PharmaSUG China 2026 – Paper AP-163

Evaluation Strategy for TLF AI-Generated Results

Han Wang, Zongfeng Lei and Patrick Zhang, AstraZeneca

Abstract

Context. Generative AI is now practical enough to support parts of TLF programming. Our motivation is simple: produce more tables with less repeated manual effort, shorten turnaround time, and still keep quality visible to reviewers. But once AI starts drafting SAS code, the key question becomes how to confirm that the code follows the shell, uses the right study data, and is suitable for statistical programmer review. TLF programming is a good setting for this question because the work already depends on familiar clinical reporting materials—shells, study specs, and runnable SAS macros—that can be checked step by step.

Design logic. Evaluation should follow the path of the generated code. First, assess whether the system understood the shell correctly. Next, assess whether the code uses the right ADaM data, variables, filters, and treatment groups. Then assess whether the SAS program is complete, runnable, and understandable for a statistical programmer. These checks inform—not replace—normal programming review and QC. To make assessment repeatable at scale, we use evaluation harnesses: automated infrastructure that applies the same checks on benchmark cases at each release. Evaluation runs in a separate environment from study delivery: changes to generation settings do not affect TLFs already delivered for a study, and production approval stays the same.

This paper shares our overall evaluation strategy and framework. It does not describe internal scoring rules, benchmark contents, or implementation details. The aim is practical guidance for teams adopting assisted TLF programming—not a regulatory validation package or a technical specification.

Keywords: TLF generation, AI-generated SAS, evaluation, human review, statistical programming

1. Background

1.1 ATLAS project brief introduction

In clinical development, **Tables, Listings, and Figures (TLFs)** are core deliverables for study reporting and regulatory submission. Producing TLFs still requires a large volume of **SAS programming**: reading the shell, mapping **ADaM** analysis data, selecting company **display macros**, and writing runnable code that matches the planned output.

ATLAS is a platform that supports this work by turning study **shells** and **ADaM specifications** into draft SAS programs. The workflow follows steps that statistical programmers already recognize:

1. **Shell interpretation** — read table layout, blocks, and placeholders from the shell.

2. **ADaM mapping** — link rows and columns to the correct datasets, variables, treatment groups, and population flags.
3. **Macro selection and code assembly** — prefer standard display macros where possible; for more complex shells, apply predefined coding strategies (**S1–S6**, from direct macro call to custom assembly).
4. **Statistical programmer review** — draft code is presented for review; statistical programmers can accept, adjust through dialogue (**Copilot**), or rerun selected steps.

Figure 1 shows the end-to-end flow from shell to draft SAS code.

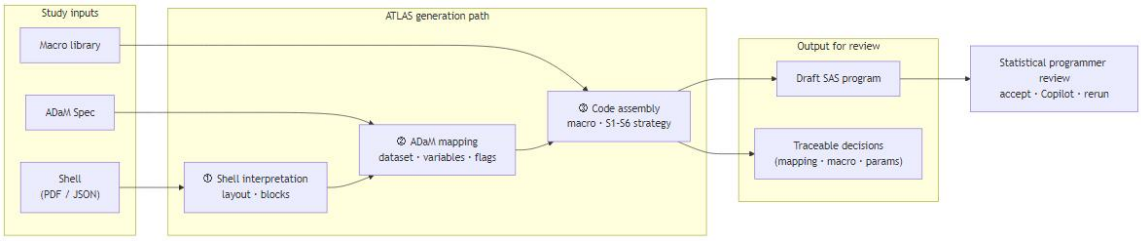

Figure 1

ATLAS is intentionally separated from routine study delivery. Changes to generation settings do not alter TLFs already delivered for a study. Final approval still follows the same programming and QC path the team uses today. ATLAS is meant to reduce repetitive drafting work and shorten turnaround—not to bypass established review.

1.2 Why evaluation are necessary

Once draft SAS code comes from a system rather than only from manual programming, teams need a clear way to answer familiar questions—and to answer them consistently as the platform evolves:

Question	What statistical programmers already check
Shell	Did the system read layout, blocks, and footnotes correctly?
ADaM	Did it use the right dataset, variables, population flags, and treatment groups?
Code	Is the SAS program complete, readable, and consistent with macro practice?
Run	Does the program run cleanly and produce output that matches expectation?

These are the same questions a lead statistical programmer or QC reviewer would ask for a single deliverable. The difference is purpose and scale. Study QC remains the path for approving a table for delivery. Evaluation is how the ATLAS team measures agent and platform quality across table types, studies, and releases—so that draft code arriving for review is worth a

statistical programmer's time, and so that release decisions rest on evidence rather than spot checks alone.

