PharmaSUG China 2026 - Paper AI-208

AI-Assisted Code Generation for Structured Table: A Structured Filtering Logic Extraction Approach

Qing Ma, Johnson & Johnson

ABSTRACT

AI is reshaping statistical programming for structured tables. In these settings (e.g., some efficacy outputs), shells are standardized and variation stems from endpoint-specific selection. We present an approach that externalizes and formalizes endpoint-level filtering logic. A small set of reference programs anchors consistency; iterative AI identifies, normalizes, and organizes rules, then reuses them to generate batch programs aligned with predefined shells.

Logic expansion is separated from validation: AI standardizes and consolidates rules, while experts perform targeted checks for consistency and clinical correctness. Traceability is supported by a rule registry with provenance, diffs across iterations, and mappings from rules to code and table cells.

When data are structured and patterns stable, the method emphasizes logic reuse, high completeness, and efficient review. Efficacy serves as an example in the paper and the approach applies to any standardized table stream, turning implicit logic into reusable, auditable artifacts that improve transparency, scalability, and consistency.

INTRODUCTION

In clinical trial statistical programming, many output types share a standardized shell — the same layout, the same analysis structure, the same reporting format — while varying only in endpoint-specific logic or slightly different layouts within the same method. When a table stream has this property, writing each program individually means repeating the same structural work dozens or hundreds of times, differing only in a small set of parameter values. This is tedious, error-prone, and expensive to maintain when design changes occur.

This paper presents a systematic, AI-assisted approach to generate programs in batch for any such standardized table stream. The core idea is to externalize the endpoint-specific variation into a parameter registry, build a template that captures the shared structure, and use AI to populate the registry automatically from existing study materials. Once in place, the entire program set can be regenerated from a single command, and design changes propagate instantly across all affected programs.

QC programs for efficacy outputs in a Phase 3 clinical trial are used as the working example throughout this paper. The approach proceeds in four stages: (1) categorize all outputs to identify structural patterns; (2) parameterize a reference program so that variation is captured in named parameters; (3) use AI to extract parameter values from existing study materials; and (4) generate, deploy, and maintain all programs from the registry. The same workflow applies to any standardized table stream — efficacy, safety, demographics, or beyond.

STEP 1: CATEGORIZE THE OUTPUT UNIVERSE

The first step is to let AI understand what it is dealing with. Before generating any code, AI needs to examine the full list of outputs and classify them by structural similarity. Outputs that share the same layout, the same analysis method, and the same data source can be generated from a single template. The programmer defines the classification rules; AI does the scanning, applying those rules consistently across hundreds of titles and producing a structured result.

CATEGORIZATION DIMENSIONS

The programmer provides the output title list and describes the dimensions along which outputs differ. For a Phase 3 clinical trial with approximately 200 efficacy outputs, the following four dimensions were sufficient to fully classify every output:

- Endpoint type: Binary (yes/no response) or Continuous (measured value); for example, a “HAQ-DI response” output is Binary while “change from baseline in HAQ-DI” is Continuous
- Endpoint multiplicity: Single parameter or Multiple parameters per output; for example, an ACR components table reporting seven individual scores in one shell is Multiple
- Statistical method: CMH test, ANCOVA with multiple imputation, or Descriptive only; for example, binary endpoints under the main estimand use a CMH test, while supplementary estimand outputs are descriptive
- Analysis tier and estimand: for example, Primary Endpoint Analysis with Primary Estimand, or Other Endpoint Analysis with Supplementary Estimand

AI then wrote a Python script that applied keyword matching to each title — flagging outputs containing words like “response,” “improvement,” “resolution,” and “remission” as Binary, extracting statistical method and analysis tier from structured title patterns, and producing a categorized CSV with one row per output. Ambiguous cases were flagged for programmer review. This separation of roles — AI handles bulk extraction, the programmer handles exceptions — keeps the process both fast and accurate.

CATEGORIZATION RESULTS

Endpoint Type	Multiplicity	Statistical Method
Binary	Single	CMH test
Binary	Single	Descriptive
Binary	Single	Binomial GLM
Binary	Single	Subgroup
Binary	Multiple	CMH test
Binary	Multiple	Descriptive
Continuous	Single	ANCOVA + MI
Continuous	Single	Descriptive
Continuous	Single	Subgroup
Continuous	Multiple	ANCOVA + MI
Continuous	Multiple	Descriptive

Table 1. Efficacy Output Categories

In the example study, approximately 200 efficacy outputs collapsed into 11 categories — meaning a small number of categories could cover the full output universe. An important structural pattern also emerged: formal statistical testing (CMH or ANCOVA with MI) appeared exclusively in Primary, Secondary, and Main analyses. Supplementary and Other analyses were always descriptive, regardless of endpoint type. This regularity simplified template design, because the presence of a statistical test was entirely determined by the analysis tier and estimand — not by the endpoint type alone.

