PharmaSUG 2026 - Paper AI-420

A specification-driven approach to improve the reliability of AI-generated SDTM transformation programs

Rostislav Markov, Amazon Web Services, Inc., New York, USA

ABSTRACT

Clinical data transformation pipelines are increasingly augmented with artificial intelligence (AI)-assisted code generation. However, in regulated environments, the uncontrolled use of AI can introduce unacceptable risks, including incorrect transformations, inconsistent traceability, and increased validation burden. This paper presents a practical, specification-driven approach to governing AI-assisted function development using software development lifecycle (SDLC) artifacts including Behavior-Driven Development (BDD) specifications, test contracts, and project conventions.

We evaluate how different levels of AI prompt governance affect the correctness and integration quality of generated transformation functions. Using eight production functions from an SDTM transformation framework, we compare loosely specified AI prompts against increasingly governed prompts grounded in BDD specifications, structural conventions, and test scripts. Each AI-generated output is scored against the production ground truth using a binary functional correctness checklist and a structured integration quality rubric.

Results show that BDD specifications alone improve functional correctness from 20% to 84% – the single largest gain. Adding project conventions and test contracts further improves functional correctness to 90% and unlocks integration quality from 0% to 70%. The approach integrates seamlessly into existing SDLC workflows. This paper demonstrates that specification quality, rather than model sophistication, is the primary determinant of safe and scalable AI-assisted development in clinical data pipelines.

INTRODUCTION

AI-assisted coding tools are rapidly becoming part of modern software development workflows. Statistical programming teams are exploring the use of large language models (LLMs) to accelerate the development of SDTM transformation logic and validation utilities. However, clinical data pipelines operate under strict regulatory and quality constraints. Incorrect or non-compliant transformation logic can result in submission delays, rework, or regulatory findings. Organizations need to balance the productivity gains offered by AI with the need for traceability, correctness, and validation.

A key observation in clinical programming environments is that teams already maintain transformation artifacts describing expected behavior, automated test suites with test datasets, and metadata constraints governing allowed inputs and parameters. These are not systematically exposed to AI tools during code generation, and it is not obvious whether their inclusion leads to better AI model predictions. This paper explores a simple idea: treat such artifacts as structured governance inputs for AI-assisted function generation, instead of relying on loosely specified prompts, and compare the correctness and quality of the AI model responses.

BACKGROUND

AI-ASSISTED CODING IN REGULATED ENVIRONMENTS

AI-assisted coding tools are often designed for general-purpose software development. They assume specifications are informal, manual review is sufficient, and iteration is relatively inexpensive. These assumptions do not hold in regulated clinical data pipelines, where behavior must be explicitly specified, failures are costly, and validation evidence is required for compliance. In this context, the primary constraint is not generating code, but ensuring that the generated code is correct, auditable, and consistent with regulatory expectations.

EXISTING GOVERNANCE ARTIFACTS

In this case study, SDTM transformation programs are composed from reusable Python functions deployed as AWS Lambda handlers within the GitHub Actions deployment pipeline. To compose an SDTM program, statistical programmers create a specification file (JSON) that maps source dataset fields, target dataset fields, and reusable functions. Function artifacts are stored in a Git code repository with formal validation and publication workflow including automated and manual reviews.

Each function is associated with a BDD specification file written in Gherkin syntax, as well as a pytest suite containing parameterized test cases and supporting test data. In addition, metadata files define allowed inputs, parameters, and structural constraints. The function artifacts collectively define the expected behavior of each transformation function and provide a natural basis for governing AI-generated code. Upon successful completion of the validation workflow, the function metadata is added to the search index of the reusable functions library.

PROBLEM STATEMENT

This work investigates the following question:

How does increasing the level of specification and governance in AI prompts affect the correctness and integration quality of generated transformation functions?

Rather than comparing different AI models, the paper compares model responses based on how prompts are constructed and what information is provided to the model. The hypothesis is that using existing SDLC artifacts as structured inputs to the code generation process will outperform loose prompting, and that the gains can be decomposed into two independent dimensions: functional correctness and integration quality.

