PharmaSUG 2026 – Paper AI-125

Enhancing ADaM Specification Validation and Generation of SAS Codes Using LLM through Amazon Bedrock: A Practical Framework

Pavan Kumar Tatikonda, Ryou Nakaya, and Sravan Kongara, Takeda Pharmaceutical Company

Abstract:

Accurate and well-documented CDISC ADaM specifications and efficient SAS programming are critical for producing high-quality clinical trial analysis datasets. However, the manual review and translation of specifications into SAS code is often time-consuming, inconsistent, and prone to human error particularly when specifications evolve over time or vary in structure across studies. Recent advances in Generative Artificial Intelligence (GenAI) present an opportunity to augment traditional statistical programming workflows with intelligent validation and automation.

This paper presents a practical, enterprise-ready framework that leverages large language models (LLMs), specifically Anthropic Claude models accessed via Amazon Bedrock, to automate the validation of ADaM specifications and generate corresponding SAS code. The solution is implemented in Python and designed as a phased pipeline: (1) specification validation and enhancement, and (2) SAS code generation from validated specifications at variable level and (3) SAS code polishing to consolidate redundant steps and improve program efficiency. The framework reads Excel-based specifications using pandas, invokes Claude through boto3, enforces structured outputs through JSON parsing and fallback logic, and generates SAS-style logs for traceability. Key design considerations, including prompt engineering strategies, JSON-safe parsing, error recovery, adaptive chunking to prevent LLM truncation, logging, and auditability are discussed in detail. Lessons learned from implementing and testing the framework highlight both the strengths and limitations of using GenAI in regulated clinical programming environments. By bridging traditional manual SAS-based workflows with modern GenAI capabilities, this work demonstrates how leveraging the two technologies for automation can improve efficiency, consistency, and traceability while maintaining human oversight and regulatory readiness.

1. Introduction

Clinical trial analysis datasets must be traceable, reproducible, and compliant with CDSIC ADaM standards. ADaM specifications serve as the foundation for clinical analysis dataset programming, ensuring traceability from source data (SDTM) and alignment with CDISC standards. Despite their importance, the creation and validation of ADaM specifications remain largely manual processes. Programmers often interpret the derivation logic written as free text usually in the statistical analysis plan; translate that logic into SAS code and iteratively refine

both the specification and implementation through review cycles. This workflow introduces variability, delays, and the potential for misinterpretation.

Recent advancements in Generative AI, particularly large language models, have demonstrated strong capabilities in interpreting natural language and generating programming code. However, applying these technologies in regulated environments such as clinical trials requires careful consideration of robustness, traceability, and error handling. This paper explores how GenAI can be integrated responsibly into ADaM generation workflow to assist and not to replace statistical programmers.

Within the scope of our POC to come up with an exploratory framework, we have leveraged a pilot sample ADaM spec and datasets published by CDISC. The sample has complete sets of SDTM and ADaM with xpt and JSON formatted datasets as well as define.xml files. https://github.com/cdisc-org/sdtm-adam-pilot-project/tree/master

2. Problem Statement

The traditional ADaM specification and programming workflow faces several challenges:

- **Poor or incomplete ADaM specification derivations result in inconsistent** implementations.
- **Specification formats vary** across studies complicating automation.
- **Late detection of issues** during QC or submission reviews increases rework.
- **No systematic validation** of specifications prior to programming.
- **Limited traceability** between specification text and final SAS code.

These challenges motivated the development of an automated and GenAI-assisted framework that validates specifications early and generates SAS code in a controlled and auditable manner.

