PharmaSUG 2026 - Paper DS-151

DQO: A Rule Engineering and AI-Driven Platform for Assessing Source Data Quality from SDTM Perspective

Nana Yang, Huan Song, Zhouhao Pan and Chunpeng Zhao, BeOne Medicines

ABSTRACT

DQO (Data Quality Overview) is a rule engineering platform implemented in Python with a Streamlit front end, designed to assess source data quality from SDTM perspective. It addresses persistent challenges in clinical data quality workflows—fragmented rule sources, inconsistent implementation, high maintenance cost, and limited reuse—by connecting the full rule lifecycle within a single operational framework: specification intake, Programming Algorithm generation, Python code generation, script execution, standardized output, and diff-based traceability.

The framework supports two complementary execution paths. When a validated implementation already exists in the rule library, the framework routes the check directly to that script, preserving consistency and continuity. When no suitable implementation exists, an AI-assisted authoring path generates programming algorithm and candidate Python code from structured inputs—rule description, variable mappings, and dummy example data—before execution and result output. As for this writing, the DQO rule library covers approximately 80 checks spanning most SDTM domains and continues to grow. Large language models serve as controlled drafting accelerators; all generated artifacts remain subject to human review and governance.

Unlike tools focused primarily on script execution, DQO emphasizes rule engineering. Programming algorithm function as an interpretable intermediate layer between business rule definitions and executable logic, reducing information loss during translation and strengthening long-term maintainability, reuse, and auditability.

INTRODUCTION

Clinical data review teams routinely define checks to identify missing, inconsistent, or clinically implausible records before downstream analysis, listings, and submission activities. In many organizations, however, these checks remain scattered across spreadsheets, reviewer instructions, programming algorithms, and ad hoc scripts. While sufficient for individual interpretation, such artifacts do not by themselves create a consistent, traceable, or reusable execution model.

A further challenge is the wide variation in rule complexity. Some checks can be executed repeatedly through stable, existing implementations; others depend on study-specific record ordering, derived thresholds, cross-domain comparisons, or domain-specific exclusion logic. Teams therefore rely on a mix of reusable programs and project-specific code, and many ad hoc scripts have limited carry-forward value. As rule volume and study complexity grow, the cost of re-authoring, re-interpreting, and re-validating similar checks accumulates.

DQO was developed to address this operational reality. We all know that edit checks play a critical role in the data cleaning process by automatically identifying and flagging erroneous, missing, or inconsistent data at the point of entry or during routine review, enabling timely query generation and resolution before database lock and analysis. While edit checks are predefined validation rules embedded within the EDC system to ensure data accuracy and consistency during data capture, DQO operates downstream on standardized datasets to systematically assess data quality, detect residual issues, and provide traceable outputs for data review and analysis readiness. In this sense, edit checks focus on real-time data validation, whereas DQO focuses on post-collection quality assessment and reporting, complementing each other across the end-to-end data lifecycle. DQO is implemented on SDTM rather than EDC data because SDTM offers a standardized and harmonized data structure across studies, making the outputs easier to interpret and compare. This standardization also enables the development of reusable, study-

agnostic programming code. In contrast, EDC data are typically study-specific and heterogeneous, often requiring one-off programming efforts for each individual study. DQO combines direct reuse of validated implementations with a structured generation of new custom Python code and preserves traceability from specification through implementation to execution output. An AI-assisted authoring layer accelerates programming algorithm and code drafting, while keeping human review at every governance checkpoint. The result is an end-to-end operating model that connects specification intake, implementation authoring, execution, and finding generation within one maintainable workflow.

SYSTEM ARCHITECTURE

FRAMEWORK OVERVIEW

The framework is organized as an end-to-end pipeline that transforms row-level review specifications into executable source data validation checks. It begins with specification intake, proceeds through artifact generation and execution routing, and ends with issue reporting and audit-ready artifacts. SDTM Plus data—standard SDTM variables supplemented with additional EDC system variables such as Instancename, RecordId, and so on that facilitate traceback to raw source records—serves as the primary data input. The Streamlit layer provides the user-facing entry point, while Python modules perform script selection, code generation, execution control, and result assembly. As shown in Figure 1, Users first provide a study folder and DQO specifications; the framework then decides whether an existing validated script can be reused or whether a new implementation path is needed; next, the selected or newly generated logic reads SDTM Plus input files, executes the rule, and assembles a reviewer-facing issue report. DQO is not only a script runner, but a connected operating framework from specification intake to issue delivery. The workflow also includes a feedback loop through which review results can be used to update the Programming Algorithm and code. This hybrid design allows the same infrastructure to handle both standard repeatable checks and study-specific complex rules without requiring separate, disconnected workflows. Table 1 summarizes the major technical components and their outputs.