METHODOLOGY

EVALUATION SAMPLE

The unit of evaluation is a transformation function module consisting of implementation code, metadata, and associated validation assets. Eight existing transformation functions were selected, representing a range of complexity from simple dose-derivation functions (~50-100 lines of production code) to complex multi-domain derivations (~400+ lines of code with external infrastructure dependencies). Table 1 provides an overview of the eight functions evaluated.

Function	Description	Complexity
get_first_dose	Earliest EXSTDTC → RFSTDTC/RFXSTDTC in DM	Low
get_last_dose	Latest EXENDTC → RFENDTC/RFXENDTC in DM	Low
get_last_contact	Latest *DTC across all domains → RFPENDTC in DM	Medium
call_patient_death	Death flags from DS/DD → DTHFL/DTHDTC in DM	Medium
call_subjid	SUBJID propagation from DC to target domains	Medium
call_suppqual	SUPPQUAL/SQAP generation from reference metadata	High
create_epoch_taetord	EPOCH/TAETORD derivation from SE with tie-breaking	High
create_se_domain	SE domain creation via SQL queries against source domains	High

Table 1. Overview of the transformation functions sample

PROMPT GOVERNANCE LEVELS

Three levels of prompt governance were defined, representing increasing degrees of specification richness.

A1 – Function description. The model receives a brief system instruction ("You are implementing a transformation function module in Python for an SDTM clinical data pipeline. Return ONLY the requested outputs. Do not include explanations unless explicitly requested."), the function name, and a prose description of its intended behavior extracted from repository documentation. No examples, no types, no

project context. The description of intended behavior comes from the function description in the repository and typically includes a list of ~10-50 items describing function logic.

A2 – Behavioral specification. Same system instruction and function name, but the prose description is replaced entirely by a complete BDD feature file in Gherkin syntax. The feature file includes (Given/When/Then) scenarios with concrete input/output (Pandas) DataFrames, domain validation scenarios with expected error messages, and edge case scenarios covering partial dates, mixed formats, missing values, and multiple subjects.

A3 – Fully governed specification. Same system instruction, function name, and BDD feature file as A2, plus three additional sections:

- Required project conventions (eight rules): import paths for shared helpers, type contracts, traceability decorator, logging pattern, module-level mappings dictionary, multi-domain iteration pattern, target domain/target key return construction, and try/except error handling with study ID logging.
- Test script: the pytest-bdd step definitions that will validate the output, providing exact function names, return types, and type instantiation patterns.
- Project context: type system definitions (source payload, target payload, source domain, target domain, options), shared helper signatures, and the mappings dictionary pattern.

EVALUATION FRAMEWORK

Each AI-generated output was scored against the production ground truth on two independent dimensions.

Functional Correctness (F%) measures whether the code solves the right problem. A generalized binary checklist of ten checks is applied consistently across all functions (see Table 2). F% equals the count of passing checks divided by ten. Each check is binary (pass/fail), derived from BDD scenarios, and independently verifiable.

Area	#	Check	Pass Criteria
Core logic (F1)	F1.1	Correct core operation	Implements the right algorithm
	F1.2	Correct join key	Links source to target on correct key(s)
	F1.3	All target variables mapped	Every target column from BDD examples is written to
	F1.4	Multi-subject support	Processes all subjects in one call
Validation (F2)	F2.1	Validates source domain	Rejects invalid source with error
	F2.2	Validates target domain	Rejects invalid target with error
	F2.3	Error messages exact match	String matches BDD-specified messages
Edge cases (F3)	F3.1	Preserves data formats	Output values match input string formats
	F3.2	Edge cases covered	All BDD scenario variants handled
	F3.3	No side effects	Does not mutate inputs or add unexpected columns

Table 2. Checklist for scoring functional correctness

Integration Quality (I%) measures whether the code fits the project. Three criteria are scored on a 0–3 scale (see Table 3). I% equals the sum of individual scores divided by nine.

Area	0	1	2	3
Imports, decorator & configuration (I1)	No sdtmutils	Partial imports	All imports, minor gaps	Exact match
Signature & structure (I2)	Wrong name/types	Partially correct	Correct with minor deviations	Exact match

Area	0	1	2	3
Test pass rate (I3)	0%	1% – 49%	50% – 99%	100%

Table 3. Checklist for scoring integration quality

The composite F% and I% scores are reported separately for each function and prompt governance level.

