PharmaSUG China 2026 - Paper CC-187

# A Hybrid Framework for SAS Code Quality Review: Combining Deterministic Static Rules with Large Language Model Semantic Analysis

Meng Zhang, Clinical Trial Service Co., Ltd.

## ABSTRACT

Clinical-trial SAS programs for SDTM, ADaM, and TLF delivery are usually reviewed manually against Good Programming Practice (GPP) guidance, study specifications, and company SOPs. Manual review is effective, but it is difficult to apply consistently across every program, every reviewer, and every delivery cycle. This paper presents a tool (hereafter referred to as SAS Code Reviewer), a locally deployable review framework that combines deterministic static checks with an optional semantic review performed by a Large Language Model (LLM). The system first runs a Python rule engine of SAS-specific checks covering syntax, naming, comments, macro hygiene, maintainability, safety, hardcode checks, and selected logic hazards. The static findings are then passed into a structured prompt so that the selected LLM can confirm, refine, and supplement the rule-based findings rather than reasoning from scratch. The tool supports both LLM API calls and locally deployed models, and users can switch between providers at runtime without code changes. The architecture separates a Vue / Monaco front end, a FastAPI back end, and a unified provider adapter; deterministic and LLM findings are merged into a single, deduplicated review report. The same set of review rules is also packaged as a Skill, so that it can be invoked directly inside compatible AI assistants.

## INTRODUCTION

Clinical-trial programming teams review SAS programs for more than syntax. Reviewers also judge readability, macro usage, step boundaries, naming, comments, and common logic risks. These checks are part of Good Programming Practice, but they are often applied through peer review and project-specific conventions. Because reviewers may emphasize different aspects of a program, the same code can receive different review comments across reviewers or delivery cycles.

Data validation and source-code review answer different questions. A program can create an acceptable data set while still containing fragile macro references, unclear derivations, unsafe file operations, or maintainability problems. The question addressed in this paper is how to make code-level review more systematic without replacing professional programming judgment.

SAS Code Reviewer was built around a hybrid principle. Static rules are used for deterministic problems, such as missing RUN statements, character assignment before explicit LENGTH or ATTRIB statements, indentation, and macro hygiene. The LLM is then used for a lighter semantic review: it checks simple logic concerns, interprets the deterministic findings in context, and proposes optimization suggestions based on the issues already identified.

The design is informed by prior PharmaSUG discussions of Good Programming Practice. Chowdhury, Foxwell, and Song (2015) summarized essential GPP principles, Dalton (2017) discussed programming practice at different levels of review, and Agnihotri, Pradhan, and Brown (2022) collected SAS standard-programming tips for day-to-day programming work. This paper uses that literature as a review background and focuses on an implementation route: a local deterministic rule engine with an optional LLM semantic review layer.

## DESIGN OBJECTIVES

- Make common SAS review rules repeatable across reviewers and delivery cycles.
- Run deterministic review locally before any model call is made.
- Allow teams to choose between API-based models and locally deployed models at runtime.

- Preserve the same review logic in two interfaces: a desktop web application and an assistant Skill.

These objectives led to a layered architecture. The static review, prompt construction, provider calling, and report rendering are separated. As a result, the application can run in a local-only precheck mode, or it can add an API or locally deployed LLM without changing the rule engine.

## SYSTEM ARCHITECTURE

The application has three major layers. The front end is a Vue 3 and TypeScript application using the Monaco editor for SAS source display. It provides single-program review, rewrite support, batch review, provider selection, issue filtering, and history navigation. The back end is a FastAPI service that owns rule execution, prompt construction, response parsing, configuration, and persisted review history. The provider layer exposes a common interface for API-based models and local model deployments. In current use, the API path can cover Anthropic, OpenAI, Gemini, GLM, DeepSeek, Qwen, Kimi, and other commonly used LLM services.

