.

# --- Inputs assembled from prior pipeline stages ---
# shell_structure_json_string : JSON string from Code Example 4 (Stage 3)
# uploaded_data()             : reactiveVal holding the loaded ADaM data frames

available_datasets <- names(uploaded_data())

# Build variable metadata: names + types only, NO data values (privacy requirement)
variable_info <- sapply(available_datasets, function(ds) {
  ds_obj <- uploaded_data()[[ds]]
  vars   <- names(ds_obj)
  paste0(ds, ': ', paste(sapply(vars, function(v) {
    type  <- if (is.numeric(ds_obj[[v]])) 'numeric' else 'character'
    label <- attr(ds_obj[[v]], 'label')  # SAS variable label if present
    if (!is.null(label)) paste0(v, ' (', type, ', "', label, '")') else paste0(v, ' (',
type, ')')
  }), collapse = ', '))
}, USE.NAMES = FALSE)

# --- System prompt: establishes role and hard output constraints ---
sysprompt <- paste(
  'You are an expert clinical data analyst specializing in CDISC ADaM standards.',
  'Identify annotations in TLF shells with their exact cell coordinates.',
  'CRITICAL: Identify parent-child nesting relationships between variables.',
  'Return ONLY valid JSON matching the schema below. No markdown, no prose.',
  sep = '\n'
)

# --- User prompt: injects all three context types ---
prompt <- paste(
  'Analyze this TLF shell and identify all annotations with cell locations.',
  '', 'AVAILABLE DATA SETS:', paste(available_datasets, collapse = ', '),
  '', 'DATA SET VARIABLES (names and types only, no data values):',
  paste(variable_info, collapse = '\n'),
  '', 'SHELL STRUCTURE (cell coordinates + content from parse_shell_to_table):',
  shell_structure_json_string,   # <-- output of Code Example 4, Stage 3
  '', 'REQUIRED OUTPUT SCHEMA (return JSON matching this structure exactly):',
  EXAMPLE_OUTPUT_SCHEMA,         # <-- schema defined in Code Example 1
  sep = '\n'
)