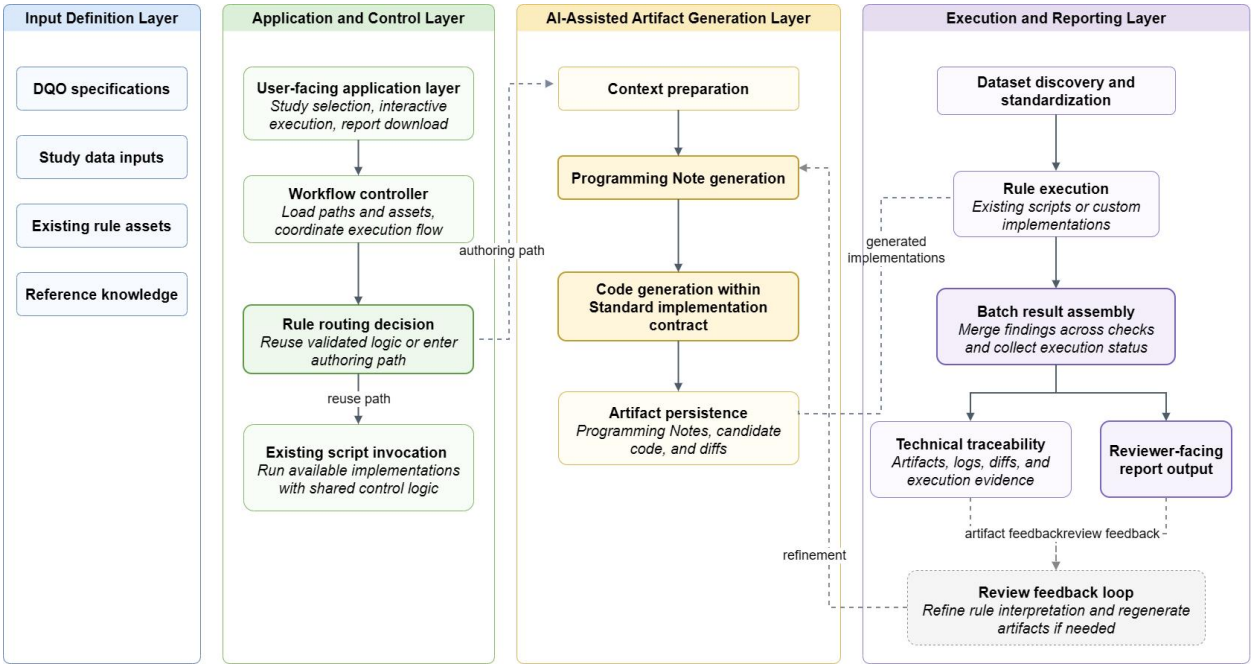

Figure 1. Hybrid Framework for Operationalizing Source Data Quality Checks Based on SDTM Plus Data

Table 1. MAJOR COMPONENTS OF THE FRAMEWORK

Component	Primary Purpose	Example Output
Specification intake	Read row-level review metadata and review text	Structured check context
Programming Algorithm generation	Translate business wording into implementation-oriented instructions	Programming Algorithm text
Custom code generation	Draft Python implementations for complex review logic	Executable code artifacts
Interface-based program debugging assistance	Analyze scripts against problem context and generate modification recommendations	Diagnostic findings and update suggestions
Validation execution	Run existing scripts or procedural Python implementations	Finding-level result set
Traceability and audit support	Preserve generated artifacts and logic changes	Diff files, logs, exported reports

INPUT DEFINITION LAYER

The system architecture begins with the premise that reviewing information, although often scattered across operational documents, should ultimately enter a more stable and coherent processing path. The content is more coherently understood as the Input Definition Layer. As shown in the hybrid framework diagram, this layer consolidates the materials that define what a review check means before any execution decision is made. These inputs include the DQO specification itself, study-level SDTM Plus data folders, existing rule assets, and supporting reference knowledge. Operationally, these materials correspond to the study input directories, the rule proposal workbook, the reusable rule repository, and the mapping and dummy-data references used elsewhere in the framework.

The importance of this layer is not limited to file loading. Its architectural role is to establish a shared interpretation boundary for downstream logic. The specification workbook provides row-level business context such as report check number, review messages, issue detail description, logic instruction, and risk rating. By standardizing key fields at the outset, the framework can carry the original review intent forward with less distortion into later stages of authoring, execution, and result organization. This row-based design also allows the same infrastructure to support both individual review checks and larger collections of checks, thereby creating a practical foundation for reuse and scaled processing. This emphasis is important because many validation checks do not originate as formal machine-readable logic. Instead, they begin as business-facing review descriptions that must later be translated into an executable form. The study data folders provide the actual domain files to be checked and treat them as the common point of departure for downstream work. Existing rule assets indicate whether a stable implementation already exists, and reference knowledge helps preserve consistency when the same business concept is translated across studies. Grouping these materials into one input-definition layer makes the framework easier to explain because it shows that DQO does not begin with code alone; it begins with a controlled combination of business definitions, study data, and reusable knowledge.