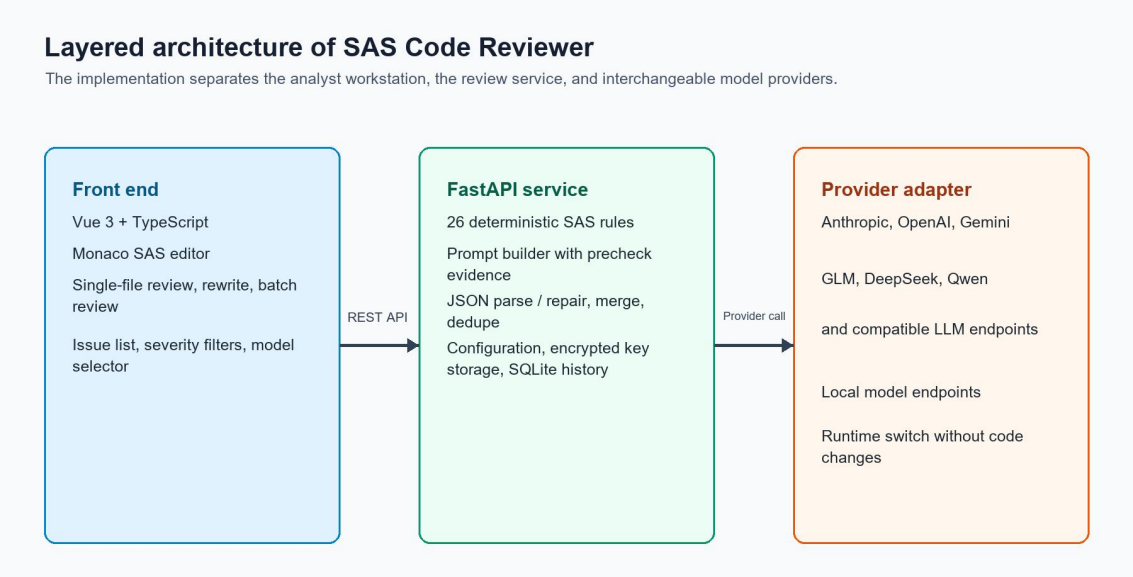


**Figure 1. Layered architecture of SAS Code Reviewer.**

Figure 1 summarizes this separation: the user interface, deterministic review service, prompt builder, and provider adapter are independent layers connected through explicit data contracts.
This separation keeps responsibilities clear. Front-end changes do not alter review rules, and provider changes do not alter deterministic findings. Static review can be run without an LLM, which gives teams a baseline mode when they only need rule-based feedback.

## REVIEW WORKFLOW

The review pipeline begins by preparing aligned views of the SAS program. The original text is retained for display; comment-stripped and string-masked views support scanning without false findings from comments or quoted keywords; and an offset-to-line index maps findings to physical source lines. The static rule engine then returns normalized Issue records and completion status before any model call. These findings can be reported directly or included in the semantic-review prompt.

