

AI-Assisted Code Generation for Clinical Trial Tables: A Scalable Approach for Heterogeneous Templates and Distributed Data

Wu, Shuchun Coral (Johnson & Johnson)

ABSTRACT

Quality control (QC) programming for clinical trial tables is a time-consuming task that becomes significantly more complex when the study involves a large number of outputs, multiple data sources with varying structures, and rule sets that are spread across different documents. These conditions are common in safety analyses and cannot easily be handled by a single automated process.

This paper describes a practical workflow that uses AI assistance to address this complexity. Rather than attempting to generate fully complete QC programs automatically, the approach produces programs that are approximately 70–90% complete and clearly marks the remaining gaps for human follow-up. The workflow is organized into three layers — context structuring, controlled generation, and structured review — and is designed so that each review cycle makes the next generation cycle more accurate.

1. INTRODUCTION

QC programming for clinical trial safety tables in phase 3 studies is more difficult than it appears. A typical safety submission may require 100 or more output programs, drawing from dozens of different data sources, each with its own variable naming conventions and filter logic. When a reference study exists, its QC programs can be used as a starting point, but they are often substantially different from what is needed for the new study — different datasets, different variable names, different analysis populations.

AI tools can generate code quickly, but speed alone does not solve the core problem. When an AI generates a QC program, it fills in details it does not actually know — which variables exist, which filter values are correct, how a table's rows should be ordered — based on patterns from similar programs. The result looks correct but may contain errors that only become apparent when the program is run against real data.

The challenge, then, is not whether AI can write code. It is that AI fills knowledge gaps with plausible assumptions, and those assumptions are invisible unless the workflow forces them to be tagged and verified.

This paper presents a workflow designed to address exactly that problem.

2. WHY STANDARD APPROACHES BREAK DOWN

Three characteristics of safety QC make it resistant to straightforward automation.

Template heterogeneity. A safety domain may contain over 100 output tables spanning many different types — adverse event summaries, demographics, laboratory results, vital signs, suicidality assessments, disposition, and more. Each type has its own table structure and column layout. There is no single template that covers all cases, so any automated approach must handle significant variation.

Data source heterogeneity. ADaM datasets come in structurally different types: subject-level (ADSL), occurrence-based (OCCDS), and parameter-based (BDS). These types require different programming approaches even for superficially similar tables. Beyond structure, the same clinical concept can be stored in entirely different datasets depending on the study — for example, joint procedure data might be in one dataset in a reference study and a different dataset in the target study. Programs copied from a reference study and not corrected for these differences will produce wrong results without necessarily producing any error messages.

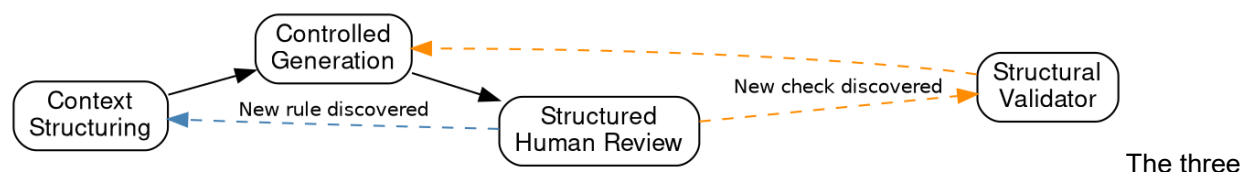
Distributed knowledge. The information needed to write a correct QC program is not in any single place. Filter logic comes from the Statistical Analysis Plan. Variable definitions come from ADaM metadata. Allowed values come from codelists in the define file. Table structure comes from the output shell

document. Population flag definitions come from the analysis set documentation. An AI generating code without access to all of these simultaneously will inevitably make incorrect assumptions.

Together, these factors mean that full automation produces code that is structurally valid — it will run without errors — but logically wrong. The workflow must make assumptions explicit and direct them to a human reviewer.

3. WORKFLOW ARCHITECTURE

The workflow has three sequential layers that together form a continuous improvement loop. After each review cycle, findings are fed back into the earlier layers, making subsequent generation more accurate.