APPLICATION AND CONTROL LAYER

The Application and Control Layer is where the Streamlit application, workflow controller, rule-routing logic, and existing script invocation path operate together. Through one interface, the framework receives the selected study and specification context, then decides how each review check should be operationalized. This means that reuse and custom handling are not independent workflows; they are both governed by the same control layer.

Within this layer, the first responsibility is orchestration. The Streamlit entry point allows users to select studies, trigger rule execution, and export outputs, while the underlying controller loads available rule assets, reads cached programming artifacts, and coordinates batch execution. The second responsibility is routing. For each check, the control logic determines whether a validated implementation already exists and can be invoked directly. If the answer is yes, the framework follows the existing script invocation path so that stable logic can be applied with continuity and low re-authoring cost. This is particularly relevant in practice because many review checks occur across studies while their core logic remains relatively stable.

For such rules, reuse of existing scripts helps maintain continuity of execution, lowers operational cost, and allows teams to focus their effort on study-specific issues that truly require judgment and adjustment. If the answer is no, the framework does not break into a disconnected manual process; instead, it routes the check into the downstream AI-assisted authoring layer and later returns the generated implementation to the same execution environment, then user can review and maintain. This layered interpretation also clarifies how complex procedural checks fit into the framework. Rules involving sequencing, grouped comparison, threshold derivation, or domain-specific exclusions are not treated as exceptions outside the system. They remain under the same application and control layer, which decides that they require a different implementation path while still preserving common execution control, output structure, and traceability. As a result, direct reuse, study-specific procedural logic, and later AI-assisted generation all remain part of one controlled architecture rather than three disconnected operating models.

Why use Streamlit? Streamlit is a lightweight Python framework for building interactive web-based applications directly from Python code. In this framework, Streamlit serves as the user-facing application layer rather than as the execution engine itself. It provides a unified interface through which users can select study inputs, trigger rule execution, review run-time behavior, and export issue reports without interacting directly with command-line scripts or multiple disconnected utilities. This design is particularly suitable for the DQO framework because the underlying workflow is already Python-centered and involves structured file inputs, rule routing, batch execution, and tabular output delivery. By using Streamlit, the framework can expose these functions through a single operational workspace with relatively low development overhead, thereby improving usability while preserving the modular Python architecture beneath it. In this sense, Streamlit is not introduced as a general front-end technology choice, but as a practical application layer that makes the rule-engineering workflow easier to operate.

AI-ASSISTED ARTIFACT GENERATION LAYER

The AI-Assisted Artifact Generation Layer is designed as a staged authoring process rather than a single-step code generator. As shown in the AI code generation flow, the process begins only after the control layer identifies a rule gap, namely a Rule ID. for which no suitable executable Python script is available. At that point, the framework enters a dedicated artifact-generation path, making the AI pathway a conditional branch inside the broader workflow rather than the default execution mode for all checks.

Once this branch is activated, the framework first prepares a structured context package before requesting any generated output. This context is assembled through shared utilities and includes the review of rule text, relevant mapping-summary content, dummy dataset descriptions, and a filtered subset of field-level reference information judged to be most relevant to the current rule. On that basis, the first AI output is not executable code but an implementation-oriented Programming Algorithm which convers rule natural language and rawdata condition to SDTM variable condition. This intermediate artifact is then cached so that the rule of interpretation is retained as an explicit, reviewable knowledge object before the workflow proceeds to code generation. They can support programmer refinement of programming algorithms, and they can also preserve interpretable difference information after code changes, so that drafting, review, and regeneration do not become disconnected stages. In other words, AI accelerates drafting rather than replacing review. It helps teams reach a discussable, comparable, and revisable version more quickly, but whether that version is accepted and how it is adjusted still depends on subsequent human review and governance. In clinical data review settings, this arrangement is more practical because the critical question is not how quickly something can be generated, but whether the generated content continues to reflect the original review intent under ongoing review. The second AI output uses that Programming Algorithm as its immediate basis for generating candidate Python implementations. Importantly, this step is constrained by a standard implementation contract, including a consistent function signature, expected result-column structure, and shared execution conventions such as logging and cutoff-date handling. The generated assets are then written to the rule output directory as Python, and diff-style files, after which the newly created Python implementation is immediately routed back into the same execution pathway used by pre-existing rules. In this sense, the layer does not end with artifact storage; it closes the loop by turning generated artifacts into operational rule objects that can be executed on real study data while remaining visible for human review, revision, and traceability. This sequence is important because it shows that AI is introduced as a controlled drafting layer rather than as an opaque end-to-end decision maker. Table 2 provides an example of how edit check logic can be converted into a DQO programming algorithm by AI.