3. Solution Overview

The proposed framework to enhance the ADaM specifications lifecycle, integrates Python automation with Amazon Bedrock-hosted Claude models (Claude 4 Sonnet). The framework leverages Python's well-known libraries to streamline interfacing with AWS LLMs. The solution is intentionally split into two following independent phases: (Figure 1)

Phase 1: Specification Validation

- Validates derivation logic
- Identifies ambiguities or inconsistencies
- Suggest improvements
- Produces a "validated" ADaM specifications

Phase 2: SAS Code Generation

- Generates SAS code snippets per variable using validated derivations
- Writes SAS code back to Excel file
- Compiles all the SAS code snippets into a SAS program for traceability

Phase 2b: SAS Program Optimization(Code Polish Step)

- Consolidates and re-structures raw snippets
- Removes redundancy and repeated steps
- Produces a polished SAS program for testing

This phased design supports governance by separating validation from code generation and program optimization. This separation mirrors established clinical programming practices, where specifications are reviewed and approved prior to coding.

4. Technical Architecture and Workflow

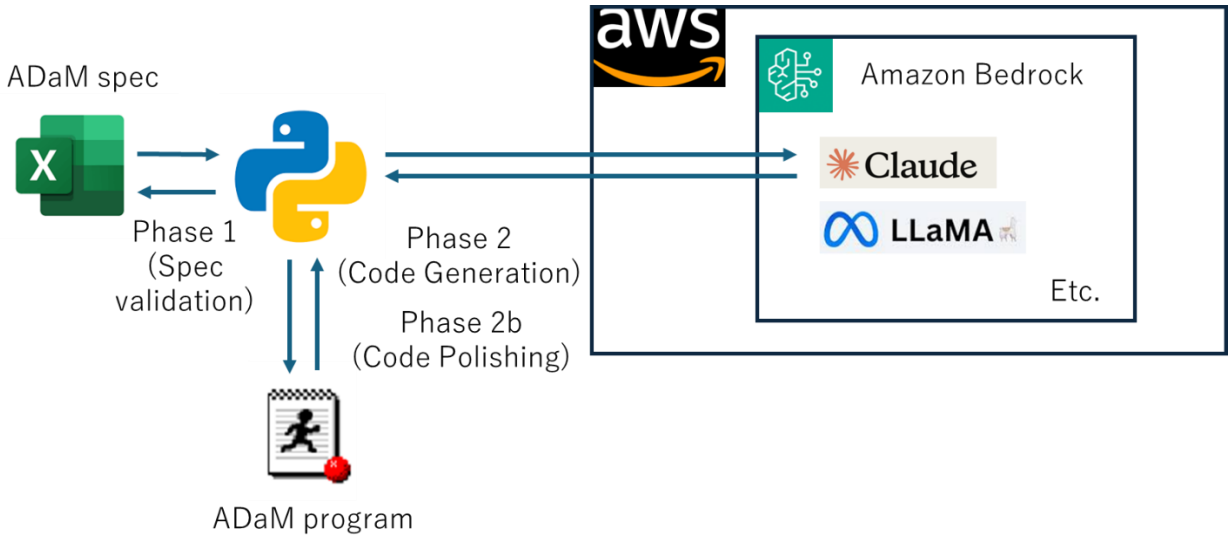

Figure 1: ADaM Automation Architecture

The framework uses standard Python modules and AWS services:

- **Python 3.10+**
- **pandas** (`read_excel`, `to_excel`)
- **boto3** (`invoke_model`)
- **json** (`loads`, `dumps`)
- **re** (regex extraction and cleanup)

- **logging** (SAS-style log files)
- **datetime** (timestamped logs)
- **time.sleep** (rate-limiting model calls)
- **os** (file and environment variable handling)

The LLM inference is executed through **Amazon Bedrock**, enabling enterprise security controls and restricted model access.

4.1 Input and output Artifacts

Input

Excel file containing variable specifications, including Variable, Derivation, Origin, Label, Data Type, Length, Format, and Codelist.

Output

- Validated specification Excel file
- SAS-enriched Excel file (includes generated SAS code)
- Raw compiled SAS program (`*_compiled_raw.sas`)
- Polished SAS program (`*_compiled_polished.sas`)
- Timestamped `.log` execution logs

5. Phase 1: Specification Validation

Phase 1 validates each variable derivation in the Excel specification.