EXPERIMENTAL SETUP

All experiments were run in a controlled internal environment using a single AI model – Claude Sonnet 4.6 via Amazon Bedrock. The test script inclusion in A3 was empirically validated: removing it kept functional correctness but dropped integration quality (for example, from 89% to 44% for get_first_dose), confirming its contribution to structural alignment.

RESULTS

Table 4 summarizes the results for each function, separated by prompt governance level (A1-A3) and evaluation dimension (F%, I%).

Function	A1 F%	A1 I%	A2 F%	A2 I%	A3 F%	A3 I%
get_first_dose	10%	0%	90%	0%	100%	89%
get_last_dose	10%	0%	80%	0%	100%	100%
get_last_contact	20%	0%	90%	0%	90%	78%
call_patient_death	40%	0%	90%	0%	90%	67%
call_subjid	20%	0%	80%	0%	90%	56%
call_suppqual	30%	0%	80%	0%	80%	56%
create_epoch_taetord	30%	0%	80%	0%	80%	56%
create_se_domain	0%	0%	80%	0%	80%	56%
Average	**20%**	**0%**	**84%**	**0%**	**90%**	**70%**

Table 4. Evaluation results for the functions sample

Moving from prose descriptions to BDD specifications produced the largest single improvement (+64%) in functional correctness (A1 F% → A2 F%). BDD scenarios provide concrete input/output examples that disambiguate the algorithm, named variables that constrain the output schema, and edge case scenarios that prevent over-simplification. However, A2 outputs universally failed integration (0% I%) because BDD says nothing about import paths, type systems, decorators, or shared helper delegation.

Adding project conventions and test scripts improved functional correctness by 6% (A2 F% → A3 F%) and unlocked integration quality from 0% (A1 I%, A2 I%) to 70% (A3 I%). The conventions provided the structural contract that BDD cannot express, such as imports, types, iteration patterns, and return construction. The test script provided exact function names, return types, and type instantiation patterns that conventions alone did not fully constrain. A3 integration quality varied dramatically by function complexity (see Table 5).

Complexity	Functions	A3 I%	Explanation
Low	get_first_dose, get_last_dose	89% – 100%	Conventions template matched the exact pattern
Medium	get_last_contact, call_patient_death, call_subjid	56% – 78%	Core interfaces correct; ground truth uses different helpers
High	call_suppqual, create_epoch_taetord, create_se_domain	56%	Ground truth uses function-specific helpers not listed in conventions

Table 5. Integration quality by function complexity

DISCUSSION

LOOSE PROMPTING SOLVES THE WRONG PROBLEM

The most consistent failure at A1 was misidentification of the core algorithm. Without behavioral examples, the model predicted plausible but incorrect interpretations. get_first_dose and get_last_dose (<90 lines of code in production) were generated as 395- and 457-line modules respectively, with per-subject processing, treatment period detection, PK half-life extension, and data quality flagging. get_last_contact was generated as 917 lines with custom enums, dataclasses, LTFU status derivation, and safety follow-up flags — when the production function simply finds the maximum date across all DTC columns. call_subjid was generated as an ID construction function (building SUBJID from SITEID + SUBJNO templates) when the production function propagates an existing SUBJID from the DC domain. create_se_domain was generated as a static reference file merger that uploads to S3, when the production function executes dynamic SQL against source domains. A1 compensates for ambiguity by over-engineering. Average A1 output for low and medium complexity functions was three times the production code length (445 lines in A1 vs. 162 lines in production).

The transition from A1 to A2 produced the largest improvement (+64% in F%) because BDD scenarios provide three things prose cannot: concrete input/output tables that disambiguate the algorithm, named target variables (RFSTDTC, RFPENDTC, DTHFL) that constrain the output schema, and error scenarios that define validation boundaries. However, A2 code is algorithmically correct but structurally incompatible with the project. Every A2 output scored 0% on integration because BDD encodes what the function does, not how it fits into the codebase.

F2.3 (error messages exact match) was the most frequently failed check: 0/8 functions passed at A1, 5/8 at A2, and 5/8 at A3. Even when the model validates correctly, the error message string is almost never an exact match unless the BDD specification includes the literal expected message. This is a fragile but operationally important check since both statistical programmers and downstream systems process error messages.