What is needed is harness-based evaluation: standardized infrastructure that runs benchmark cases, collects outputs, scores results, and supports regression. It does not invent new clinical standards; it applies existing programming expectations in a repeatable way.

For ATLAS, evaluation serves three linked goals. The highest priority is agent reliability: whether the TLF generation agent consistently produces code that is correct, runnable, and trustworthy enough for statistical programmer review. The second goal is to track agent performance over time so the team can improve before each release. The third goal is vendor delivery quality: confirming that the ATLAS platform—including UI workflows and integrations—meets agreed expectations alongside the agent itself.

Figure 2 shows how validation follows the same path as code generation.

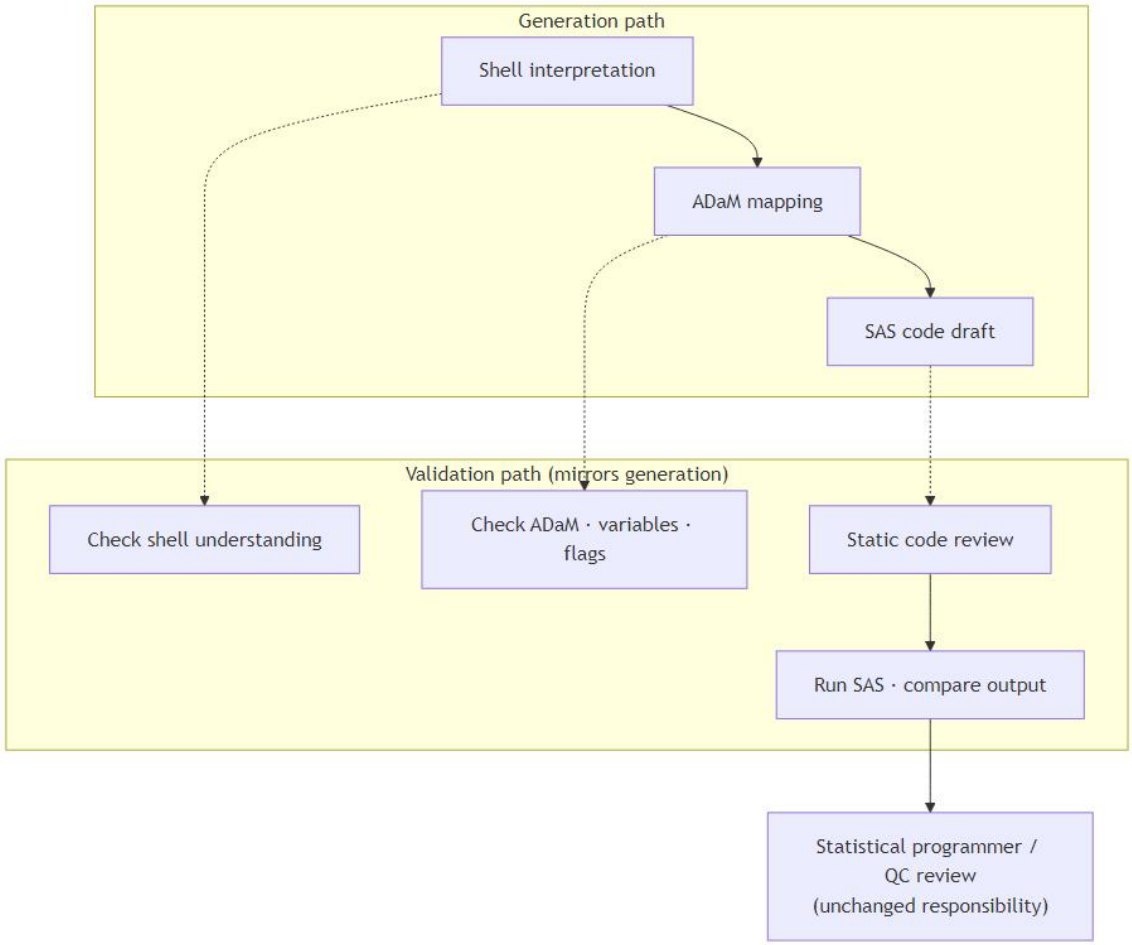

Figure 2

2. How we validate

Our approach has two layers: **system-level evaluation** (benchmark + scored framework) and **code-level validation** (static review, then runtime execution).

Figure 3 gives the overall validation architecture.