- The script reads the sheet using: `pandas.read_excel()`
- Each row is processed using: `DataFrame.iterrows()`
- Prompts are dynamically built using a function such as: `build_validation_prompt(variable, derivation)`
- The Claude response is requested in structured JSON format to support automation. Amazon Bedrock is invoked using: `boto3.client("bedrock-runtime"), invoke_model()`
- Results are parsed using: `json.loads()`
- fallback regex extraction using `re.search()`
- Validated feedback is appended into new columns (Figure 2) and written back using: `DataFrame.to_excel()`

Derivation	Validation Feedback	Suggested Edits	Validated Derivation
Y if DCREASCD='Adverse Event'. Null otherwise	Valid		
DM.DTHFL	Valid		
WEIGHTBL / ((HEIGHTBL*100)**2)	Incorrect	The formula contains an error in unit conversion. HEIGHTBL is typically in cm, so dividing by 100 (not multiplying by 100 converts to meters for standard BMI calculation in kg/m².	WEIGHTBL / ((HEIGHTBL/100)**2)
BMIBLGR1=Normal if . < BMIBL <25. BMIBLGR1=Overweight if 25 <=BMIBL <30. BMIBLGR1=Obese if BMIBL >=30	Incomplete	Missing handling for missing/null BMIBL values and lower bound condition. The logic states '. < BMIBL <25' but doesn't specify what happens when BMIBL is missing or ≤0.	BMIBLGR1 = 'Normal' if 0 < BMIBL < 25; BMIBLGR1 = 'Overweight' if 25 <= BMIBL < 30; BMIBLGR1 = 'Obese' if BMIBL >= 30; BMIBLGR1 = missing if BMIBL is missing or <= 0
VSSTRESN when VS.VSTESTCD='HEIGHT' and VS.VISITNUM=1	Incomplete	Missing dataset reference prefix (VS.), missing handling for multiple height records at baseline visit, and missing units specification. Should specify VS.VSSTRESN and add logic to handle potential multiple height measurements.	VS.VSSTRESN when VS.VSTESTCD='HEIGHT' and VS.VISITNUM=1. If multiple records exist, use the last non-missing value.
VSSTRESN when VS.VSTESTCD='WEIGHT' and VS.VISITNUM=3	Incomplete	Missing dataset reference prefix (VS.), missing handling for multiple records per subject at baseline visit, and missing specification for missing value handling when no weight record exists at Visit 3.	VS.VSSTRESN when VS.VSTESTCD='WEIGHT' and VS.VISITNUM=3. If multiple records exist for the same subject at Visit 3, use the last non-missing value. Set to missing if no weight record exists at Visit 3.
SC.SCSTRESN where SC.SCTESTCD=YEARSEDU	Valid		

Figure 2: Validated Spec output from Model

6. Phase 2: SAS Code Generation

Phase 2 generates SAS code using validated derivations and metadata.

- The SAS prompt is constructed using `build_sas_prompt(variable, derivation, origin, label, format, codelist, datatype, length)`

The framework uses metadata-driven rules to guide SAS coding patterns for Predecessor variables, assigned variables, and derived variables.

- Claude returns SAS code in strict JSON format: {"sas_code":"<SAS CODE>"}
- A robust parser (`extract_json()`) is implemented using: `json.loads()`
- regex extraction (`re.search()`)
- fallback SAS recovery when JSON fails
- Outputs are written back to Excel (Figure 3) and compiled into a raw SAS program using Python `open()` file write operations.

This raw SAS program preserves traceability by including variable-level headers.

