Paper TT09

Leveraging Generative AI for SAS Macro Testing and Test Data Preparation

Denis Gavriusev, AstraZeneca, Barcelona, Spain

ABSTRACT

GenAI is transforming clinical SAS programming by automating macro testing and test data preparation. Developers can leverage Generative AI to auto-generate comprehensive SAS macro test cases, spanning routine executions, boundary scenarios, and invalid parameter combinations, to ensure robust validation. GenAI can also generate code that converts real or synthetic datasets into test-ready data: anonymizing and masking sensitive fields, and enriching datasets with both routine and edge-case values to meet testing requirements without exposing confidential information. This integrated approach streamlines testing workflows, accelerates quality assurance, and upholds data privacy standards, empowering teams to deliver reliable and compliant analytical tools more efficiently.

This paper illustrates these ideas through personal experiments with coding-oriented assistants available in my corporate environment, which I could use safely without compliance concerns, and outlines future plans for a specialized internal assistant.

INTRODUCTION

Validation of SAS macros is critical in clinical programming: defects in, for example, denominator logic for percentages can cascade into incorrect clinical tables, rework, and potential compliance findings. Traditionally, macro testing and test data preparation involve manual test design, data creation, and validation scripting — slow, expertise-dependent, and often incomplete.

GenAI introduces a practical paradigm for programmers: automated test case generation and specification-driven test data preparation. This paper presents early experiments and lessons learned from day-to-day work with GenAI tools, focusing on what statistical programmers can adopt today, while also noting plans for scaling with an internal assistant in the future.

BACKGROUND AND MOTIVATION

CHALLENGES IN SAS MACRO TESTING

Testing SAS macros is inherently challenging due to their flexibility and complexity. Based on my experience, several recurring pain points can be highlighted:

- Macro test design is inconsistent and error-prone.
- Boundary and invalid parameter cases are often overlooked.
- Test coverage is rarely quantified.
- Complex macros with many parameters imply combinatorial scenario growth.
- Time pressure leads to abbreviated testing.

In practice, coverage means making sure that tests exercise not only common parameter settings but also boundary conditions and business rules.

CHALLENGES IN TEST DATA PREPARATION

Besides the challenges of macro testing, preparing suitable test data also brings several difficulties:

- Preparing datasets for validation requires anonymization or synthetic generation.
- Edge-case enrichment is tedious.
- GDPR/HIPAA and internal data protection policies limit use of real data and prohibit personal identifiable information (PII) in prompts.

- Consistency across multiple scenarios is difficult to maintain manually.

THE ROLE OF GENAI

GenAI can support programmers by:

- Generating test cases (routine, boundary, error) and suggesting assertions.
- Producing specification-driven anonymization/enrichment code that preserves analytical value and creates realistic edge cases.
- Enabling broader coverage without proportional manual effort, while allowing automated checks against specifications and CT.

METHODOLOGY: PROMPT ENGINEERING FOR SAS MACRO TESTING

STRUCTURED PROMPTING

At this stage, I work with a small set of reusable prompts rather than a large prompt library. A typical template prompt for macro testing includes:

- **Role**: Act as a clinical SAS programmer responsible for ensuring quality and compliance in programming deliverables. The role emphasizes knowledge of CDISC standards, validation practices, and macro development. The AI is expected to mirror the mindset of a programmer who anticipates regulatory needs, edge cases, and the expectations of quality assurance reviewers.

- **Task**: Generate test cases for the provided macro and requirements.

- **Requirements**: Routine, boundary, and error scenarios; CDISC alignment; positive/negative tests; expected outcomes and assertions.

- **Output**: Runnable SAS code with comments and validation checks.

- **Constraints**: No PII; deterministic seeds; follow naming/label/format conventions.

These prompts are refined iteratively. Outputs are treated as drafts and always reviewed manually.

CATEGORIES OF TEST CASES

I experiment with tests across:

- **Routine executions**: common parameter combinations.

- **Boundary scenarios**: empty and single-record inputs, extreme values, missingness, 0%/100% percentages.

- **Invalid parameters**: non-existent variables, wrong data types, conflicting flags, mismatched lists/orders.