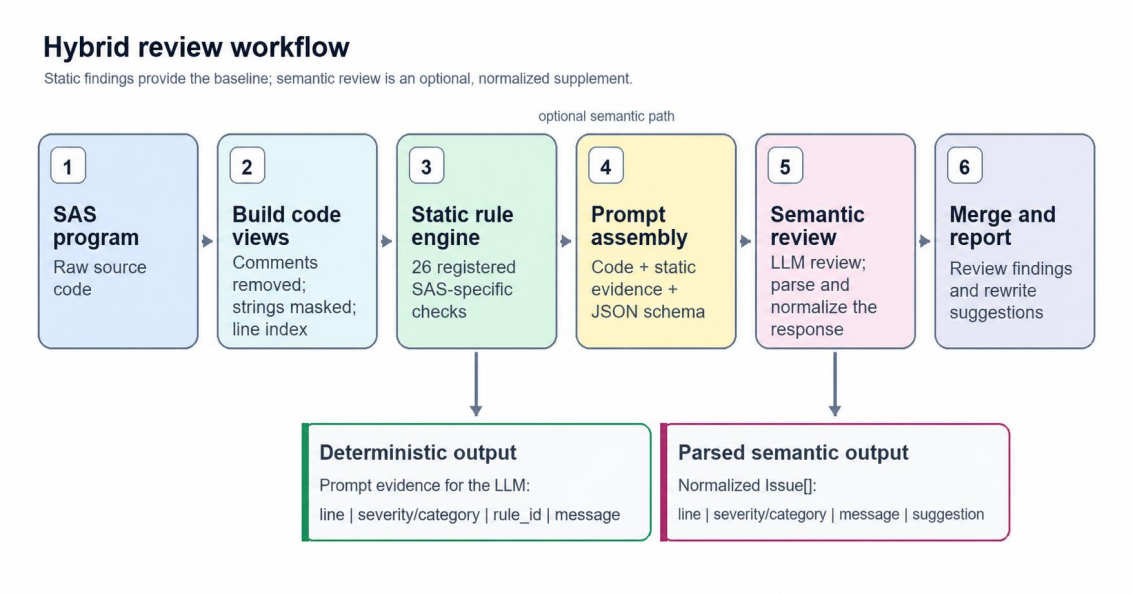


**Figure 2. Hybrid review workflow.**

Figure 2 distinguishes the deterministic and optional semantic paths. Deterministic findings can be reported directly or used as formatted prompt evidence. When semantic review is enabled, the model response is extracted, repaired where possible, and normalized to the common Issue schema before merge. The user-facing outputs are the review findings and, when optimization is requested, the rewrite suggestions.

The review result is not a simple concatenation. Deterministic issues are retained first, followed by parsed semantic issues. Duplicates are removed using the physical line number and the lower-cased first 40 characters of the message, so a repeated model finding does not displace its deterministic counterpart. Rewrite suggestions are generated in a separate step from the retained issue list and do not replace the merged review findings. Unparseable output becomes a diagnostic Issue rather than being silently discarded. The programmer or reviewer retains authority over relevance and acceptance.

The application interface reflects the same workflow. Figure 3 shows the merged and deduplicated review findings in the source-code view, while Figure 4 shows the optimization view used to compare the original program with the generated rewrite suggestions.