Validated Derivation	Sample SAS Code
	data ADSL_DTHFL; length DTHFL $ 1; label DTHFL="Subject Died?"; set sdtm.DM; DTHFL=DTHFL; keep USUBJID DTHFL; run;

WEIGHTBL / ((HEIGHTBL/100)**2)	data ADSL_BMIBL;
	length BMIBL 8;
	label BMIBL="Baseline BMI (kg/m^2)";
	set ADSL;
	if not missing(WEIGHTBL) and not missing(HEIGHTBL) and HEIGHTBL > 0 then do;
	BMIBL = WEIGHTBL / ((HEIGHTBL/100)**2);
	end;
	keep USUBJID BMIBL;
	run;

If BMIBL is not missing and 0 < BMIBL < 25 then BMIBLGR1 = 'Normal'. If BMIBL is not missing and 25 <= BMIBL < 30 then BMIBLGR1 = 'Overweight'. If BMIBL is not missing and BMIBL >= 30 then BMIBLGR1 = 'Obese'. Otherwise BMIBLGR1 = missing.	data ADSL_BMIBLGR1; length BMIBLGR1 $ 6; label BMIBLGR1="Pooled Baseline BMI Group 1"; set ADSL; if . < BMIBL < 25 then BMIBLGR1="Normal"; else if 25 <= BMIBL < 30 then BMIBLGR1="Overweight"; else if BMIBL >= 30 then BMIBLGR1="Obese"; keep USUBJID BMIBLGR1; run;

Figure 3: SAS Code written back to Specifications by Model for each variable

7. Phase 2b: SAS Program Optimization (Code Polish Step)

While Phase 2 improves traceability, row-wise SAS generation typically produces redundant patterns such as multiple DATA steps per variable, repeated PROC SORT operations, and unnecessary intermediate datasets. To improve program efficiency (Figure 4), Phase 2b performs GenAI-based refactoring on the compiled SAS program.

7.1 Chunking Strategy

Because program-level refactoring may exceed model token limits, the framework applies adaptive chunking:

- Raw SAS is split into variable blocks using header comments.
- Blocks are grouped into chunks (e.g., 10 variables per chunk).
- Each chunk is polished independently using Claude.
- Truncation is detected using a heuristic function (e.g., `looks_truncated()`).
- If truncation is detected, the chunk is retried at smaller size:
  - 10 → 5 → 2 → 1

This approach ensures complete output generation while preventing pipeline failures.

7.2 Outputs

Phase 2b produces: Both versions are retained for governance and traceability.

- Raw SAS program: `*_compiled_raw.sas`
- Polished SAS program: `*_compiled_polished.sas`