GENERATIVE AI FOR TEST DATA PREPARATION

Beyond generating test cases, another critical application where GenAI shows significant promise is in the preparation of test data.

SPECIFICATION-DRIVEN APPROACH

In test data preparation, one promising use of GenAI is to build upon existing dataset specifications. Instead of hardcoding variables, the AI can generate code guided directly by metadata. Specifications (e.g., ADSL) provide variable types, lengths, derivations, controlled terminology, formats, and dependencies.

Using this metadata, GenAI-generated code can:

- Identify PII for anonymization.
- Apply consistent date shifting across spec-defined date variables.
- Preserve distributions within valid ranges and recompute derived variables per spec.
- Flag deviations from CDISC CT.
- Generate labeled edge cases (e.g., screen failures, missing baseline, extreme values) as separate datasets.

VALIDATION FRAMEWORK (CONCEPTUAL)

A complementary validation macro can check:

- Required variables, attributes, and labels against the spec.
- Derived variable logic (e.g., BMI, age groupings).
- Population flags per project-defined rules.
- Controlled terminology compliance.
- Basic distribution sanity.

REPRODUCIBILITY

For cases where reproducibility is important, I save seeds and offsets locally so that the same results can be regenerated. Prompt versions and outputs are also saved locally to track which prompt led to which result.

CASE STUDY: %LBCTCAE MACRO

MACRO CHARACTERISTICS

As an example, I tested GenAI-assisted methods on %lbctcae, a macro I developed for assigning CTCAE grades to laboratory results (ADLB). This macro is complex, with multiple parameters and dependencies, requiring careful handling of reference ranges, units, and toxicity grading rules.

Macro declaration (abbreviated):

```
%macro lbctcae(
        dataIn       = adlb,
        whereClause  = ,
        dataModel    = adam,
        dataOut      = ,
        spidVar      = lbspid,
        spidExl      = yes,
        subjid       = usubjid,
        testcdVar    = lbtestcd,
        resNumVar    = ,
        resCharVar   = ,
        resUnitVar   = ,
        specVar      = lbspec,
        ulnVar       = ,
        llnVar       = ,
        nrindVar     = ,
        baseFlVar    = ,
        toxgrVar     = ,
        toxgrnVar    = ,
        fileType     = xpt,
        ctcaeFile    = %str(…/ctcaev5/ctcv5ref.xpt),
        ctcaeDS      = ctcv5ref,
        debug        = no,
        unitCheck    = no,
        nrindCheck   = no,
        critVarCheck = no
);
```

This declaration illustrates the macro's complexity: multiple parameters, optional flags, and dependencies. These parameters provide the basis for AI-generated test cases. For example, GenAI models attempted to vary ulnVar and llnVar to test boundary conditions, toggle flags such as unitCheck and critVarCheck, and explore invalid scenarios by misusing or omitting required parameters.

AI-GENERATED TEST SUITE (EXPERIMENTAL)

Using Claude 4.0 and Mistral Medium 3, I generated a suite of test cases covering:

- **Routine**: valid lab results with correct reference ranges and expected grades.
- **Boundary**: missing reference ranges, values at exact ULN/LLN boundaries, unusual units.

- **Error**: unsupported test codes, inconsistent parameter combinations, or invalid unit mappings.

Some illustrative examples of AI-generated macro calls:

```
/* Routine: expected usage */
%lbctcae(dataIn=adlb, dataOut=out1, ulnVar=uln, llnVar=lln, toxgrVar=toxgr);

/* Boundary: missing reference limits */
%lbctcae(dataIn=adlb, dataOut=out2, ulnVar=, llnVar=, toxgrVar=toxgr);

/* Error: non-existent parameter (generated by AI incorrectly, e.g., 'visitDate=adt')
*/
%lbctcae(dataIn=adlb, dataOut=out3, visitDate=adt, toxgrVar=toxgr);

/* Error: confusion between parameters */
%lbctcae(dataIn=adlb, dataOut=out4, ulnVar=lln, llnVar=uln, toxgrVar=toxgr);
```

These examples show both the usefulness of AI in exploring edge cases and the types of mistakes (e.g., introducing non-existent parameters or swapping roles) that require human review.