# --- API call: returns a text string containing the JSON response ---
raw_response <- chatGPT_API(API_key, prompt, model = 'gpt-4-32k', sysprompt)
# raw_response is a character string. Next step: clean and parse (see Appendix B).
```

Program 5. Annotation Extraction Prompt - Assembled from Stage 3 JSON, Data set Metadata, and the Output Schema from Program 1

THE ANNOTATION EXTRACTION FLOW

Figure 2. Shell Annotation Pipeline: Three Deterministic Stages (Programs 2–4) Followed by LLM Inference (Program 5)

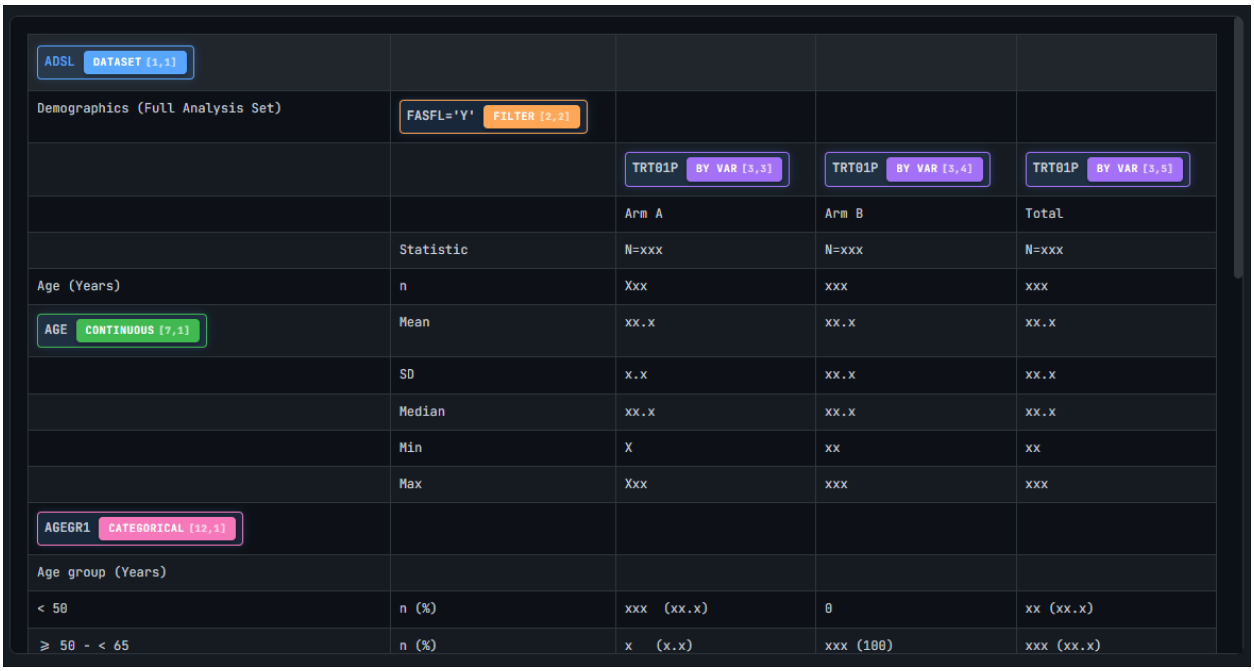
ANNOTATION TYPE CLASSIFICATION

The prompt instructs the LLM to classify each identified annotation into one of five categories: data set, population_filter, by_variable, continuous_variable, or categorical_variable. These categories map directly to code generation requirements. A by_variable annotation tells the code generator to use that variable as a column grouping key. A population_filter annotation provides the subsetting condition. This semantic classification eliminates the need for subsequent re-interpretation of the annotations.

The prompt also instructs the model to identify hierarchical parent-child relationships, for example, a body system variable (AEBODSYS) as the parent of a preferred term variable (AEDECOD). This nesting information is propagated through the metadata and used by the code generator to produce properly layered table rows with correct indentation.

POST-PROCESSING

After the API response is received, the application strips any markdown code fences that the model may have added, parses the JSON, and separates the metadata and cell_annotations components. The complete response cleaning and parsing logic is shown in Appendix B. The cell annotations are then passed to create_annotated_table_html(), which renders the original shell with visual type badges overlaid on each annotated cell, providing immediate visual confirmation of what the model has extracted without requiring the user to read raw JSON. Display 1 is a demonstration of this, captured from the reference application UI after processing a table shell based on Table 2.



Display 1. UI Example of Identified and Tagged Annotations from Demographics Shell.

USE CASE 2: ITERATIVE CODE GENERATION WITH ERROR HANDLING

THE CODE GENERATION CHALLENGE

The true power of treating LLMs as backend services emerges when multiple LLM calls are chained together, with each stage's structured output becoming the next stage's structured input. Code generation demonstrates this principle: the rich metadata extracted from the shell in Use Case 1 becomes the complete specification that drives automated code generation in Use Case 2. The output of chatGPT_API() in Code Example 5 feeds, after JSON parsing, directly into the context of the code generation call.

This chaining is the architectural backbone of the framework: structured data flows between processing stages without human intervention, with the LLM acting as a reasoning step that converts one structured representation into another.

THE CODE GENERATION PROMPT

The JSON metadata produced by the annotation stage is serialized back to a string and injected into the code generation prompt. Code Example 6 shows the system prompt, the prompt construction, and how the metadata from Use Case 1 becomes the primary context for this LLM call.

```
# Code Example 6: Code generation prompt — chaining Use Case 1 output as input.
# This code runs inside observeEvent(input$generateCode, {...}) on the Shiny server.

# --- Input assembled from Use Case 1 output ---
# extracted_json() : reactiveVal holding json_data$metadata from Appendix B parsing.
#                    This is the 'metadata' section of the JSON returned by Program 5.