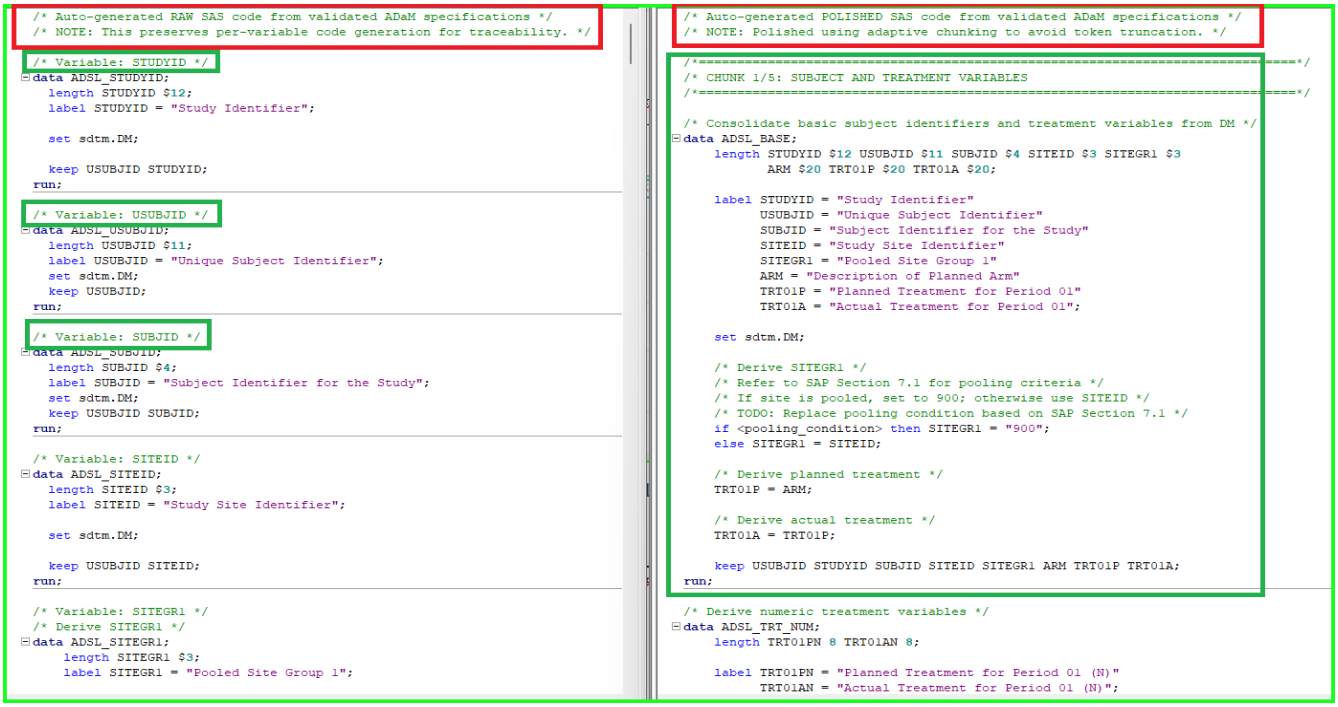

Figure 4: Comparing the SAS Code files before and after code polish

8. Logging and Auditability

A critical requirement for integrating Generative AI into regulated clinical programming workflows is ensuring traceability, reproducibility, and audit readiness. To support this requirement, the framework implements a structured logging strategy designed to mimic traditional SAS execution logs (Figure 5) while capturing additional information specific to GenAI-driven processing.

The framework generates timestamped `.log` files using Python's built-in `logging` module. Logging configuration is initialized using:

- `logging.basicConfig()`
- `logging.FileHandler(..., encoding="utf-8")`
- `logging.StreamHandler()`