Table 2. EXAMPLE OF EDIT CHECK LOGIC AND DQO PROGRAMMING ALGORITHM

Rule ID	Edit Check Logic	DQO Programming Algorithm
AE001	Output if [DS_EOS.DSSTDAT2] is entered and [AE.AEENDAT] is null.	Output when DSSTDTC is entered where EPOCH is TREATMENT or FOLLOW-UP and AEENDTC is null.

REPORTING AND TRACEABILITY

At the final end of the architecture, the framework organizes execution results together with the supporting materials needed for later review, explanation, and maintenance. The emphasis here is not simply on producing outputs, but on preserving the relationship between outputs and the process that produced them. Reviewer-facing issue reports, code artifacts associated with implementation, and the difference information, runtime logs, and related explanatory materials retained during updates are treated as parts of the same traceability chain rather than as isolated files. In this way, the final deliverable is not merely a list of findings, but a set of materials that can jointly answer where a finding came from, what logic produced it, and how the current implementation differs from an earlier version.

From a practical delivery perspective, the report is better understood as a structured issue report organized at the level of individual findings rather than as a simple list of hits. In many cases, each row corresponds to a specific issue, with the front portion of the report used to locate the associated check, subject, page, or record position, and the later portion used to capture issue description, relevant variable values, rating, and follow-up review information. Once rule-level issue records have been created, the framework moves into a batch assembly stage. The batch controller merges outputs across rules into a unified finding-level result set and, in parallel, captures execution failures in a separate error structure rather than allowing them to disappear silently. The subsequent report-export step then reshapes the merged results into a report-ready form by adding review-oriented columns such as Flag and Comments, deriving fields, and arranging columns in a sequence that supports business reading. In that sense, the report is not merely an accessory generated after execution, but a primary vehicle through which issue identification, issue localization, and result interpretation are tied together.

This arrangement is particularly important in practice. Review teams usually need to explain not only what was found, but also which check produced the finding and whether the current implementation has changed relative to a prior version. Reporting and traceability should therefore be understood not as optional add-ons after execution, but as necessary architectural features that support long-term maintenance, reviewability, and reuse. This is especially important when rules undergo repeated revision: if issue reports, code differences, runtime traces, and explanatory information about changes are retained together, teams can more easily interpret a given execution result in the context that produced it and can more confidently decide what may be carried forward and what still requires adjustment in later studies or later versions. In other words, the reporting output here functions as a structured issue report: it supports not only finding detection, but also issue localization, result interpretation, and downstream traceability.

Figure 2 brings together the full handling logic for an individual review check, from assessment of rule context and implementation choice to downstream artifact assembly. The workflow begins by assessing rule context and implementation need, then determining whether a reusable validated implementation already exists. For existing implementations, the process further considers whether study-specific parameterization or controlled adjustment is needed. When no suitable implementation is available, the framework moves into a custom path that includes programming algorithm drafting, AI-assisted artifact generation, programmer review and refinement, and execution of the custom implementation. The process then enters a reporting and traceability stage in which issue reports, supporting artifacts, and audit-trail information are assembled before results are consolidated into a validation and traceability package. The figure also highlights both when an existing validated script can continue to be reused with controlled adjustment and when study-specific needs require the workflow to proceed into the downstream custom implementation path; if review, refinement, or result analysis identifies remaining gaps, the workflow can iterate back to earlier authoring steps to update programming algorithm, code, and related difference artifacts. Table 3 is summary information of DQO report.