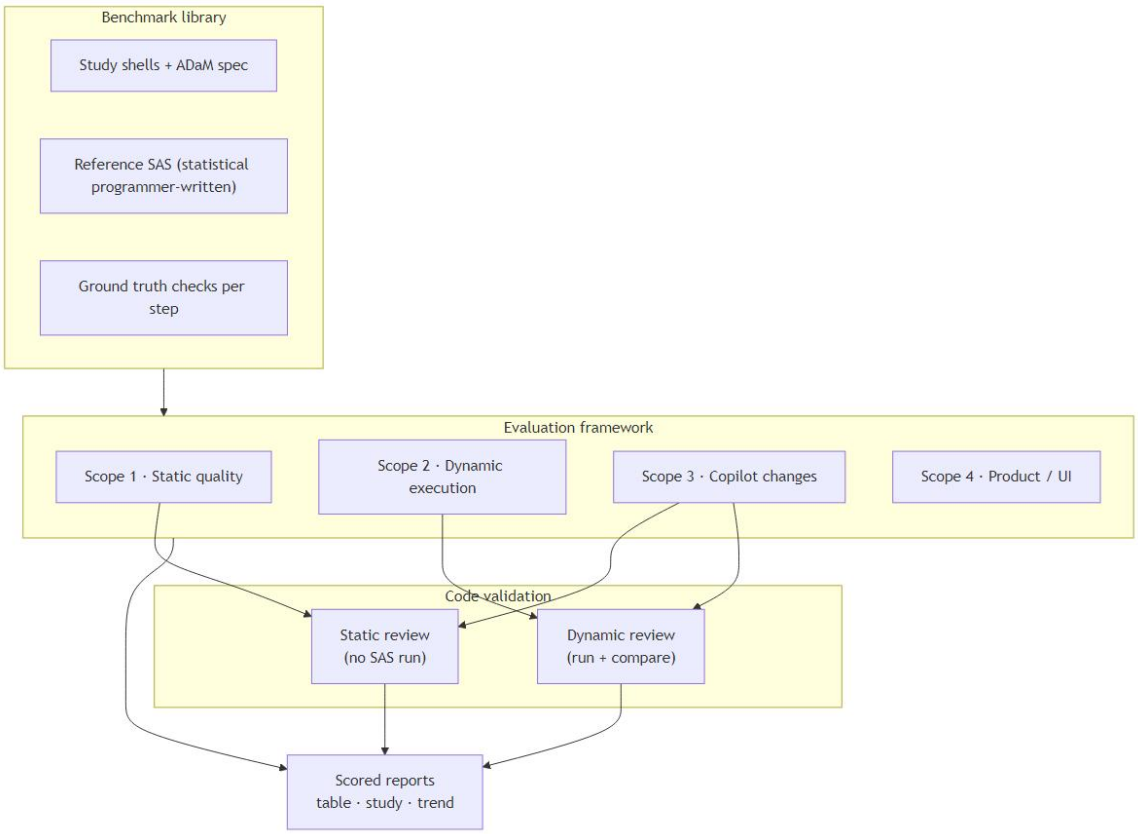

Figure 3

2.a Static code evaluation

Static evaluation examines generation outputs without executing SAS. The focus is whether the system understood the shell, mapped ADaM correctly, and assembled code that a statistical programmer would recognize as a reasonable draft. This mirrors the first half of normal programming practice: read the program before you submit the job—but here the purpose is scored, repeatable measurement across a benchmark, not study-level QC sign-off.

We group evaluation into four scopes. Scopes 1 and 2 are the current priority for code quality; Scopes 3 and 4 extend coverage to post-draft edits and the user interface.

Scope	**What we assess**
S1 — Static quality	Generation outputs without running SAS
S2 — Dynamic execution	Run generated SAS on controlled infrastructure
S3 — Copilot interaction	Multi-turn code changes after first draft

S4 — Product / front-end	Key paths in the ATLAS UI

Static checks cover the same dimensions statistical programmers review before submission: shell alignment, ADaM mapping, macro choice, and overall program quality. Scoring combines rule-based checks (where there is a clear right or wrong) with expert-calibrated review where judgment is needed. Results are summarized at table level and rolled up for study- and release-level decisions. We do not publish internal severity rules, weights, or scoring formulas.

2.b Benchmark foundation

A benchmark is a curated set of representative tables drawn from study materials—shell, ADaM spec, and reference SAS code accepted under normal programming standards. The benchmark is the foundation for both static checks and dynamic output comparison. Each table links shell requirements to an accepted reference program; from that we derive expected outcomes used in evaluation. The benchmark spans multiple therapeutic areas and complexity levels and is maintained as a living asset as new table types come into scope.