STEP 2: PARAMETERIZE A REFERENCE PROGRAM

Once the output categories are defined, the programmer — drawing on experience and judgment — decides which categories are similar enough to be covered by a single template. For example, categories that differ only by whether to apply a statistical method or produce descriptive results only can often share one template with a conditional block, sharing the same generator. Recognizing these opportunities early reduces the number of templates that need to be built and maintained. For each template, select one representative output and write a working, validated QC program for it by hand. This is the reference program. The goal of parameterization is to transform this program into a template where every endpoint-specific value is replaced by a named parameter.

WHY PARAMETERIZATION MATTERS

Parameterization separates the structure of a program from its content. The structure — the sequence of data steps, procedure calls, macro invocations, and output formatting — is identical across all outputs in the same category. The content — which dataset, which variable, which visit weeks, which population condition — varies per output. By making this distinction explicit, parameterization provides three key benefits:

1. Traceability. Every endpoint-specific value is recorded in one place: the parameter registry (a CSV file). When a reviewer wants to understand why a program filters to a particular subpopulation, they look at the registry — not at the code. Changes are made in the registry, not scattered across dozens of individual files.
2. Consistency. Because all programs for a category are generated from the same template, structural errors are impossible to introduce selectively. A fix to the template propagates to every program in the category upon regeneration.
3. Speed of update. When a design change occurs — for example, a new visit is added to the schedule, or a population condition is corrected — the registry is updated and all affected programs are regenerated in seconds.

IDENTIFYING PARAMETERS

This step is a dialogue between the programmer and AI. The programmer provides the reference program; AI reads it and proposes an initial set of parameters — the values that are likely to vary across outputs. The programmer reviews, confirms, and adds any parameters AI may have missed or that require domain knowledge to identify. AI then revises the parameter list and refactors the program accordingly, replacing hardcoded values with placeholders. The programmer reviews again, and the cycle continues until the template is complete.

For example, a parameter such as the dataset name or endpoint code is straightforward for AI to identify. A parameter such as the subpopulation condition — which may depend on understanding the clinical meaning of a baseline flag — is more likely to require programmer input. The interaction is efficient precisely because AI handles the structural scanning while the programmer contributes the domain knowledge.

Once the parameter list is agreed upon, AI builds the Python generator script that reads the registry CSV and writes a standalone SAS file per row.

STANDALONE FILES OVER MACRO DRIVERS

An important design decision was to generate standalone SAS files rather than building a macro-driven framework that reads the parameter CSV at runtime. A macro driver approach requires all programs to be submitted together and the full CSV to be loaded each time — this becomes time-consuming when running individual outputs during development or debugging. Standalone files are easier to debug, easier to deploy, and easier to diff — version comparison shows exactly what changed between runs. The CSV remains the single source of truth, but the generated SAS files are self-contained artifacts.

STEP 3: PROVIDE MATERIALS TO AI FOR PARAMETER EXTRACTION

The most time-consuming part of the manual approach is populating the parameter registry: for each output, look up the dataset name, find the correct PARAMCD, determine the estimand flag, identify the visit schedule, and parse the subpopulation condition from the title or statistical analysis plan. This lookup work is well-suited to AI.

TYPES OF SOURCE MATERIALS

The programmer assembles the materials AI needs to extract parameter values. Four types of inputs are typically required:

1. ADaM dataset specifications (Excel). Each ADaM dataset has a metadata file listing all variables,

PARAMCD values, and their definitions. AI reads these files to identify the correct PARAMCD for each endpoint and the variable names for population flags and estimand indicators.

2. Output title list (Excel). The full list of output identifiers and their titles contains structured information: the analysis tier and estimand are encoded in the title pattern. AI parses the title text to determine which estimand flag applies to each output and whether statistical testing is required.
3. Supporting study code or reference files that carry general information across the study — for example, a visit schedule file defining which analysis weeks apply to each endpoint family. AI reads these files and maps each output to its correct parameter values without the programmer typing them manually.
4. The programmer's own understanding of the study. This is provided as rules in plain language: for example, which estimand flag corresponds to which analysis tier, or which subpopulation condition applies to which output family. These rules are what AI uses to map source materials to registry values.

TEACHING AI TO EXTRACT AND MAP

The rules provided by the programmer are the core of this step. Rather than asking AI to guess the mapping logic, the programmer states it explicitly — for example: outputs under the supplementary estimand are always descriptive and require no statistical test; outputs under the main estimand with binary endpoints use a CMH test; a specific analysis tier maps to a specific estimand flag variable. With these rules in hand, AI applies them systematically across all rows in the title list, populating the registry.

The approach that works best is to provide the rules together with one or two completed example rows, then ask AI to apply the same logic to the remaining outputs. AI fills the registry, flags any rows where the rules are ambiguous or conflicting, and the programmer resolves those exceptions. The important principle is that AI handles the bulk extraction while the programmer contributes the rules and resolves edge cases — not the reverse.

VALIDATION OF EXTRACTED PARAMETERS

After AI populates the registry, the programmer reviews it before generating code. Because all values for a given category follow consistent rules, a cross-tabulation of the registry columns is sufficient to identify outliers. Rows that deviate from the expected pattern are immediately visible and can be corrected in the CSV before any code is generated.