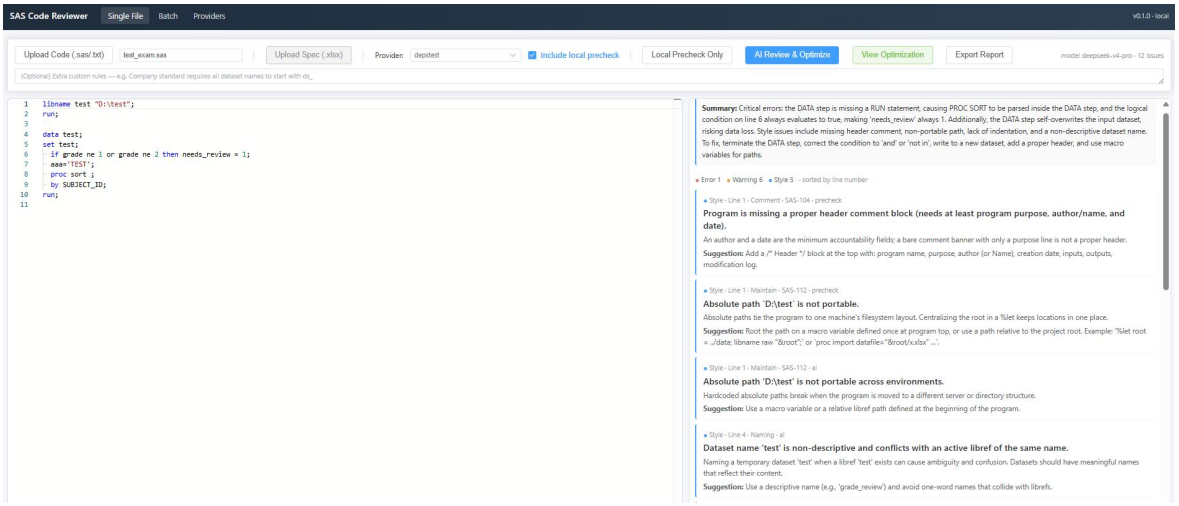

**Figure 3. Application review result with source code and merged findings.**

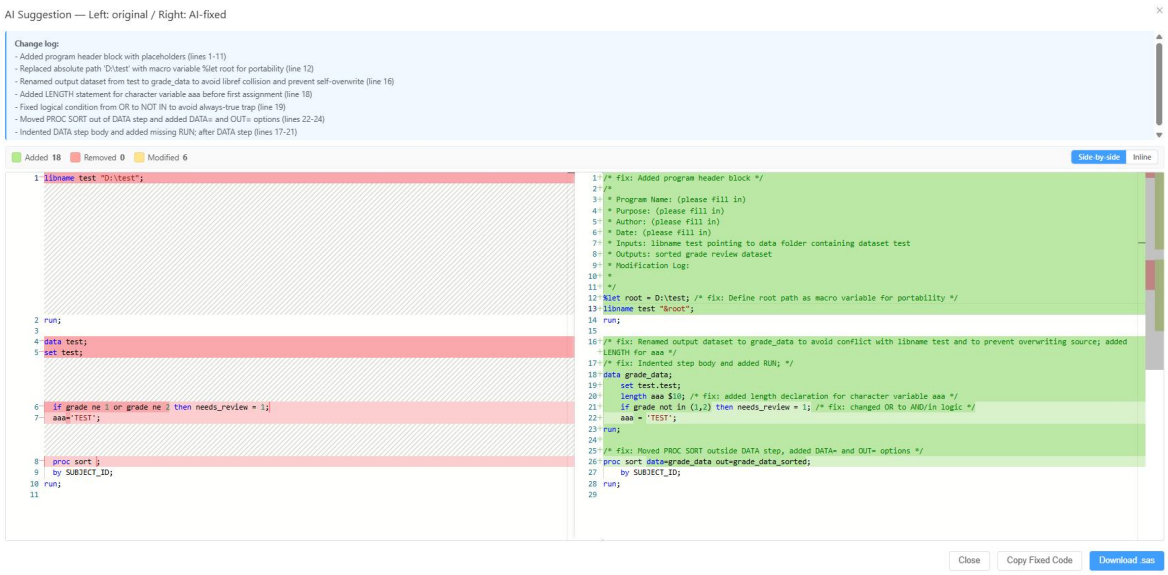


**Figure 4. Application AI optimization view showing original and revised code.**

## DETERMINISTIC RULE ENGINE

The deterministic rule engine currently contains 26 active SAS-specific checks. The active rules cover style and layout, syntax risk, macro hygiene, safety, hardcode checks, and selected logic hazards. Each finding carries a line number, severity, category, message, explanation, and suggested remediation.

## STATIC RULE CONSTRUCTION

The rule engine is built as a registry of small Python functions. Each function receives a shared RuleContext and returns zero or more Issue objects. RuleContext is prepared once for a program and contains the original source lines, a comment-stripped view, a string-masked view, a line index, macro definitions and references, and step boundary information. This prevents each rule from re-parsing the program independently and reduces false positives from comments, quoted text, or inline data regions. A rule is intentionally narrow: it identifies one review concern, assigns a stable rule number, and writes a short suggestion. The registry controls execution order. If one rule fails unexpectedly, the runner records the error and continues with the remaining rules, so a single rule defect does not incorrectly produce a clean review.

| Rule family | Examples | Review purpose |
|---|---|---|
| Style and layout (SAS-101 to SAS-113) | Long identifiers, missing header, line length, mixed indentation, unindented step bodies | Improve readability, reviewer orientation, and consistent source formatting |
| Syntax and step control (SAS-201, SAS-202, SAS-209) | Missing RUN or QUIT, PROC SQL closed with RUN, character assignment before LENGTH | Detect code patterns likely to create log warnings, truncation, or ambiguous step termination |
| Macro hygiene (SAS-301 to SAS-304) | Unresolved macro token, missing local macro scope, positional macro parameters, nested macro definitions | Reduce hidden side effects and make reusable code easier to validate |
| Safety and governance (SAS- | Self-overwrite, PROC FORMAT | Protect study assets from unsafe |

| Rule family | Examples | Review purpose |
|---|---|---|
| 203, SAS-206, SAS-211, SAS-501) | without OTHER, ERROR: literal in PUT, PROC DATASETS KILL | operations and log patterns that require reviewer attention |
| Hardcode check (SAS-601) | Hardcoded literal assignment candidate when metadata is available | Highlight values that may need derivation review or reviewer confirmation |
| Logic hazards (SAS-701 to SAS-702) | MERGE without BY, missing value silently recoded to a constant | Identify SAS behaviors that can change analysis populations or derived values |

**Table 1. Deterministic Rule Families.**

The following simplified example shows the construction pattern used by one static rule. SAS-209 checks a DATA step in which a character variable is first assigned a string literal without a preceding LENGTH or ATTRIB statement. The production implementation prepares comments, quoted strings, and step boundaries before this rule is called. The simplified rule structure is:

```
def rule_char_var_no_length(ctx):
    for step in ctx.data_steps:
        declared = collect_length_and_attrib_names(step)
        for assignment in find_char_literal_assignments(step):
            if assignment.name in declared:
                continue
            yield Issue(
                rule_id='SAS-209',
                line=assignment.line,
                severity='Warning',
                category='Syntax',
                message='Missing LENGTH/ATTRIB before assignment.'
            )
ALL_RULES = [rule_char_var_no_length, ...]
```

For a program such as `data out; trt = 'Placebo'; run;`, the context builder first identifies the DATA step and collects variables declared by LENGTH or ATTRIB. The rule then finds the first character literal assignment and emits a SAS-209 finding if the target variable has no prior length declaration in the same step.