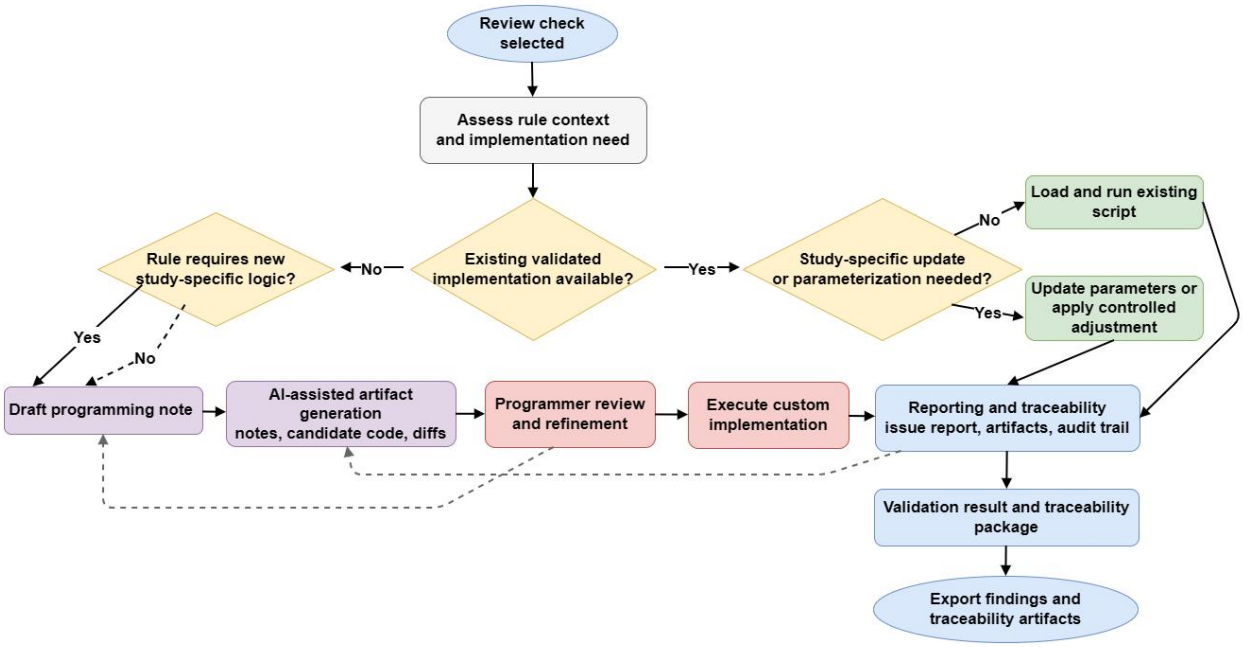

Figure 2. Rule Routing Logic for Existing and Custom Execution Paths

Table 3. SUMMARY OF KEY INFORMATION CATEGORIES IN THE DQO ISSUE REPORT

Information Category	Purpose in the Issue Report	Representative Content
Study and subject context	Identifies the study, site, and subject associated with each finding.	Study identifier, site information, subject identifier
Check identification and severity	Links each finding to the corresponding review rule and supports prioritization.	Check number, review message, DQO rating
Source record traceability	Helps reviewers locate the source record or form associated with the finding.	CRF page, record reference information
Issue-specific variable summary	Provides a compact view of the variables and values that triggered the finding.	Relevant SDTM variables and corresponding values
Review follow-up information	Supports downstream review, comment capture, and issue disposition.	Flag, comments, reviewer notes

USER INTERFACE

The Streamlit-based system interface demonstrates how DQO translates the underlying rule-engineering framework into an accessible operational workspace. Rather than requiring users to move between separate scripts, configuration files, temporary outputs, and manual execution steps, the interface consolidates the core activities of the workflow into a single application layer. These activities include specification upload, study-level parameter entry, targeted rule selection, execution control, debugging support, and report export. As shown in Figure 3, the user first uploads the review specification and then loads the available check definitions through the Programming Note Editor. After a specific Rule ID. is selected, the user provides the path to the study-level SDTM Plus datasets and initiates the run from the same interface. The system then determines whether an implementation-oriented Programming Algorithm already exists for the selected check. If no such artifact is available, the AI-assisted authoring path is triggered to generate the Programming Algorithm and the corresponding Python implementation before execution proceeds. The resulting findings are then displayed in the Result panel, allowing users to review the check output without leaving the application environment. This interface design is important because it makes the framework operationally usable by non-developer users while still preserving the modular rule-routing, artifact-generation, and execution logic beneath the surface. In other words,

Streamlit functions not simply as a front-end screen, but as the practical control point through which specification intake, rule authoring, execution, and result review are connected in one workflow.

Platform also includes an interface-based program debugging assistance function to support investigation of potential issues in check scripts. At run time, the system submits the target program together with relevant problem context to the model, which then analyzes likely update locations and generates modification recommendations in “Conversation” panel. Because program backup, version control, and change review requirements remain important, this function does not yet automatically rewrite the script. Instead, it is intentionally limited to generating update recommendations for programmer review and manual implementation. This design allows the model to assist with issue localization and revision suggestions while keeping final code changes under human control.