# Re-serialize the R list back to a JSON string for prompt injection.
# (It was parsed from JSON in Appendix B; we serialize it again here for the prompt.)
json_str <- jsonlite::toJSON(extracted_json(), pretty = TRUE, auto_unbox = TRUE)

available_datasets <- names(uploaded_data())

# --- System prompt: establishes code style and critical constraints ---
sysprompt <- paste(
  'You are an expert R programmer specializing in clinical trial data analysis.',
  'Generate code that builds tables dynamically from data, not hardcoded values.',
  'Use tidyverse for data manipulation and kable/kableExtra for formatting.',
  'Store the final formatted table in a variable named final_output.',
  sep = '\n'
)

# --- User prompt: injects annotation metadata as the primary specification ---
initial_prompt <- paste(
  'Generate R code to create a TLF table from the following annotated metadata.',
  '',
  '# JSON METADATA (output of annotation stage, defines table structure):',
  json_str,   # <-- extracted_json() from Use Case 1, serialized back to string
  '',
  '# DATA SETS AVAILABLE IN THE R ENVIRONMENT (do not load, they are pre-loaded):',
  paste(available_datasets, collapse = ', '),
  '',
  '# REQUIREMENTS:',
  '- Apply population filter from metadata$population$filter',
  '- Group columns by metadata$table_structure$column_by variable',
  '- Add a Total column if metadata$table_structure$include_total is true',
  '- Handle nested parent-child variables with proper row indentation',
  '- Return ONLY executable R code, no markdown fencing, no explanatory prose',
  sep = '\n'
)
# initial_prompt is passed to chatGPT_API() in iteration 1 of the retry loop.
# See Code Example 7 for how this prompt is used and how errors are handled.
```

Program 6. Code Generation Prompt: Annotation Metadata from Use Case 1 Is Injected as the Primary Table Specification

THE ITERATIVE REFINEMENT LOOP

The core of the code generation system is an automated retry loop that runs up to three iterations. In each iteration, the model generates code, the application evaluates it immediately using R's eval(parse()) construct (Program 7), and the result is tested for the presence of an error condition. If the code fails, the error message is captured and injected into a revised prompt for the next iteration. Code Example 7 shows the complete loop.

```
# Code Example 7: Iterative refinement loop — automated error detection and correction.
# INPUT:  initial_prompt  — code generation prompt built in Code Example 6.
#         sysprompt       — system prompt from Code Example 6.
# OUTPUT: final_code (character), code_output() reactiveVal set on success.

max_iterations  <- 3
current_iteration <- 1
code_validated  <- FALSE
final_code      <- NULL
last_error      <- NULL

while (current_iteration <= max_iterations && !code_validated) {

  # Iteration 1: use the full specification prompt from Code Example 6.
  # Iterations 2+: send the original prompt PLUS the specific error that occurred,
  #   so the model knows exactly what to fix rather than regenerating from scratch.
  prompt_to_use <- if (current_iteration == 1) {
    initial_prompt
  } else {
    paste(
      'The code below produced an error. Fix only what caused the error.',
      'Do not rewrite the entire solution, return corrected code only.',
      '',
      '# ERROR MESSAGE:', last_error,
      '',
      '# CODE THAT PRODUCED THE ERROR:', final_code,
      '',
      '# ORIGINAL SPECIFICATION (for reference):', initial_prompt,
      sep = '\n'
    )
  }

  # Call the LLM and extract code from the response.
  # extract_code() strips markdown fencing if the model added it despite instructions.
  raw_response <- chatGPT_API(API_key, prompt_to_use, model, sysprompt)
  final_code   <- extract_code(raw_response)

  # Execute the generated code in the current R session.
  # tryCatch captures runtime errors without stopping the application.
  result <- tryCatch(
    eval(parse(text = final_code)),
    error = function(e) list(error = TRUE, message = e$message)
  )

  if (isTRUE(result$error)) {
    # Code failed: record the exact error message for the next iteration's prompt.
    last_error        <- result$message
    current_iteration <- current_iteration + 1
  } else {
    # Code succeeded: store output and exit the loop.
    code_validated <- TRUE
    code_output(result)   # code_output() is a reactiveVal that triggers the UI
  }
```