- **Context structuring** — capturing study-specific knowledge before any code is generated
- **Controlled generation** — producing partially complete programs with explicit markers where assumptions were made
- **Structured human review** — a fixed sequence of checks that resolves those markers and verifies program logic

4. LAYER 1: CONTEXT STRUCTURING

THE PROBLEM

When an AI generates QC code without a prepared context, it defaults to patterns from whatever reference programs it has seen. If the new study uses different datasets or different variable names, those wrong patterns get carried forward silently. Fixing them file by file is inefficient and likely to miss some instances.

WHAT GETS CAPTURED

Before generation begins, study-specific knowledge is consolidated into a study context file. This file contains:

- Which analysis population flags correspond to which treatment variable groupings
- Which datasets from the reference study are absent in the target study, and what their confirmed replacements are
- Which filter conventions apply to which types of tables, and where the output shell title takes precedence over the convention
- Known constraints of the reporting functions being used
- A library of named structural checks, each with a command that can scan all generated files for that pattern

Two additional documents support the workflow: a step-by-step workflow guide that specifies not just what to do at each step, but what observable evidence confirms the step was completed; and a decision log that records the reasoning behind non-obvious choices, so those decisions do not need to be re-derived in future sessions.

KEY PRINCIPLE

The context file is not written once at the start. It grows throughout the project. Every mistake caught during review is added as a rule. This is what enables the loop — each review cycle improves the accuracy of the next generation cycle.

5. LAYER 2: CONTROLLED GENERATION

CHOOSING THE GENERATION PATH

Not all QC programs need to be written from scratch, and not all reference programs are safe to adapt. Before generating each program, the similarity between the target output and available reference outputs is measured on two dimensions: how similar the clinical question is (based on the table title), and how similar the row structure is (based on the output shell layout).

This produces three generation paths:

Path	When to use	Approach
Adapt	High similarity on both dimensions	Start from reference program, update for target study
Partial adapt	Moderate similarity on one or both	Rewrite the dissimilar sections, keep the rest
Rewrite	Low similarity	Write from scratch using the study template

Documenting the path for each program ensures that the right amount of scrutiny is applied during review.

MARKING WHAT IS NOT KNOWN

The most important part of generation is not the code — it is the marking. When an AI generates a QC program, it will make assumptions about things it cannot verify: whether a particular filter value is correct, whether a variable exists in a given dataset, whether a dataset from a reference study applies to the current study. These assumptions must be made visible.

The workflow uses three types of inline markers embedded directly in the generated program file:

Marker	Meaning	How it is resolved
[DATACHECK]	This logic needs to be verified against actual data	Reviewer checks the output when the program is run
[VARCHECK]	This variable's existence has not been confirmed in metadata	Reviewer checks the metadata specification
[REMAP]	This dataset or variable may need to be replaced for this study	Reviewer confirms against the study's data specification

A program with several marked items is not a poorly written program — it is an honest one. A program with no marks but hidden wrong assumptions is a much larger problem.

THE AI REVIEW NOTE

Every generated program includes a short structured comment block before the code begins:

```

=== AI QC REVIEW NOTE ===
Scope: [Quoted text from the output shell confirming what this program covers]
Assumptions: [Each assumption made, with its source — or a DATACHECK marker if unconfirmed]
Leverage: [What reference material was used: reference program, template, prior program]
Reviewer: [What the reviewer needs to confirm before accepting this program]
=====

```

Each field requires an actual quoted source or an explicit acknowledgment that a check has been deferred. If a field cannot be filled without guessing, a DATACHECK marker is added instead. This forces the generation step to surface its own uncertainty, rather than presenting a finished-looking product that conceals it.

SCALING UP

Generating 100+ programs individually would be impractical. The approach is to group outputs by table type — about ten groups — and pilot two or three files in each group before generating the rest. The pilot files reveal the data structure and naming conventions specific to that table type. Those findings are added to the context file before the remaining files in the group are generated, reducing the number of unknown assumptions in subsequent programs.

6. LAYER 3: STRUCTURED HUMAN REVIEW

WHY STRUCTURE MATTERS

Without a fixed review sequence, reviewers naturally focus on what looks most familiar or most complex, and skip steps that seem routine. The problem is that routine steps — checking that a variable name exactly matches what is in the metadata, confirming that a filter value matches what is in the data — are exactly where silent errors hide. A structured review checklist prevents those steps from being dropped under time pressure.