## SEMANTIC REVIEW WITH LARGE LANGUAGE MODELS

The semantic review is intentionally narrower than a general-purpose code critique. The model is asked to inspect simple logic concerns, clarify deterministic findings, and suggest code-level improvements. It is not asked to replace the programmer or to certify the program as submission ready. The review output is treated as reference material for a human reviewer.

This approach has two practical effects. First, the model sees the static findings rather than starting from an empty prompt. Second, the model can add contextual comments that deterministic patterns cannot reliably capture, such as a suspicious IF condition, an unnecessary repeated sort, or a macro call whose parameters make the program difficult to read. The report keeps the source of each finding visible, so reviewers can distinguish rule-based evidence from model-generated advice.

## PROMPT CONSTRUCTION

In implementation, the prompt is assembled in a fixed sequence so that the model receives the review task and scope, review categories, deterministic findings, source code, and output schema in a predictable order. A simplified prompt assembly is:

```
review_prompt = [
    review_task_and_scope,
    category_definitions,
    deterministic_findings_json,
    sas_source_code,
    required_output_schema,
]
semantic_findings =
parse_json_with_repair(provider.review(review_prompt))
final_report = merge_and_deduplicate(static_findings, semantic_findings)
```

The parser accepts only schema-compatible findings for the final report. If the model returns extra prose, the response is repaired or rejected before merge and deduplication.

## PROVIDER ADAPTER AND LOCAL MODEL SUPPORT

The provider adapter isolates provider-specific HTTP details from review logic. A configuration record supplies the provider name, model, base URL when applicable, timeout, token budget, and credential reference. OpenAI-compatible providers are handled through the same adapter path, so an API model and a locally deployed model can be selected without changing the rule engine or prompt assembly.
The following simplified adapter example shows the minimum configuration surface needed to switch between API-based and local model endpoints:

```
class LLMProvider:
    async def review(self, prompt: str, config: ProviderConfig) -> str:
        raise NotImplementedError
config = ProviderConfig(
    provider='openai_compat',
    model='local-sas-review-model',
    base_url='xxxx',
    timeout_seconds=120,
    max_tokens=4096,
)
```

For teams that prefer local deployment, the same adapter can call an internal model endpoint. For teams that prefer API access, the provider can be changed through configuration rather than code.

## SKILL PACKAGING

The same review method is also packaged as a Skill. In that form, the entry point is SKILL.md. It defines when the Skill should be invoked, how to distinguish single-file and batch review, how to handle uploaded zip archives safely, and the required execution order. The deterministic engine is provided as scripts/static_check.py, and the report renderer is scripts/render_report.py. The Skill therefore gives an assistant a reproducible workflow rather than a loose instruction to inspect SAS code.
The Skill follows a standard assistant-skill structure. The root SKILL.md file serves as the instruction contract. The scripts directory contains the executable checker, report renderer, encoding helper, and indentation helper. The docs and examples directories describe the rule set and provide small SAS

programs for reproducible testing. The tests directory contains regression tests and SAS fixtures used to keep rule behavior stable.