AI SUPPORT FOR TEST DATA PREPARATION

Beyond macro calls, I also experimented with using AI to generate code that enriches the ADLB dataset with additional records needed for testing. To achieve more relevant results, I provided the model with the ADLB specification as input, so that it could reference variable names, types, and attributes correctly.

For example, the models produced DATA steps that attempted to insert edge cases such as:

- Records with missing reference limits.
- Extreme lab values beyond plausible clinical ranges.
- Uncommon specimen types or units.

Highlighted scenarios included in the AI-generated code were:

- **Critically high values**:
  ```
  aval = anrhi * (3 + ranuni(12345));
  ```
- **Borderline low values**:
  ```
  aval = anrlo * (0.89 + ranuni(67890) * 0.1);
  ```
- **Negative values**:
  ```
  aval = -abs(ranuni(78901) * anrlo);
  ```
- **Problematic text values**:
  ```
  avalc = 'NOT DONE'; aval = .;
  ```

These highlights show how AI used existing variables as the basis for generating unusual scenarios rather than hardcoding absolute numbers.

The full AI-generated SAS code for enriching ADLB test data is provided in Appendix A.

OBSERVATIONS

The generated suite helped highlight scenarios I might have missed in a manual setup, especially edge cases and negative tests. While outputs required review and corrections, they provided useful starting points.

OBSERVED BENEFITS (QUALITATIVE)

From these experiments, several qualitative benefits emerged, even without formal metrics. The most noticeable were:

- **Efficiency**: Noticeable reduction in manual effort for designing tests and preparing datasets.
- **Coverage**: Broader scenario exploration, including boundary and negative cases.
- **Quality**: Earlier identification of logical inconsistencies and parameter handling issues.

Formal quantitative evaluation will require future pilot studies.

IMPLEMENTATION CONSIDERATIONS

CURRENT PRACTICE

In my current practice, I rely on general-purpose assistants with coding capabilities to experiment with these workflows:

- Working with Claude 4.0 and Mistral Medium 3 as general-purpose assistants.
- Small set of reusable prompts, manually refined.
- Outputs always reviewed and validated by hand.

FUTURE PLANS

Looking ahead, I plan to extend these experiments into a more structured environment:

- Build an internal multi-agent assistant specialized for statistical programming.
- Extend the prompt library with reusable templates.
- Use existing corporate validation macros within AI-generated workflows to ensure reproducibility and compliance.

CHALLENGES AND LIMITATIONS

Despite the promising results, these experiments also showed several challenges and limitations:

- AI-generated code can be syntactically correct but logically flawed.
- Context limitations for large macros.
- Variability of outputs across models and prompts.
- Synthetic data may drift or encode bias if not monitored.
- Human expertise remains essential for review.

WHAT AI GOT WRONG (PRACTICAL EXAMPLES)

From my experiments with Claude 4.0 and Mistral Medium 3:

- **Macro testing**: models sometimes generated calls to %lbctcae with non-existent macro parameters or confused the purpose of parameters (e.g., mixing up ulnVar and llnVar).
- **Test data preparation**: models occasionally used variable names not present in ADLB or introduced unrealistic values that resembled data quality issues rather than valid derivations.

These issues were straightforward to identify during review or through validation checks, underlining the importance of human-in-the-loop.

CONCLUSION

GenAI shows strong potential to support SAS macro testing and test data preparation. My experiments with Claude 4.0 and Mistral Medium 3 suggest that programmers can already benefit from automating routine and edge-case generation. While results so far are qualitative and based on personal practice, they indicate opportunities for efficiency and quality improvements. The next step is to scale these approaches through a dedicated internal assistant, moving from individual usage toward team-level adoption.