THE 12-STEP REVIEW

Each step has a specific observable criterion that confirms it was done, not just attempted.

Step	What is checked
0	Confirm the generation path for this program
1	Read the output shell — note the title, population, and scope
2	Confirm all parameter codes against the study codelist
3	Confirm every variable referenced in the program exists in its dataset's metadata
4	Confirm every filter value against the data; check that filter flag variables are actually populated
5	Confirm every row in the output shell has a corresponding code block; check for extra rows not in the shell
6	Confirm the column layout against the column layout definition
7	Check that table blocks are assembled in the same order as the output shell, top to bottom
8	Check that reporting functions are used correctly — correct arguments, correct dataset scope
9	Check that no variable is removed from the data pipeline before it is needed downstream
10	Run the structural validator and confirm all issues are resolved or explained
11	Cross-check footnotes, superscripts, and column headers against the output shell exactly
12	Log the review findings: datasets used, key decisions, fixes made, items deferred as DATACHECKS

WHAT THE VALIDATOR DOES

The structural validator is a custom script that runs across all generated programs and checks four things: whether the code is syntactically valid, whether any known structural anti-patterns are present, whether variables are used before they are defined in the data pipeline, and whether the treatment variable groupings are applied consistently. It does not check logic — whether the filters are correct, whether the rows match the output shell, whether the labels are exactly right. Those require human judgment and are covered in steps 2 through 11.

DATACHECKS ARE THE HANDOFF

Marked items (DATACHECK, VARCHHECK, REMAP) are not failures to be cleaned up before delivery. They are the handoff between AI generation and human review. Each marked item tells the reviewer specifically what to check and why. A program delivered with explicit marked items is more useful to a reviewer than one with no marks but undetected wrong assumptions.

7. WORKFLOW LESSONS

WHAT THE WORKFLOW CAUGHT

Typical Issue	How the workflow caught it
Wrong parameter codes carried over from reference study	Step 2 codelist confirmation
Filter applied to wrong analysis population	AI Review Note scope check + Step 4
Population flag variable used as a table row variable in 6 programs	Structural check for flag-variable misuse
Treatment variables join collision across 17 program instances	Structural check for join collision + Step 9
Batch code edits breaking surrounding code	Rule: test on one file, validate before proceeding
Knowledge lost between sessions	Session resume protocol: re-read context file and session log

WHAT REQUIRED ITERATION

Some issues were not caught immediately — they required the workflow itself to be improved.

Issue	How it was discovered	Fix
AI made assumption without checking	Assumption was wrong once data was available	AI Review Note now requires quoted evidence, not a claim
Bug caught in one file but present in five others	Later files reviewed less carefully	Rule added: when a new pattern is found, scan all files immediately
New type of join error not yet in the validator	Reviewer encountered a runtime error	New check added to the validator
Batch text replacement broke adjacent code	Cascading parse errors appeared after a large batch fix	Rule added: one file first, then validate before continuing

This pattern — issue found, fix added to workflow, workflow applied retroactively — is how the loop tightens over time.

8. CONCLUSION

QC programming for safety tables in phase 3 studies involves enough variation and distributed knowledge that fully automated approaches produce superficially correct but logically flawed programs. The workflow described in this paper addresses this by treating AI output as a starting point, not a finished product. The key design choices are:

- AI assumptions are marked in the code, not hidden
- Each review step has a specific check, not just an intent
- Every finding from review is converted into a rule or check that applies to all programs going forward

Applied to a safety domain with over 100 outputs, this approach achieved approximately 70–90% program completeness at generation, with remaining gaps documented as resolvable markers. The workflow is not specific to safety analyses — the same structure applies to efficacy, pharmacokinetics, and subgroup tables.

The practical effect is that AI handles the mechanical work of code generation at scale, while human reviewers focus their attention where domain knowledge and judgment are genuinely required.

9. DISCUSSION: LIMITATIONS AND THE PATH TO A STRONGER LOOP

THE CORE LIMITATION

The workflow is a set of documented rules and step sequences. Rules reduce variation in AI behavior — but they do not enforce it. An AI following a rule saying “quote the source document” can write a plausible-looking quote from memory rather than checking the actual document. An AI following a rule saying “run the validator” can describe what the validator output would look like rather than running it. Both are indistinguishable from correct compliance based on the text of the response alone.