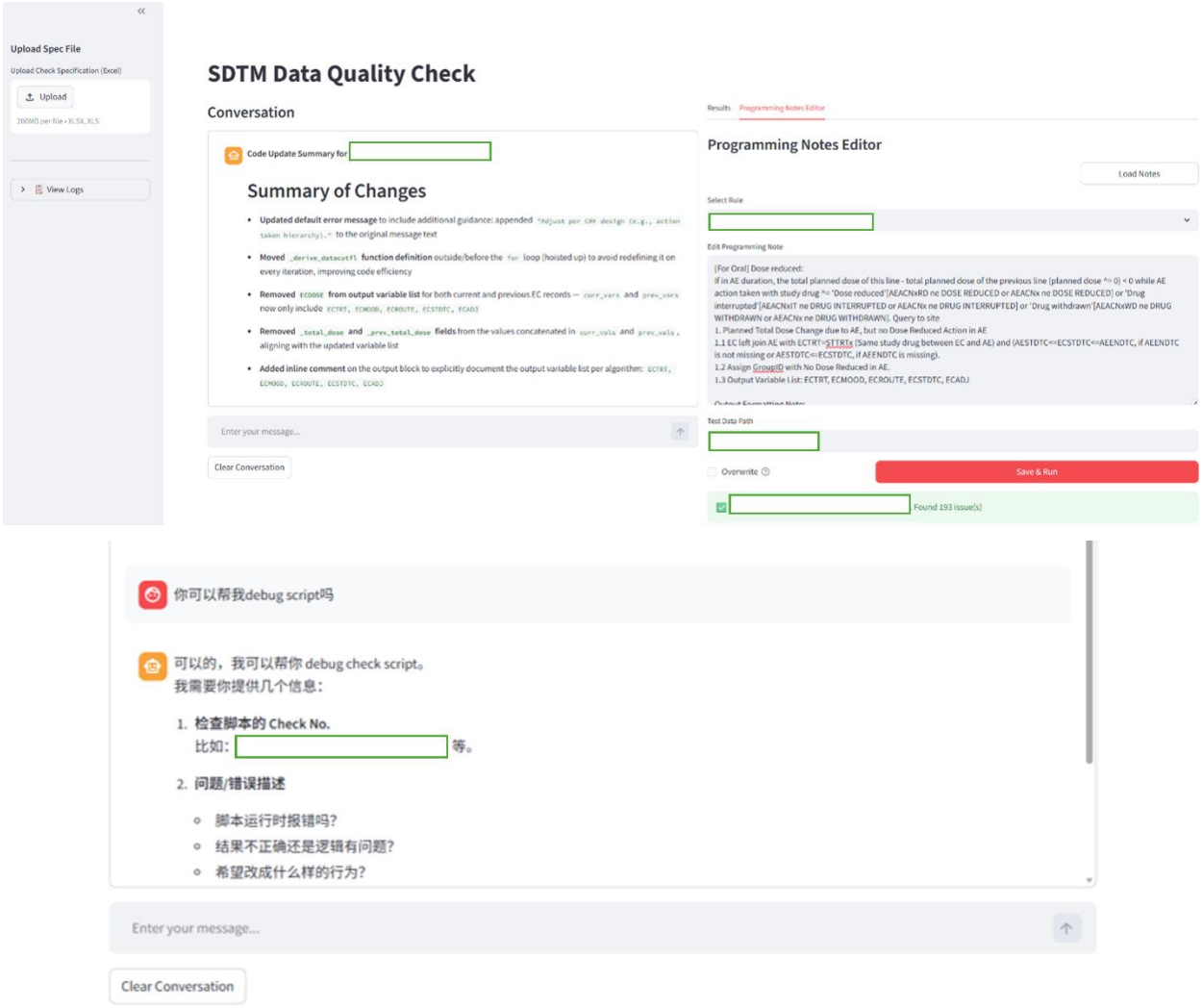

Figure 3. Streamlit-Based System Interface for Review Check Execution and Program Debugging Assistance

CASE STUDY: EC GAP DETECTION FOR ORAL EXPOSURE RECORDS

A useful illustration of the procedural path is a review check that identifies gaps between consecutive Exposure as Collected records for oral treatment. In operational review, such a check can help identify discontinuities in recorded dosing periods that may require follow-up. In the implementation, the check reads the EC dataset from an SDTM plus data directory, standardizes variable names, and verifies that required fields are present. It then applies domain-aware filtering, such as focusing on oral records and excluding scheduled contexts when appropriate. Date fields are parsed into comparable values, and

records are ordered by subject, treatment, and timing. The rule then evaluates consecutive records within subject and treatment groups. When the time gap between the end of one record and the start of the next exceeds a threshold, both records are written to the output with a shared group identifier so that reviewers can evaluate the pair together. If a threshold is not supplied, the implementation derives a default value based on the most frequent observed gap in the dataset.

This example is instructive for several reasons. First, the review check depends on record ordering and pairwise comparison rather than simple row-level conditions. Second, it uses business-aware filtering and grouped output construction. Third, it demonstrates why some review logic is more naturally implemented procedurally than as a fixed reusable script template. The resulting output includes study-level and subject-level identifiers, record-level traceability, selected contextual fields, and a compact variable-value summary. This structure improves reviewer usability while still preserving the provenance of the finding. From an architecture perspective, this case study validates the decision to support a procedural execution path. The review check is easy to understand in business language, but its implementation depends on ordered comparison, contextual filtering, and grouped result assembly. A focused Python implementation is therefore more maintainable than forcing the logic into an oversimplified template. Figure 4 illustrates the procedural sequence used for a custom EC review check. The workflow loads EC records, applies business-specific filtering, derives or accepts a gap threshold, compares consecutive records within subject and treatment groups, and assembles grouped findings for reviewer follow-up.