2.c Dynamic runtime evaluation

Dynamic evaluation is the closest analogue to running the program in QC—generated SAS is executed in a controlled environment, and log and output are examined. Static assessment filters draft quality; the runtime harness confirms the program actually works and produces expected results under controlled conditions.

We use an A vs. B design: generated code (A) and reference code from the benchmark (B) run in the same environment. The runtime harness handles submission, log review, and output collection. Runnable is a hard gate—a program that fails to run cannot pass regardless of static score. Output from A is compared to output from B using the same principles statistical programmers apply when reviewing QC results. Reference output is produced by running reference code in the same environment, so the comparison baseline stays aligned with accepted programming practice. Figure 3 summarizes the A vs B design and the main runtime signals.

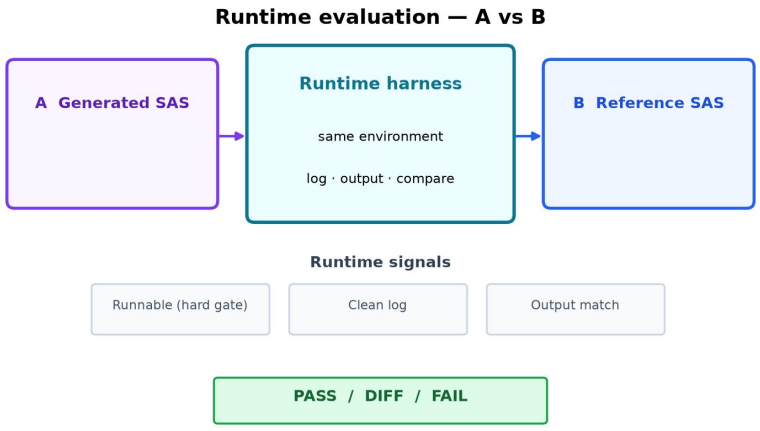

Figure 3

2.d Future direction

The static and runtime harnesses are being built as standalone capabilities first. The next step is to orchestrate them—along with Copilot and front-end checks—into a unified evaluation workflow triggered at release or on schedule. That workflow will select benchmark samples, invoke the harnesses, aggregate results, and support Go / No-Go release decisions. Implementation details, tooling, and timelines remain internal.

3. Discussion and conclusion

How this fits statistical programmer workflow

Our evaluation strategy is designed to support, not replace, normal programming review and QC:

5. Benchmark expectations encode what statistical programmers already use—shell, ADaM spec, macros, reference code.
6. Static evaluation filters obvious mismatches before a statistical programmer opens the job.
7. Runtime evaluation confirms runnability and numerical output under controlled conditions before formal study QC.
8. Post-draft interaction checks (planned) will confirm that code changes through dialogue remain safe and targeted.

ATLAS output should arrive as a reviewable draft. Evaluation makes that expectation measurable at scale across releases.

Sharing approach while protecting proprietary detail

Assisted TLF generation and harness-based evaluation are still emerging capabilities in the industry. We share the evaluation logic and framework structure so other teams can adapt the same principles—follow the code path, use benchmarks, separate static from runtime checks, keep statistical programmer review central. We intentionally omit internal benchmark studies, scoring rubrics, harness implementation, agent architecture, and vendor integration details. Those elements represent proprietary investment and are not required for readers to understand the strategy. Teams building similar capabilities should expect to develop their own benchmarks and scoring criteria aligned to local macro libraries, QC standards, and regulatory context.

Conclusion

TLF programming is a natural setting for assisted code generation because the work is already structured: shells, ADaM specs, company macros, and QC habits give clear checkpoints. ATLAS addresses the drafting bottleneck; evaluation addresses the trust bottleneck—how to know, release after release, that draft quality is holding or improving.

Our strategy is straightforward: build a representative benchmark from materials statistical programmers trust; apply static evaluation before any SAS job is submitted; confirm with

runtime execution and output comparison; and roll results into a repeatable release workflow. For statistical programming teams, the practical message is: treat generated SAS as you would any junior draft—verify against shell and spec, run the program, compare output—and invest in benchmark-based evaluation so first-pass review is informed and consistent.

4. Thanks

We thank our leader team for the support to ATLAS and statistical programming colleagues who contributed reference code, shells, and ADaM specifications to benchmark studies (Lanlan Zhao, Zhian He and Ouying Zhao); and the ATLAS development and evaluation teams building the platform and validation tooling described in this paper (Roy Wen); and OGEM team for the basic macro ATLAS use.