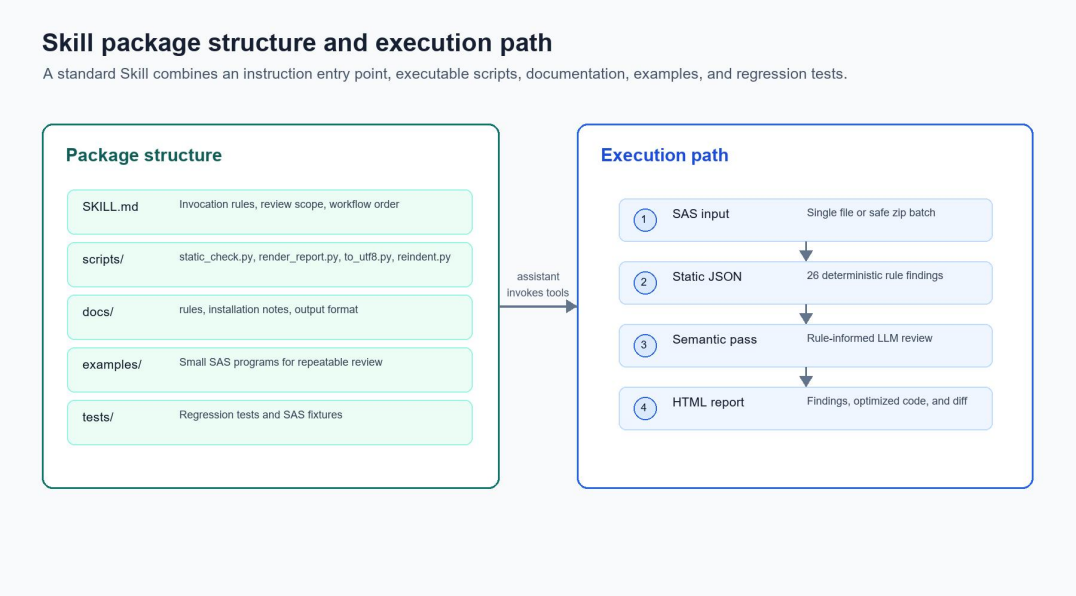


**Figure 5. Skill package structure and execution path.**

Figure 5 combines this package layout with the execution path: a compatible assistant invokes the Skill, runs the deterministic checker to create JSON findings, applies semantic review when requested, and renders a self-contained HTML report.

In the Skill implementation, each input SAS file receives its own self-contained HTML report. The optimized version of the program is embedded in that report together with a side-by-side diff. This choice avoids presenting an uncontrolled rewritten source file as the deliverable. The reviewer can copy accepted changes after inspection.

Figures 6 and 7 show the two main Skill outputs: a structured review report and an embedded optimization example that remains tied to the review artifact.