REFERENCES

- PHUSE. (2024). *Best Practices in SAS Macro Validation*. PHUSE EU Connect Proceedings.
- CDISC. (2023). *ADaM Implementation Guide v1.3*. CDISC.
- CDISC. (2024). *Controlled Terminology*. CDISC CT Snapshot.
- FDA. (2023). *21 CFR Part 11: Electronic Records; Electronic Signatures*. FDA Guidance.
- EMA. (2025). External guidance on the implementation of the European Medicines Agency Policy 0070 on the publication of clinical data for medicinal products for human use. EMA.
- ACM Computing Surveys (2023). Artificial Intelligence Applied to Software Testing: A Tertiary Study. ACM Computing Surveys, Volume 56, Issue 3.

GLOSSARY OF ACRONYMS

- **ADaM** – Analysis Data Model
- **ADLB** – Laboratory Data Analysis Dataset
- **ADSL** – Subject-Level Analysis Dataset
- **AI** – Artificial Intelligence
- **CDISC** – Clinical Data Interchange Standards Consortium
- **CT** – Controlled Terminology
- **CTCAE** – Common Terminology Criteria for Adverse Events
- **FDA** – Food and Drug Administration
- **GDPR** – General Data Protection Regulation
- **HIPAA** – Health Insurance Portability and Accountability Act
- **ICH** – International Council for Harmonisation
- **LLM** – Large Language Model
- **PII** – Personally Identifiable Information
- **QA** – Quality Assurance
- **SAS** – Statistical Analysis System
- **ULN/LLN** – Upper/Lower Limit of Normal