```
}

# After the loop, final_code holds the last generated code (successful or not).
# iteration_count(current_iteration) records how many attempts were needed.
```

Program 7. Iterative Refinement Loop: LLM Receives the Exact Error Message From Each Failed Attempt; Loop Runs Until Success or Max Iterations

PROCESS FLOW

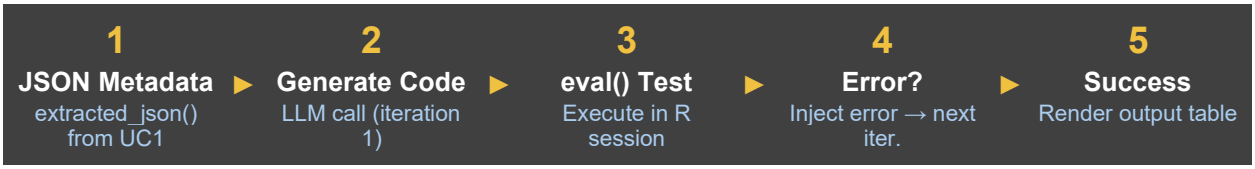


Figure 3. Iterative Code Generation Loop: Automated Error Detection and LLM Self-Correction (Programs 6–7)

WHY ITERATION WORKS

The effectiveness of this approach relies on a property of LLMs that is underappreciated in the context of code generation: models are significantly better at correcting a specific error than at avoiding that error proactively. When the model receives 'Error in bind_rows(): Column AGE is type numeric, but expected character', it can precisely identify and fix the problem. When writing code from scratch, it cannot anticipate all possible type interactions in a specific user's data set.

PRACTICAL CONSIDERATIONS AND LESSONS LEARNED

The reference implementation presented in this paper serves as a proof-of-concept demonstration of well-established methods, rather than a production-ready solution for clinical TLF code generation. Our intent is to illustrate architectural patterns for backend LLM integration through a concrete, domain-relevant example: shell annotation extraction feeding into iterative code generation. The workflow is deliberately simplified to highlight the core principles of structured prompt engineering, multi-stage API orchestration, and automated error handling without the complexity of a full production system.

WHAT WORKS WELL

- Schema specification is the single most impactful prompt engineering technique. Providing a complete JSON example in the prompt (as in Program 1) reduces output variability more than any other single instruction.
- Negative constraints prevent the most common failure modes. Explicitly instructing the model what NOT to do eliminates predictable error classes without lengthening the positive instructions.
- Deterministic pre-processing before LLM calls reduces complexity. Parsing the shell into structured JSON before sending it to the LLM means the model receives clean coordinate data rather than raw HTML.
- Error message injection in retry prompts is highly effective. LLMs are accurate error interpreters and typically produce correct fixes when given the exact error message, the failing code, and the original specification together.
- Separating metadata extraction and code generation into distinct LLM calls produces better results than combining them into a single call. Each call can be optimized independently.

CHALLENGES

- Prompt length management: the code generation prompt approaches the context window limit of some models when data sets have many variables. Careful curation of what context to include is necessary for production use.
- Non-determinism: even at temperature=0, LLMs occasionally produce different outputs for identical inputs. Production systems should log all LLM inputs and outputs for reproducibility and debugging.
- JSON parsing reliability: models sometimes return JSON wrapped in markdown code fences or with trailing commas that violate the JSON specification. Defensive parsing with regex cleaning before JSON parsing is essential (see Appendix B).

PROMPT DESIGN BEST PRACTICES

Based on development experience, the following practices are recommended for backend LLM integration:

- Always include a complete example of the expected output format, not just a schema description. The model performs better when shown a concrete example rather than an abstract specification.
- Use numbered, hierarchically organized instructions. LLMs attend to prompt structure and numbered lists improve instruction following compared to prose paragraphs.
- Provide data set metadata (variable names and types) rather than data values. This satisfies the model's need for context while keeping sensitive data out of API calls.
- Specify a timeout for API calls (the application uses 120 seconds) and handle timeouts gracefully. LLM APIs under load can be slow.
- Implement a minimum of three retry iterations. Most recoverable errors are resolved within two iterations; a third catches the remaining edge cases without excessive API cost.

CONCLUSION

The central argument of this paper is that the chat interface is not an inherent property of LLM integration; it is merely the most visible one. LLMs are versatile computational tools capable of performing structured analytical tasks when their inputs are carefully engineered and their outputs are parsed programmatically.

The architectural framework demonstrated here: developer-owned prompts, structured output schemas, deterministic pre-processing, multi-stage chaining, and automated retry loops, specifically in the context of a clinical programming workflow, is the contribution of this paper. The TLF demonstration application is illustrative context. Readers familiar with clinical programming will find the use cases immediately recognizable, but the patterns transfer directly to any data science or clinical programming domain: automated report generation from data dictionaries, translating analyst requests in plain English into SQL or pandas operations, generating documentation from code, or identifying data quality issues from schema descriptions.

The key mental shift required is to stop thinking about what an LLM says and start thinking about what an LLM returns. When LLM outputs are treated as structured data rather than conversational responses, they become first-class components of analytical pipelines.

As LLMs continue to evolve, the patterns demonstrated here: structured input/output, deterministic preprocessing, and automated validation, will become increasingly important. The pharmaceutical industry's stringent requirements for reproducibility and auditability make it an ideal proving ground for production-grade LLM integration. Future work should explore formal validation frameworks, regulatory guidance for LLM-assisted programming, and standardized prompt libraries for common clinical programming tasks.

REFERENCES

Brown, T. B., et al. (2020). Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.

Bubeck, S., et al. (2023). Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv:2303.12712.

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

Chang, R., Cheng, J., Allaire, J. J., et al. (2023). shiny: Web Application Framework for R. R package version 1.7.5.

Chen, M., et al. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35.

White, J., et al. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382.

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

Yao, S., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023.

CONTACT INFORMATION

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

Tyler Rowsell

AstraZeneca PLC

tyler.rowsell@astrazeneca.com

APPENDIX A: CHATGPT_API() LLM INTEGRATION FUNCTION

The chatGPT_API() function is the sole point of contact between the R application and the LLM provider. It encapsulates all authentication, request construction, HTTP dispatch, status code handling, and response extraction. The rest of the application calls it as a simple function that accepts a prompt and returns a text string.