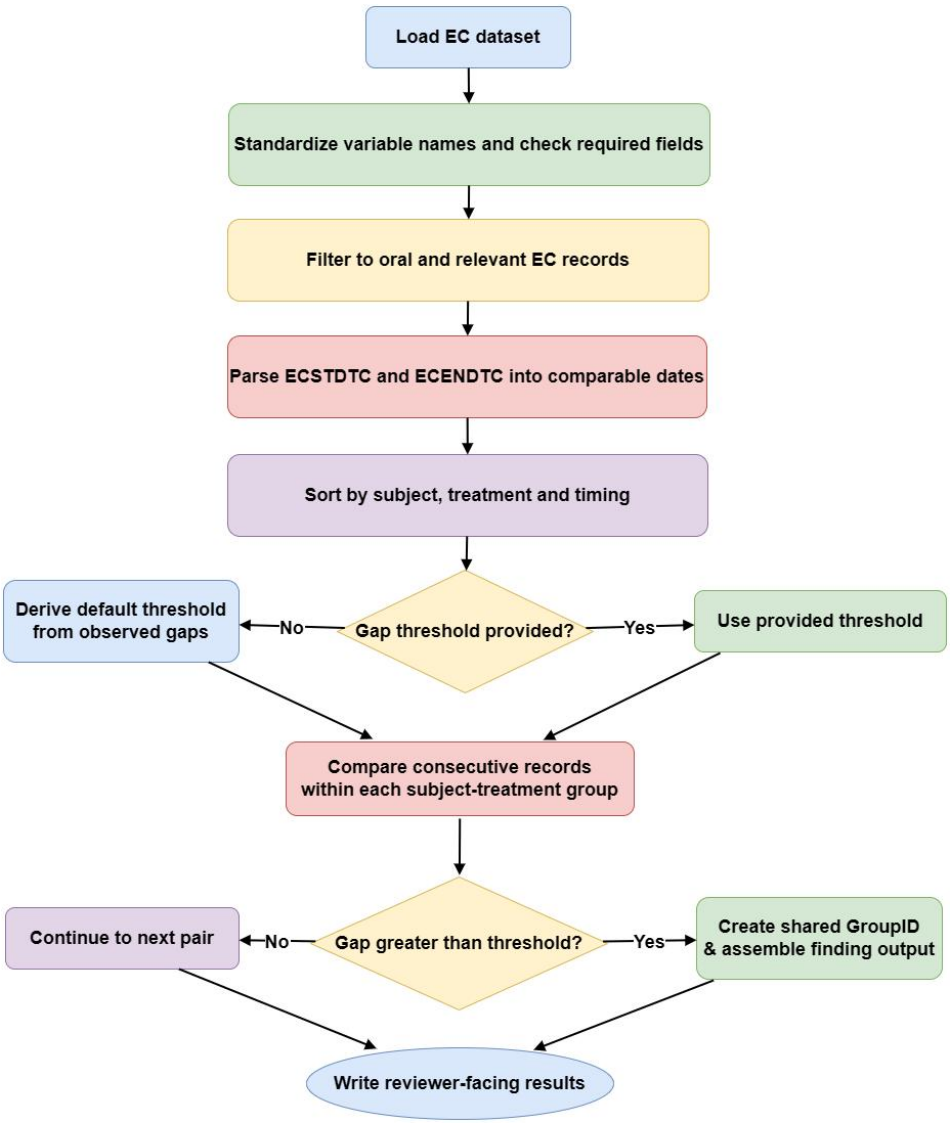

Figure 4. Procedural Logic Example for EC Oral Exposure Gap Detection

DISCUSSION

FRAMEWORK CONTRIBUTION

The main contribution of the framework is not any single technology choice, but the operational model it provides for linking business-facing review requirements from rawdata perspective to executable, auditable validation logic from SDTM perspective. In many clinical programming environments, the principal bottleneck is not execution itself, but the inconsistent process by which review checks are interpreted, authored, and maintained. By preserving the connection between specification, programming algorithm, code artifact, and execution result, the framework improves both productivity and implementation confidence. Its practical value therefore lies not only in an integrated operational model, but also in exploring broader possibilities and application scenarios for realizing the value of SDTM datasets, rather than in a collection of isolated technical features.

RULE DEVELOPMENT AND REUSE

Because row-level review specifications can move directly into programming algorithm drafting, candidate implementation generation, and execution, the transition from identifying a new review requirement to producing a runnable check becomes shorter and more consistent. When several new checks must be introduced within the same study, this reduces not only manual effort but also the friction associated with repeated movement across narrative instructions, temporary algorithm, and disconnected scripts. Reuse of validated scripts and development of new custom implementations are managed within the same operating model, eliminating the need for separate workflows for standard and study-specific checks. Retaining generated artifacts, programming algorithms, code differences, and execution traces together also makes it easier to preserve the relationship between specification intent and final implementation, thereby strengthening reuse, collaboration, and long-term maintenance. Effective cross-study reuse, however, depends on a prerequisite that extends beyond the framework itself: sufficient consistency in CRF design and CRF-to-SDTM variable mapping across studies. When CRF structures differ substantially between studies—due to varying form designs, field naming conventions, or collection granularity—the SDTM variable conditions underlying a validated rule may not translate reliably into a new study context, limiting the practical scope of direct reuse. In this sense, CRF standardization and mapping harmonization are not merely upstream data management concerns; they are enabling conditions for the rule reuse that DQO is designed to support.