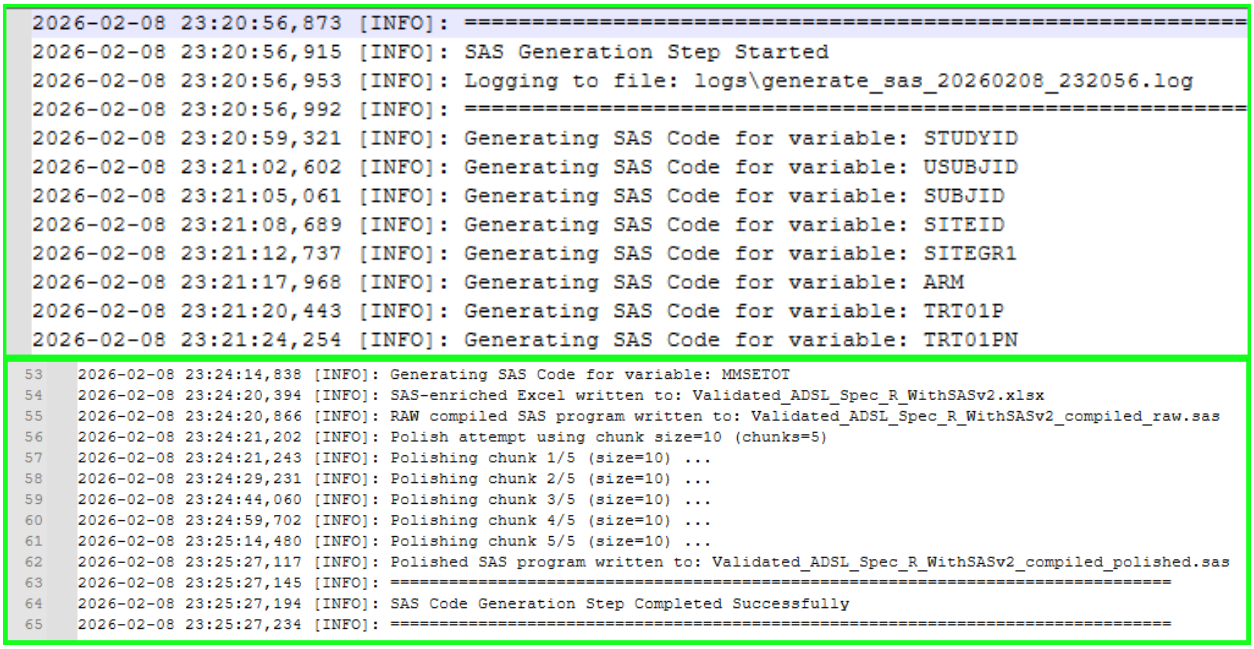

Figure 5: Logging facility for traceability

Logs include start/end markers, variable-level processing messages, parsing failures, model invocation exceptions, chunk retry events in the polish step, this ensures reproducibility and supports audit readiness.

9. Results and Observations

SAS codes for ADaM ADSL(subject-level analysis dataset) and ADAE(adverse event analysis dataset) as an initial step were successfully generated by the framework. Specification validation detected unclear description and potential typos and suggested validated specifications of derivation. Code generation part generated code snippets per variable based on the instruction and integrated final SAS code was output to .sas file.

The Implementation demonstrated:

- Improved consistency in derivation interpretation
- Reduced manual effort in review and programming
- Enhanced SAS code quality when metadata is included
- Reliable handling of malformed JSON responses
- Significant improvement in program efficiency after code polish
- Successful mitigation of LLM truncation using adaptive chunking
- The raw vs polished SAS output provides both traceability and usability

The framework proved adaptable to real-world specification variability and scalable to additional ADaM domains beyond initial exploration.

10. Lessons Learned

Key lessons include:

- GenAI output must be handled defensively (JSON parsing + fallbacks).
- Variable-level generation improves traceability but reduces efficiency.
- A dedicated refactoring step improves program usability.
- Token limits require chunking and retry strategies.
- Maintaining raw and polished artifacts increases reviewer confidence.

11. Human in the loop (HITL)

This framework supports statistical programmers rather than replacing them. Human interventions include:

- Generated code requires review and study-specific adjustment.
- Variable dependency ordering may require enhancement.
- Some complex derivations require additional context.
- Refactoring must be reviewed to confirm logic preservation.

12. Conclusion

This paper presented a practical Python-based GenAI framework for ADaM specification validation and SAS code development using Amazon Bedrock. The phased pipeline enables structured validation, SAS code generation, and consolidated program optimization. Robust parsing, logging, and chunking strategies ensure reliable execution in real-world environments. By integrating GenAI into traditional statistical programming workflows, the framework demonstrates how modern tools can improve efficiency and quality while maintaining traceability and governance readiness with human accountability.

13. Acknowledgments

The author wishes to express sincere gratitude to Maria Dalton, Sharad Chhetri, and Venky Chakravarthy for their valuable review and insightful feedback on this paper. Appreciation is also extended to colleagues and peers who contributed their input throughout the exploration and evaluation of this framework.

14. References

CDISC ADaM documents available at https://www.cdisc.org/standards/foundational/adam

Amazon Bedrock https://docs.aws.amazon.com/bedrock/

Anthropic Claude Model Documentation https://platform.claude.com/docs/en/build-with-claude/claude-on-amazon-bedrock

Nakaya, Ryo. 2025. "Considerations on Creation of Statistical Deliverables with Generative AI Tools" PharmaSUG SDE Japan.
https://pharmasug.org/wp-content/uploads/2025/04/PharmaSUG-Japan-2025-10.pdf

15. Contact Information

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

Pavan Kumar Tatikonda
Takeda Pharmaceutical Company
pavan.tatikonda@takeda.com

Ryo Nakaya
Takeda Pharmaceutical Company
ryou.nakaya@takeda.com

Sravan Kongara
Takeda Pharmaceutical Company
sravan.kongara@takeda.com