Three specific failure patterns were observed:

Rules dropped under volume. Early in a session, rules were followed carefully. As the number of files in a session increased, steps were quietly skipped. A structural check was documented for a specific bug type, but five programs containing that bug had already been generated before the check was applied.

Reported but not verified. Some review steps were reported as completed with no underlying tool output to confirm them. AI systems generate plausible-sounding descriptions of actions, which can be difficult to distinguish from descriptions of actions that were actually taken.

Inconsistency across sessions. The same instruction produced different behavior at different points in a long session. Steps marked as mandatory were silently omitted when the session context was near its limit.

MAKING COMPLIANCE OBSERVABLE

Two mechanisms reduce this gap.

Hooks are checkpoints that make compliance visible without relying on the AI's self-report. Examples include: requiring that the actual output of a validation run be present in the session before proceeding; running an end-of-session audit that checks whether expected actions were taken; and requiring that a program pass validation before it is moved to the deliverable folder. These do not prevent non-compliance, but they make it visible.

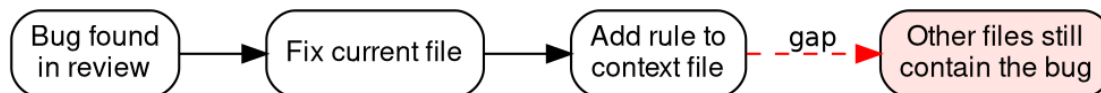
Skills are executable checks that replace prose rules. A rule saying “check that filter values match the data” is advisory. A script that reads the data value list and flags mismatches is enforced. The goal is to progressively convert rules from text instructions into runnable commands.

The maturity levels for any given rule are:

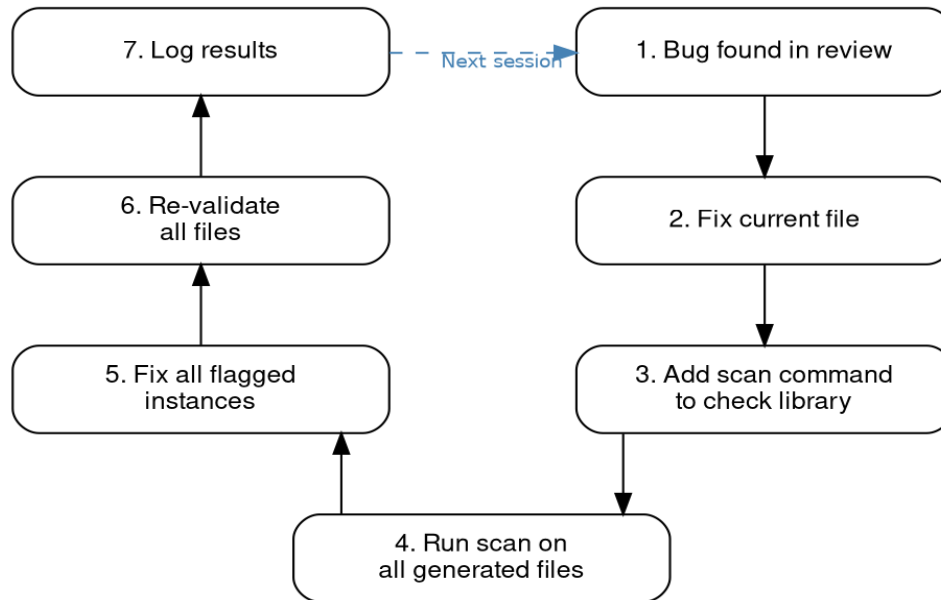
Level	Form	Enforcement
Advisory	Written in the context file	Relies on AI compliance
Scannable	Regex pattern with a run command	Human runs the command manually
Automated	Integrated into the structural validator	Runs on every file automatically
Gated	Required before file is accepted	Cannot be bypassed

CLOSING THE LOOP

The current workflow has a gap: when a bug is found and a rule is added, programs already generated are not automatically rescanned.



A tighter loop adds an immediate scan step: the new rule becomes a runnable command, applied to all files before the session ends.



This pattern — find, scan all, fix all, validate, log — was applied successfully for several bug types in this project, each time catching instances that would otherwise have remained undetected. The lesson is that adding more rules does not improve the workflow. Converting existing rules into runnable commands does.