AI-ASSISTED AUTHORING

Within this framework, AI-assisted authoring should be understood primarily as a drafting accelerator. Its role is to reduce repetitive authoring effort and make implementation development more scalable, particularly when similar review checks recur across studies. However, durable value lies in the framework's traceable structure rather than in automated text generation alone. Existing scripts become reusable assets over time, and newly created custom implementations become easier to reassess, refine, and carry forward because the path from rule definition to implementation and review remains interpretable and reviewable. In practice, the effectiveness of AI-assisted authoring depends heavily on how reference materials are prepared and supplied. Several implementation skills can improve both accuracy and efficiency. First, global CRF standards, CRF-to-SDTM mapping specifications, and representative dummy datasets should be maintained as reusable reference assets so that the model can better understand how edit check wording maps to SDTM-level variables, records, and conditions. Second, these materials should be structured into modular, searchable units, such as form-level mapping summaries, domain-specific variable dictionaries, controlled terminology references, and example input-output cases, so that only the relevant subset is loaded for a given rule instead of sending the entire reference package at once. This reduces unnecessary context consumption and helps keep the model focused on the current rule logic. Third, prompts should require the model to explicitly separate business interpretation, SDTM variable mapping, executable condition logic, and unresolved assumptions. This makes it easier for programmers and reviewers to identify whether an error comes from source interpretation, mapping translation, or implementation detail. Fourth, few-shot examples from previously

reviewed checks can be used as controlled patterns for new rules, especially when they share similar domains, visit logic, date comparisons, or cross-domain dependencies. Finally, generated outputs should be handled through an iterative review loop: the Programming Algorithm, candidate code, test results on dummy data, and observed differences from expected output should all be fed back into the refinement process. These practices treat AI not simply as a text generator, but as a context-aware drafting assistant embedded in a governed rule-engineering workflow.

OPERATIONAL LIMITATIONS AND GOVERNANCE

The framework also has clear operational boundary conditions. It cannot compensate for an inadequately defined source specification. If a review instruction is incomplete, critical conditions are missing, or the business wording is ambiguous, neither manual programming nor LLM-assisted drafting can reliably produce sound logic without further clarification. AI should therefore be treated as an aid to drafting, not a substitute for judgment. No matter how quickly Programming Algorithm, candidate code, or difference summaries are generated, these materials still require human review, particularly in regulated or higher-risk settings. What merits trust is not generation alone, but the review, revision, and governance process that follows. In this sense, the framework is best understood as a controlled extension of existing validation practice through reusable script assets, custom Python implementations, and AI-assisted authoring workflows. More broadly, automation maturity should not be defined only by the number of checks that can be executed automatically, but also by how reliably teams can translate review intent into implemented logic and maintain that logic over time. The framework's emphasis on reuse and gradual implementation does not imply that every rule will transfer across unchanged studies; rather, reuse should be understood as a stronger starting point for adaptation than as a guarantee of direct portability. If the framework is extended toward production-scale use, governance becomes especially important at the transitions from specification to programming algorithm, from programming algorithm to executable logic, and from executable logic to a released rule library. A framework that strengthens these transitions while acknowledging their governance requirements can deliver meaningful value even before a large library is fully established.

CONCLUSION

This paper presented a Python-based, Streamlit-based and AI-assisted hybrid framework for operationalizing source data quality review checks based on SDTM Plus data. The framework connects specification intake, programming algorithm drafting, implementation generation, execution, reporting, and traceability within one operating model, and supports both standardized checks and study-specific complex rules by combining reuse of validated scripts with new procedural Python implementations. A rule library covering approximately 80 checks across most SDTM domains has been built and continues to grow, demonstrating the framework's scalability within an active clinical development program. The main contribution of this work is not a single technology, but a more stable rule-engineering operating model. As clinical data review grows more complex, practical hybrid frameworks of this kind can help organizations move from experience-driven review practice toward execution models that are more maintainable and reproducible. Future development directions include expanding the rule library to cover additional edge cases and domain-specific scenarios and exploring tighter integration between the DQO rule library and upstream data management planning activities.

RECOMMENDED READING

Python Software Foundation. Python Documentation. Available at: https://docs.python.org/3/. Accessed May 25, 2026.

Streamlit. Streamlit Documentation. Snowflake Inc. Available at: https://docs.streamlit.io/. Accessed May 24, 2026.

CONTACT INFORMATION

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

Nana Yang

Huan Song

nana.yang@beonemed.com
huan.song@beonemed.com
Zhouhao Pan
Chunpeng Zhao
zhouhao.pan@beonemed.com
chunpeng.zhao@beonemed.com