STEP 4: GENERATE, DEPLOY, AND MAINTAIN

With a validated registry and a tested template, code generation is a single command. The Python generator reads the CSV, substitutes parameters into the template for each row, and writes one SAS file per output. For approximately 175 outputs across three analysis categories, this takes seconds.

HANDLING TEMPLATE VARIANTS

Because the flag columns in the registry were already populated using the mapping rules in Step 3, the generator can use them directly to control conditional blocks. Not all outputs in a category are identical — some require additional analysis blocks, some include formal statistical tests while others are descriptive only. The generator checks the relevant flag value for each row and includes or excludes the corresponding code block accordingly, without any manual intervention per output.

PROPAGATING CHANGES

Two types of changes arise after initial generation:

1. Registry-level changes — where only certain outputs are affected. For example, a dataset name correction, a PARAMCD update, or a subpopulation condition change. These are handled by updating the relevant rows in the CSV. AI identifies all affected rows from a plain-language description of the change, produces the corrected CSV, and regenerates the affected files.

2. Structural changes — where the code template itself needs to change, affecting all outputs in a category. For example, a new variable must be added to every program, or a procedure call needs to be restructured. AI updates the template based on a description of the required change and regenerates all files in the category. Because all programs derive from the same template, the fix is made once and propagates everywhere.

BUILDING STUDY-SPECIFIC AI SKILLS

The study-specific knowledge accumulated through this process — how to extract parameters from the ADaM specifications, how to read the visit schedule file, the mapping rules between title components and estimand flags, which subpopulation conditions correspond to which output families — can be captured as a reusable skills document for AI. Once defined, these skills allow AI to onboard to a new output category or a new study amendment immediately, without the programmer re-explaining the rules each session. The programmer maintains this living document; AI references it at the start of any new task.

DOCUMENTATION AND PROVENANCE

The parameter registry is the documentation of record. Every decision about how an endpoint is coded — its dataset, its variable, its population condition — is recorded as a row and column value in the CSV. The CSV can be version-controlled, shared with the production team for cross-checking, and retained as a study deliverable. Conversation logs from each AI session provide additional provenance: they record what materials were provided, what rules were applied, and why exceptions were handled the way they were.

RESULTS AND LESSONS LEARNED

The approach was applied to a Phase 3 clinical trial with approximately 200 efficacy outputs. The outcome across three analysis categories was:

Category	Template Type	Registry Rows	Programs Generated
Continuous ANCOVA + MI	Single parameter	29	29
Binary response (CMH + Descriptive)	Single parameter	111	111
Continuous descriptive	Single + Multiple	35	35

Table 2. Programs Generated per Category

A total of 175 QC programs were generated from three template files and three registry CSV files. The efficiency gain comes from several directions: without automation, a programmer would likely maintain separate code for statistical and descriptive variants even when they share the same underlying logic; with a single parameterized template covering both, that duplication is eliminated. Initial generation is reduced to building the templates and registries once, after which all files are produced in seconds. And ongoing maintenance is transformed — a design update that would require touching dozens of individual programs manually becomes a registry edit or a template structure update, and a simple regeneration run.

KEY LESSONS

- Start simple, automate incrementally. Begin with one working program, parameterize it for one category, validate the output, then scale.
- The CSV is the contract. Everyone on the team — QC programmers, production programmers, biostatisticians — can read a CSV. Making the parameter registry the primary artifact improves communication and makes cross-checking easy.
- AI gets better as you give it more context. Early in the process, parameters were entered manually. Later, AI extracted parameters from metadata, titles, and reference files with minimal instruction once the mapping rules were established.

- Capture study-specific skills. The mapping rules, estimand logic, and extraction patterns developed during this process are valuable beyond the current task. Saving them as a reusable skills document means AI can onboard to a new output category or a study amendment immediately, without starting from scratch.
- Save conversation logs. Each AI session log is a record of what was decided and why — what materials were provided, what rules were applied, what edge cases were resolved. These logs are lightweight documentation that costs nothing to keep and can be invaluable when revisiting decisions months later.

CONCLUSION

This paper presented a four-step process for AI-assisted batch code generation for structured tables: categorize the output universe, parameterize a reference program, use AI to extract parameters from study materials, and generate and maintain programs from a parameter registry. QC programs for efficacy outputs served as the working example, but the approach is grounded in a property that applies broadly: whenever a table stream has a standardized shell and varies only by endpoint-specific logic, the same workflow applies.

The parameter registry provides traceability, the generator provides consistency, and AI accelerates the lookup and extraction work that would otherwise be done manually.

As AI tools become more capable, the proportion of the extraction work that can be automated will increase. The process described here is designed to accommodate that progression: the registry and template remain the stable artifacts, while the AI's role in populating the registry can expand over time.

REFERENCES

There are no external references cited in this paper. All methodology described is original work developed during study programming activities.

CONTACT INFORMATION

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

Qing Ma
Johnson & Johnson
qma25@its.jnj.com