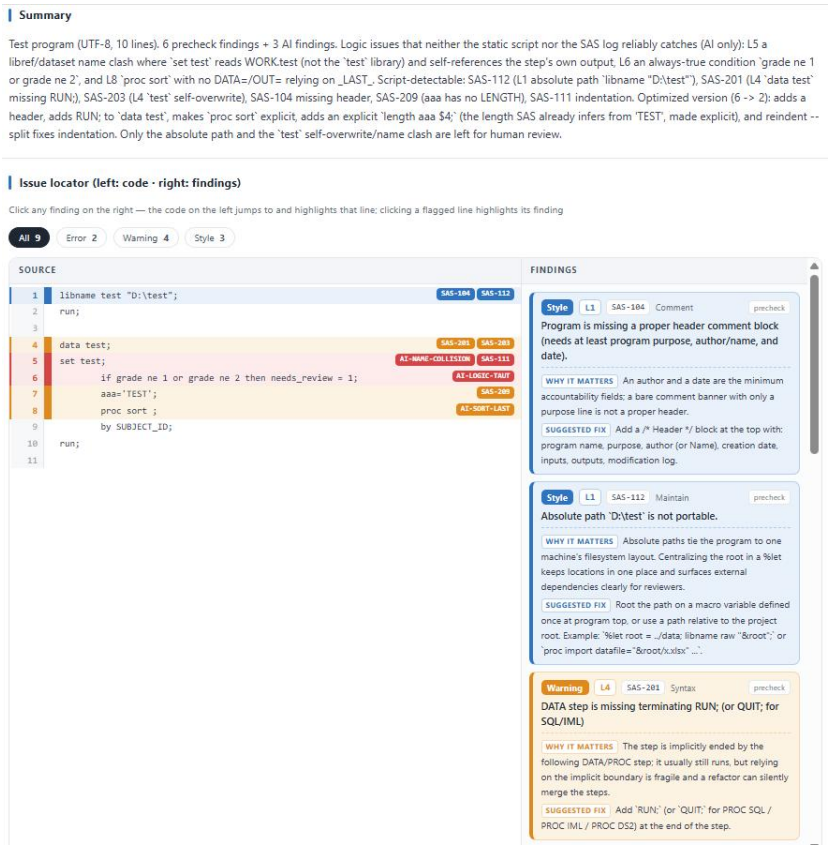


**Figure 6. Skill-generated HTML review report.**

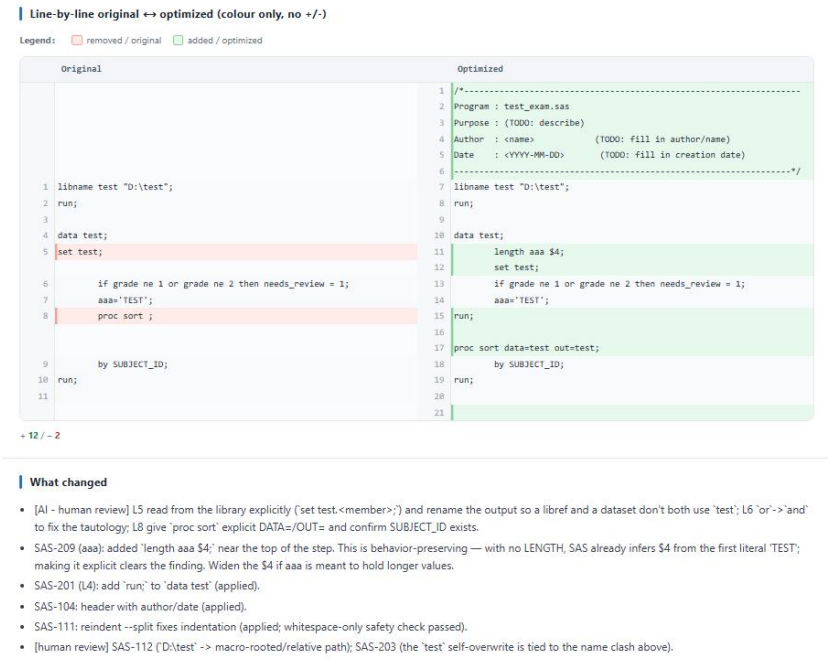


**Figure 7. Skill AI optimization example embedded in the report.**

The Skill workflow can be reproduced by two command-level steps: first generating deterministic JSON findings, and then rendering the final HTML report with the original program, curated findings, and optimized program embedded in the same file:

```
python scripts/static_check.py --file program.sas --pretty
python scripts/render_report.py \
    --original program.sas \
    --report-json curated_findings.json \
    --optimized scratch_optimized.sas \
    --out program.review.html
```

These commands are not presented as an end-user interface; they show how the Skill separates deterministic checking from final report rendering.

## DISCUSSION AND LIMITATIONS

The framework should be understood as a review accelerator rather than a final review authority. Static findings and model suggestions provide structured reference material, but the programmer or reviewer still decides whether a finding is relevant and whether a suggested rewrite is acceptable. In this sense, the tool is intended to shorten the review process, not to remove human review from the process.

During testing on an ordinary personal computer, a locally deployed quantized Qwen3.5-9B model produced reasonable comments on short SAS programs, for example programs below about 50 lines. For longer programs, review time increased substantially and the quality of semantic findings became less stable. Larger local models may improve semantic quality, but they also increase hardware cost. This trade-off is important for teams considering local deployment.

API-based review avoids local model setup, but response time may be affected by network latency and provider-side scheduling. The Skill path can run the deterministic check and render the HTML report directly in the assistant workflow. When semantic review or code rewriting is requested, different LLMs may produce different review comments, optimization suggestions, and revised code.

The current rule set focuses mainly on Good Programming Practice: layout, comments, macro hygiene, step control, simple safety checks, and selected logic patterns. It does not yet attempt to encode a broad set of CDISC conformance rules. A natural extension is to combine project specifications and selected CDISC rule knowledge with the current source-code review framework.

## CONCLUSION

SAS Code Reviewer demonstrates a practical framework for improving the consistency of clinical-trial SAS code review. The key design choice is to place deterministic static checks before LLM semantic review. Static rules provide repeatable findings for well-defined issues. The LLM provides limited semantic comments and optimization suggestions under a structured schema. Provider adapters allow the same application to run against API-based or locally deployed models, while Skill packaging makes the workflow available inside compatible assistants. The result is a configurable review aid that can be adopted incrementally by SAS programming teams.