```
# Appendix A: Complete chatGPT_API() implementation.
# INPUT:  api_key       — from Sys.getenv('OPENAI_API_KEY'), never hardcoded.
#         prompt        — the user-turn prompt (built in Programs 5 or 6).
#         model_name    — the deployment name, e.g. 'gpt-4-32k'.
#         sysprompt     — the system-turn prompt (built in Programs 5 or 6).
#         temperature   — 0 for maximum determinism in production use.
#         max_length    — maximum tokens to generate in the response.
#         timeout_seconds — abort the HTTP request after this many seconds.
# OUTPUT: a single character string containing the model's response text.

API_key <- Sys.getenv('OPENAI_API_KEY')   # never hardcode credentials
URL     <- Sys.getenv('OPENAI_URL')        # must be https:// (enforced below)

chatGPT_API <- function(api_key, prompt, model_name = 'gpt-4-32k',
                        sysprompt, temperature = 0,
                        max_length = 16384, timeout_seconds = 120) {

  # Security checks before any network activity.
  if (is.na(api_key) || nchar(trimws(api_key)) == 0)
    stop('Authentication failed: API key is missing or invalid')
  if (is.na(URL) || !grepl('^https://', URL))
    stop('Security error: API endpoint must use HTTPS')

  # Build the message list: system turn followed by user turn.
  messages <- list(
    list(role = 'system', content = sysprompt),
    list(role = 'user',   content = list(list(type = 'text', text = prompt)))
  )

  # Dispatch the HTTP POST request with a hard timeout.
  response <- tryCatch({
    httr::POST(
      url    = sprintf(URL, model_name),
      httr::add_headers('api-key' = api_key, 'Content-Type' = 'application/json'),
      encode = 'json',
      body   = list(messages    = messages,
                    temperature = temperature,
                    max_tokens  = max_length),
      httr::timeout(timeout_seconds)
    )
  }, error = function(e) stop('HTTP request failed: ', e$message))

  # Handle HTTP error codes before attempting to parse the response body.
  status <- httr::status_code(response)
  if (status %in% c(401, 403)) stop('Authentication failed: check API key')
  if (status >= 400)           stop(paste('API error: HTTP status', status))

  # Extract the response text from the parsed JSON body.
  body <- httr::content(response, 'parsed')
  if (is.null(body$choices) || length(body$choices) == 0)
    stop('API returned no choices in response body')

  return(body$choices[[1]]$message$content)  # character string
}
```

Appendix A. Complete chatGPT_API() Implementation: Security Validation, HTTP Dispatch, and Response Extraction

APPENDIX B: RESPONSE CLEANING AND JSON PARSING

After chatGPT_API() returns the raw response string, a cleaning and parsing step converts it to a structured R list. This step is essential because models occasionally wrap JSON in markdown code fences or introduce trailing commas despite instructions not to. The cleaned metadata and cell annotations are stored in separate reactive values for use by the rest of the application.