THE AI MODEL REIMPLEMENTS WHAT IT CANNOT IMPORT

The conventions template prescribes a single structural pattern: MAPPINGS dict, validate_domain_pairs, zip (source, target, transformations) iteration, process_dm_dose_transformation delegation. Functions that follow this exact pattern (get_first_dose, get_last_dose) achieved 89–100% integration quality. Functions that require different shared helpers plateau at 56% because the model reimplements locally what it cannot import. For example, create_se_domain reimplemented SQL execution via SQLite + sqlglot instead of using AthenaDatabase. call_suppqual reimplemented code list lookup instead of importing from sdtmutils.codelist. These reimplementations are often functionally reasonable but structurally incompatible with the test harness.

Three functions required knowledge that no prompt level could provide. The tie-breaker reference file schema for create_epoch_taetord was invented incorrectly by all three prompt levels. The EpochSQLCode column in create_se_domain contains executable SQL that must be run against source domains, a pattern no specification described. The convert_to_iso8601 cross-function dependency in call_suppqual for DATE class variables was missed by all levels. These represent irreducible knowledge gaps: information that exists in the codebase and functions registry but could not be inferred from the specifications.

PRACTICAL IMPLICATIONS

The evaluation results suggest the following lessons.

1. BDD is the single highest return on investment. Moving from A1 to A2 added +64% in functional correctness. If you can only do one thing, write BDD scenarios with concrete input/output examples. BDD specifications are a reusable asset. The same BDD feature files that improved code generation also serve as executable documentation, regression tests, and onboarding material.
2. Treat reference file schemas as first-class specifications. The reference files for

create_epoch_taetord and call_suppqual were the primary sources of functional errors. Documenting these schemas in the BDD specification would close the gap.

3. Project conventions must consider function specificity. A generic conventions template works for pattern-matching functions but degrades for complex ones. Consider maintaining a library of convention templates per function archetype.
4. Include the test script in your prompt. Removing it halved integration quality in the evaluated sample. The test script provides exact function names, return types, and type instantiation patterns that conventions alone do not fully constrain.
5. List every shared helper the function needs. The model will use what you list and reimplement what you do not. Incomplete helper lists are the primary cause of integration failures in complex functions.
6. AI-generated code is not production-ready without structural context. Even at A3, average integration quality is 70%. Simple functions benefit the most. Functions that follow a repeatable pattern achieved 100% F% and 89% - 100% I% at A3. Functions with unique infrastructure dependencies plateau at 56% I% regardless of prompt quality.
7. Standardize transformation patterns. The dose functions achieved high scores because they follow an identical pattern. The more your transformation functions share a common architecture, the more effectively the AI model can generate them.
8. Use AI-generated code as a convention compliance probe. Comparing AI outputs to ground truth revealed five undocumented conventions embedded in production code (e.g., reference file loading patterns, per-archetype error handling). This creates an automated feedback loop: generate, score, extract gaps, update conventions – surfacing implicit organizational knowledge with each cycle and simultaneously improving AI output.

CONCLUSION

AI-assisted development can be safely and effectively introduced into regulated SDTM transformation pipelines by treating existing specifications as prompt-level governance. BDD specifications alone improve functional correctness from 20% to 84%, representing the single largest gain observed. Adding project conventions and test contracts further improves correctness to 90% and unlocks integration quality from 0% to 70%.

The results decompose cleanly. BDD governs what the function does (functional correctness), while conventions govern how it fits the project (integration quality). Neither alone is sufficient. Together, they shift errors from semantic (=wrong algorithm) to mechanical (=wrong import path), making failures easier to detect and cheaper to correct.

Rather than focusing on model selection or fine-tuning, organizations should invest first in improving specification quality and systematically exposing those specifications to AI tools. This shift enables scalable, compliant, and auditable AI-assisted development aligned with regulatory expectations.

ACKNOWLEDGMENTS

The author would like to thank the statistical programming team led by Janet Low at Merck & Co., Inc. for making this study possible.

CONTACT INFORMATION

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

Rostislav Markov
rmarkov@amazon.com
https://www.linkedin.com/in/rostislavmarkov/