Paper ML35

Democratizing SDTM Transformations: AI-Assisted Function Development with Business-Driven Governance

Rostislav Markov, Amazon Web Services, Inc., New York, USA
Jacques Lanoue, Merck & Co., Inc., North Wales, USA
Ulf Nielsen, MSD Merck Sharp & Dohme AG, Zurich, Switzerland

ABSTRACT

Traditional SDTM dataset generation systems are constrained by rigid software development lifecycles, limiting adaptation to evolving study needs. We present an innovative “Bring Your Own Function” (BYOF) framework that democratizes function development through context-aware AI assistance and study-specific scoping.

Our solution leverages a comprehensive code registry that serves as context for AI agents, containing validated functions with strict templates for code, metadata (JSON format), automated tests, and human-readable requirements. New functions are introduced as study-specific business objects, enabling targeted protocol-specific adaptations without system-wide impact.

Key innovations include: 1) comprehensive function templates with code, metadata, and tests, 2) study-scoped deployment of mapping functions and derivations without system-wide impact, 3) context-aware AI agents generating complete function packages through natural language interaction, 4) automated validation against registry patterns, and 5) rapid business-data publication workflow.

Our results demonstrate that combining AI assistance with robust governance frameworks enables safe democratization of Study Data Tabulation Model (SDTM) transformation development while maintaining compliance standards and data quality.

INTRODUCTION

SDTM dataset generation relies on mapping functions and derivations—typically SAS programs—which are managed centrally across all studies in the organization. This centralized governance ensures accuracy and performance but creates a critical bottleneck: changes require tedious reviews before global release, introducing delays in time-sensitive activities. This constrains statistical programmers in rapidly adapting to study-specific needs, despite every study presenting (some form of) novel requirements where customizability is essential.

We address this constraint by reimagining the statistical programmer’s role as analogous to software developers who contribute directly to reusable code repositories. Our **“Bring Your Own Function” (BYOF)** framework enables programmers to write, test, deploy and publish study-scoped derivations within minutes, while maintaining compliance standards.

Successfully integrating artificial intelligence (AI) into this regulated environment requires rejecting “black box” approaches where AI autonomously generates transformations without transparency and reusability. We observe attempts to give statistical programmers AI code generators without context engineering and without a workflow to validate generated code and make it reusable for other studies. Such throwaway approaches to function generation compromise traceability and create unsustainable technical debt. Instead, our framework positions AI as a code generation assistant within a structured workflow:

1. Statistical programmers architect function designs and express requirements using a formal syntax
2. Context-aware AI generates code suggestions based on validated repository patterns
3. Comprehensive templates enforce standards (code, metadata, tests, documentation)
4. Study-specific scoping prevents system-wide impact from new, unvalidated functions

This approach makes use of AI to handle implementation details while programmers focus on architecture, business logic, and system scalability. We believe the optimal outcome is not the fastest code generation, but the highest quality—accurate, performant, and broadly reusable functions leading to minimal maintenance effort across hundreds of studies. Next, we describe the BYOF framework which we’ve adopted at Merck/MSD for all new studies in production.

ADAPTIVE FRAMEWORK FOR MAPPING FUNCTIONS AND DERIVATIONS

We reimagine the traditionally centralized management of mapping functions and derivations by enabling business-driven governance. While our SDTM engine software runs data transformation functions on demand, the functions themselves are business objects that can be rapidly adapted and scoped to specific studies outside the software development lifecycle (SDLC).

GUIDING PRINCIPLES

1. **Agility**. Software systems supporting hundreds of studies per year, each with diverse data collection systems and schemas, should enable function modifications within hours, not via quarterly releases. Our "preview" environment provides self-service testing and frictionless publication workflows.
2. **Reusability**. Code reused across studies represents validated quality. While AI can generate customized code instantly, unique implementations increase cognitive load and limit the organizational capacity to manage lifecycles and respond to regulatory queries. Version-controlled storage enables reproducibility, with reuse frequency serving as a quality metric.
3. **Transparency**. Comprehensive logging captures all publication events, changes, function invocations, and errors. This enables auditability and traceability. Pull request reviews and preview testing enforce human-in-the-loop validation. Successfully published functions enter a discoverable registry (our functions repository).