APPENDIX A

```
/* Enriching ADLB dataset with test values for CTCAE grading algorithm testing */

/* Step 1: Prepare source dataset */
data adlb_original;
    set adlb;
    /* Create unique identifier for tracking changes */
    _original_record = 1;
    _test_scenario = '';
run;

/* Step 2: Analyze existing data to determine ranges */
proc sql noprint;
    /* Get statistics for numeric values by parameter */
    create table param_stats as
    select
        paramcd,
        avalu,
        count(*) as n_records,
        min(aval) as min_aval,
        max(aval) as max_aval,
        mean(aval) as mean_aval,
        std(aval) as std_aval,
        min(anrlo) as min_anrlo,
        max(anrhi) as max_anrhi,
        mean(anrlo) as mean_anrlo,
        mean(anrhi) as mean_anrhi
    from adlb
    where aval is not missing and anrlo is not missing and anrhi is not missing
    group by paramcd, avalu
    having count(*) >= 10; /* Only parameters with sufficient records */
quit;

/* Step 3: Create test scenarios */
data adlb_test_cases;
    set adlb;

    /* Counter for selective record modification */
    retain record_counter 0;
    record_counter + 1;

    /* Flag to determine if record needs modification */
    modify_record = 0;

    /* Select every 35th record for modification, only if necessary data exists */
    if mod(record_counter, 35) = 0 and
       aval ne . and anrlo ne . and anrhi ne . and
       paramcd ne '' and avalu ne '' then do;
        modify_record = 1;
    end;

    /* If record not suitable for modification, keep as is */
    if modify_record = 0 then do;
        _original_record = 1;
        _test_scenario = '';
        output;
    end;
    else do;
        /* Save original values */
        orig_aval = aval;
        orig_avalc = avalc;
        orig_anrind = anrind;

        /* Calculate ranges for test values */
        normal_range = anrhi - anrlo;
```

```
/* Create different test scenarios */

/* Scenario 1: Original record */
_original_record = 1;
_test_scenario = 'ORIGINAL';
output;

/* Scenario 2: Critically high values (Grade 4) */
if anrhi > 0 then do;
    aval = anrhi * (3 + ranuni(12345)); /* 3-4x above upper limit */
    avalc = strip(put(aval, best12.));
    anrind = 'H';
    _original_record = 0;
    _test_scenario = 'CRITICAL_HIGH';
    output;
end;

/* Scenario 3: Critically low values (Grade 4) */
if anrlo > 0 then do;
    aval = anrlo * (0.1 + ranuni(23456) * 0.2); /* 10-30% of lower limit */
    avalc = strip(put(aval, best12.));
    anrind = 'L';
    _original_record = 0;
    _test_scenario = 'CRITICAL_LOW';
    output;
end;

/* Scenario 4: Moderately high values (Grade 2-3) */
if anrhi > 0 then do;
    aval = anrhi * (1.5 + ranuni(34567) * 0.8); /* 1.5-2.3x above normal */
    avalc = strip(put(aval, best12.));
    anrind = 'H';
    _original_record = 0;
    _test_scenario = 'MODERATE_HIGH';
    output;
end;

/* Scenario 5: Moderately low values (Grade 2-3) */
if anrlo > 0 then do;
    aval = anrlo * (0.4 + ranuni(45678) * 0.3); /* 40-70% of lower limit */
    avalc = strip(put(aval, best12.));
    anrind = 'L';
    _original_record = 0;
    _test_scenario = 'MODERATE_LOW';
    output;
end;

/* Scenario 6: Borderline values (slightly above normal) */
if anrhi > 0 then do;
    aval = anrhi * (1.01 + ranuni(56789) * 0.1); /* 1-11% above normal */
    avalc = strip(put(aval, best12.));
    anrind = 'H';
    _original_record = 0;
    _test_scenario = 'BORDERLINE_HIGH';
    output;
end;

/* Scenario 7: Borderline values (slightly below normal) */
if anrlo > 0 then do;
    aval = anrlo * (0.89 + ranuni(67890) * 0.1); /* 89-99% of lower limit */
    avalc = strip(put(aval, best12.));
    anrind = 'L';
    _original_record = 0;
    _test_scenario = 'BORDERLINE_LOW';
    output;
```

```
        end;

        /* Scenario 8: Negative values (where not applicable) */
        if anrlo > 0 then do;
            aval = -abs(ranuni(78901) * anrlo); /* Negative values */
            avalc = strip(put(aval, best12.));
            anrind = 'L';
            _original_record = 0;
            _test_scenario = 'NEGATIVE_VALUE';
            output;
        end;

        /* Scenario 9: Zero values */
        aval = 0;
        avalc = '0';
        anrind = 'L';
        _original_record = 0;
        _test_scenario = 'ZERO_VALUE';
        output;

        /* Scenario 10: Extremely high values */
        if anrhi > 0 then do;
            aval = anrhi * (10 + ranuni(89012) * 90); /* 10-100x above normal */
            avalc = strip(put(aval, best12.));
            anrind = 'H';
            _original_record = 0;
            _test_scenario = 'EXTREME_HIGH';
            output;
        end;

        /* Restore original values for next iteration */
        aval = orig_aval;
        avalc = orig_avalc;
        anrind = orig_anrind;
    end;
run;

/* Step 4: Add problematic text values */
data adlb_text_cases;
    set adlb;

    retain text_counter 0;
    text_counter + 1;

    /* Select every 50th record for adding text problems */
    if mod(text_counter, 50) = 0 and avalc ne '' then do;

        /* Original record */
        _original_record = 1;
        _test_scenario = 'ORIGINAL';
        output;

        /* Problematic text values */
        orig_avalc = avalc;
        orig_aval = aval;

        /* Missing values */
        avalc = '';
        aval = .;
        _original_record = 0;
        _test_scenario = 'MISSING_VALUES';
        output;


        /* Non-numeric values */
        avalc = 'NOT DONE';
```

```
        aval = .;
        _original_record = 0;
        _test_scenario = 'NON_NUMERIC';
        output;

        /* Values with symbols */
        avalc = '>1000';
        aval = .;
        _original_record = 0;
        _test_scenario = 'SYMBOL_VALUE';
        output;

        /* Restore values */
        avalc = orig_avalc;
        aval = orig_aval;
    end;
    else do;
        _original_record = 1;
        _test_scenario = 'ORIGINAL';
        output;
    end;
run;

/* Step 5: Combine all test cases */
data adlb_enriched;
    set adlb_test_cases adlb_text_cases;

    /* Add label to identify test records */
    if _original_record = 0 then do;
        usubjid = strip(usubjid) || '_TEST';
        /* Add prefix to visit for differentiation */
        if visit ne '' then visit = 'TEST_' || strip(visit);
    end;

    /* Sort by subject and time */
    proc sort;
        by usubjid paramcd avisitn adt;
run;
```