## APPENDIX: REVIEW RULE CATALOG

Table A1 lists the active deterministic rules in the reviewed implementation. The table summarizes what each rule checks without attempting to reproduce the full source code.

| Category | Rule | Detection focus |
|---|---|---|
| Style/Layout | SAS-101 | Path literal contains spaces or non-ASCII characters. |
| Naming | SAS-102 | Identifier exceeds the 32-character SAS name limit. |
| Comment | SAS-104 | Program lacks a reviewer-oriented header block. |
| Naming/Layout | SAS-105 | SAS keyword casing is inconsistent across the file. |
| Layout | SAS-106 | Multiple executable SAS statements appear on one physical line. |
| Layout | SAS-107 | DATA/PROC steps are not visually separated. |
| Maintainability | SAS-108 | Debugging residue remains in production code. |
| Layout | SAS-109 | Indentation mixes tabs and spaces. |
| Layout | SAS-110 | Line length exceeds the configured review width. |
| Layout | SAS-111 | DATA/PROC step body is not indented. |
| Maintainability | SAS-112 | Absolute path makes the program machine-dependent. |
| Layout | SAS-113 | DO or %DO block body is not indented deeper than the opener. |
| Syntax | SAS-201 | DATA/PROC step is missing RUN or QUIT. |
| Syntax | SAS-202 | SQL-like procedures are closed with RUN instead of QUIT. |
| Safety | SAS-203 | DATA step writes to the same data set it reads from. |
| Safety | SAS-206 | PROC FORMAT VALUE block lacks an OTHER= branch. |
| Syntax | SAS-209 | Character variable is first assigned a literal without LENGTH/ATTRIB. |
| Safety | SAS-211 | PUT writes an ERROR: literal that can be promoted in the SAS log. |
| Macro | SAS-301 | Macro variable reference has no visible definition or may need a dot delimiter. |
| Macro | SAS-302 | %LET inside a macro has no explicit local/global scope. |
| Macro | SAS-303 | Macro uses positional rather than keyword parameters. |
| Macro | SAS-304 | Macro definition is nested inside another macro. |
| Safety | SAS-501 | PROC DATASETS KILL appears outside a sanctioned WORK cleanup pattern. |
| Hardcode | SAS-601 | Hardcoded literal assignment candidate identified from available metadata. |
| Logic | SAS-701 | MERGE statement has no BY statement. |
| Logic | SAS-702 | Missing numeric value is silently recoded to a constant. |

**Table A1. Active deterministic SAS review rules.**

Table A2 summarizes the semantic checks used in the AI review layer. These checks are applied as review guidance for the model and are interpreted together with the deterministic findings.

| AI review | Review focus |
|---|---|
| Syntax and step logic | Review DATA/PROC boundaries, IF/WHERE conditions, and simple branch logic that may be syntactically valid but operationally risky. |
| Naming and layout | Assess whether names, indentation, line breaks, and code organization are reviewer-friendly. |
| Comment and header | Check whether the program header and local comments explain |

| AI review | Review focus |
|---|---|
| adequacy | purpose, inputs, outputs, and non-obvious derivations. |
| Macro semantics | Review macro parameters, scope, visible side effects, and whether a macro call is understandable to a reviewer. |
| Performance | Identify avoidable repeated sorts, unnecessary passes through data sets, and inefficient step structure. |
| Maintainability | Look for duplicated logic, excessive hardcoding, overly complex steps, and opportunities to simplify without changing intent. |
| Safety | Review overwrite risk, destructive operations, production path exposure, and changes that should remain subject to human confirmation. |
| Optimization suggestion | Suggest a behavior-preserving rewrite or reviewer action only after considering the deterministic findings. |

**Table A2. AI semantic review checklist.**

## REFERENCES


Shafi Chowdhury, Mark Foxwell, and Cindy Song. 2015. "Essential Guide to Good Programming Practice." PharmaSUG 2015 - Paper TT10.

Maria Dalton. 2017. "Good Programming Practices at Every Level." PharmaSUG 2017 - Paper IB02.

Navneet Agnihotri, Sumit Pradhan, and Rachel Brown. 2022. "SAS Standard Programming Practices - Handy Tips for the Savvy Programmer." PharmaSUG 2022 - Paper AP-072.

FastAPI. "FastAPI Documentation." Available at https://fastapi.tiangolo.com.

Vue.js. "Vue.js Documentation." Available at https://vuejs.org/guide/introduction.html.

Microsoft. "Monaco Editor Documentation." Available at https://microsoft.github.io/monaco-editor/docs.html.

Anthropic. "Claude Tool Use Overview." Available at https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview.


## CONTACT INFORMATION

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


Meng Zhang
Clinical Trial Service Co., Ltd.
eamon.zhang12@gmail.com
Skill package: https://github.com/eamonzhang12-007/sas-code-review