```
# Appendix B: Cleaning and parsing the annotation extraction API response.
# INPUT:  raw_response  — character string from chatGPT_API() call in Program 5.
# OUTPUT: extracted_json() reactiveVal set to json_data$metadata (R list).
#         cell_annotations() reactiveVal set to json_data$cell_annotations (R list).

# Step 1: strip markdown code fences that the model may have added.
clean <- raw_response
clean <- gsub('```json\\s*', '', clean)  # remove opening ```json fence
clean <- gsub('```\\s*',     '', clean)  # remove closing ``` fence
clean <- trimws(clean)

# Step 2: parse the cleaned string as JSON.
json_data <- tryCatch({
  jsonlite::fromJSON(clean, simplifyVector = FALSE)
}, error = function(e) {
  # Second attempt: remove trailing commas, which are invalid JSON
  # but occasionally generated by LLMs despite explicit instructions.
  clean_fixed <- gsub(',\\s*([}\\]])', '\\1', clean)
  jsonlite::fromJSON(clean_fixed, simplifyVector = FALSE)
})

# Step 3: validate the top-level structure before storing.
if (is.null(json_data) || !is.list(json_data))
  stop('API returned invalid JSON structure')
if (is.null(json_data$metadata) || is.null(json_data$cell_annotations))
  stop('API response missing required metadata or cell_annotations fields')

# Step 4: store the two sections in separate reactive values.
# extracted_json()    feeds into Code Example 6 (code generation prompt).
# cell_annotations()  feeds into create_annotated_table_html() for visual display.
extracted_json(json_data$metadata)
cell_annotations(json_data$cell_annotations)
```

Appendix B. Response Cleaning and JSON Parsing: Markdown Stripping, Trailing-Comma Repair, and Separation into Metadata and Cell Annotations

APPENDIX C: DATA SET METADATA EXTRACTION

Rather than transmitting raw data values to the LLM, the application extracts only structural metadata: variable names, data types, and SAS labels. This metadata is sufficient for the LLM to perform entity matching against shell cell contents, and it satisfies data privacy requirements by keeping actual clinical values out of API requests.

```
# Appendix C: Data set metadata extraction for prompt injection.
# INPUT:  uploaded_data()    — reactiveVal; named list of data frames.
# OUTPUT: variable_info      — character vector, one element per data set.
#         Each element is a string like:
#         'ADSL: USUBJID (character), AGE (numeric, "Age at Screening"), ...'
#         This vector is pasted into the LLM prompt in Program 5.

available_datasets <- names(uploaded_data())

variable_info <- sapply(available_datasets, function(ds_name) {
  ds   <- uploaded_data()[[ds_name]]
  vars <- names(ds)

  var_strings <- sapply(vars, function(v) {
    # Classify as numeric or character for the LLM's type-matching logic.
    type  <- if (is.numeric(ds[[v]])) 'numeric' else 'character'
    # SAS variable labels are preserved by haven::read_xpt() / read_sas()
    # as R attributes; include them so the LLM can match labels to shell text.
    label <- attr(ds[[v]], 'label')
    if (!is.null(label) && nchar(label) > 0) {
      paste0(v, ' (', type, ', "', label, '")')
    } else {
      paste0(v, ' (', type, ')')
    }
  })

  paste0(ds_name, ': ', paste(var_strings, collapse = ', '))
  # Note: data values (ds[[v]]) are never included — only names, types, labels.
}, USE.NAMES = FALSE)

# variable_info is now ready for injection into the prompt in Program 5:
# paste(variable_info, collapse = '\n')
```

Appendix C. Data set Metadata Extraction: Variable Names, Types, and SAS Labels Only; Data Values Are Never Included in API Requests