KEY COMPONENTS OF THE BYOF FRAMEWORK

CODE REPOSITORY

The BYOF code repository (GitHub) enforces a minimal but consistent standard for each transformation function: Python (Pandas-based) execution logic, JSON metadata describing requirements (for example, applicable source forms and required reference files), pytest test cases, and validation test data. The repository serves a dual purpose:

1. Supporting transparent code governance
2. Acting as AI prompting context by providing curated examples of validated functions

Base/global functions are designed to address ~80% of common study needs while serving as few-shot learning examples for AI-assisted code generation of custom functions. Business requirements captured as Behavior-Driven Development (BDD)-style specifications (feature files) provide natural, structured invocation points for AI tools to propose, refine, and validate new implementations.

```
byof-functions/
├── call_study_day/
│   ├── v1/
│   │   ├── app.py
│   │   ├── metadata.json
│   │   ├── __init__.py
│   │   └── tests/
│   │       ├── scenarios.feature
│   │       ├── steps/
│   │       └── data/
│   ├── v2/
│   │   └── ...
├── derive_ae_end_date/
│   ├── v1/
│   │   ├── app.py
│   │   ├── metadata.json
│   │   ├── __init__.py
│   │   └── tests/
│   │       ├── scenarios.feature
│   │       ├── steps/
│   │       └── data/
├── scripts
```

Figure 1. BYOF code repository structure

Maintaining multiple versions as separate folders is intentional. A subsequent function version is not necessarily an "upgrade" that replaces the previous one. A newer version reflects a variation or extension of the behavior that does not make existing versions obsolete. This allows different studies (or even different mappings within a study) to continue using the version that best matches their original requirements.

Each version has its own metadata.json because the functionality, parameters, inputs/outputs, or applicable studies may differ. This ensures accurate discovery, governance, and traceability at the version level.

Each tests/ directory has the following layout:

- scenarios.feature – Gherkin scenarios that describe how the function should behave across different conditions.
- steps/ – step definitions binding Gherkin steps to executable test code.
- data/ – input/output datasets used by the scenarios.

The repository is designed to make function development reproducible and well-governed. A standardized local setup (Python 3.11, virtual environments, and the uv package manager) is documented for both Mac and Windows,

including commands to keep shared SDTM utility and function packages up to date. This reduces environment drift and supports consistent behavior across contributors.
A BYOF sub-module generator (scripts/initialize_module.py) interactively collects key metadata (for example, study IDs, valid source/target domains and fields, function type) and scaffolds a new module: versioned directory, metadata.json, and a Python function template. This enforces consistent layout and metadata standards.
Quality assurance is built in through a common toolchain: Pylint and Flake8 for static code analysis (invoke lint), Black for formatting (invoke format), and pytest (including BDD feature files) for testing (invoke test).
Metadata files use an extensible JSON structure, for example:

```
{
    "id": "<auto-generated-uuid>",
    "arn": "resourceId",
    "moduleName": "byof_functions",
    "packageName": "sdtmfunctions",
    "subModuleName": "<your_module_name>",
    "version": "1",
    "name": "<your_module_name>",
    "displayName": "<your_module_name>",
    "description": "Description about <your_module_name>",
    "studyIds": ["SID.1", "SID.2"],
    "lookUpCodeList": true,
    "validSourceDomains": ["SDM1", "SDM2"],
    "validTargetDomains": ["TDM1", "TDM2"],
    "validSourceFields": ["SF1", "SF2"],
    "validTargetFields": ["TF1", "TF2"],
    "functionType": "derivation"
}
```

FEATURE SPECIFICATIONS

Each BYOF function is accompanied by a dedicated feature file that serves as its requirements specification, written in BDD syntax. These feature files describe the expected behavior of the function in business terms, using concrete, testable examples that can be automated as acceptance tests. Below is an example feature file for the study_day derivation.

```
Feature: call_study_day function

  The CALL_STUDY_DAY function is designed to calculate the reference study day of a
given date,
  relative to RFSTDTC from the DM domain, move the derived value to an existing
xxDY/xxxxDY variable in the target dataset, and record the traceability information.
  This function ensures data consistency and provides traceability of mapping.

  Scenario Outline: Study day is derived from RFSTDTC in DM and date in source
variable
    Given the <source_dataframe> exists
    And the <source_variable> exists
    And value within <source_variable> is a date
    And the RFSTDTC exists in <DM>
    And the <target_variable> exists in the <target_dataframe>
    When the call_study_day function executes
    Then the <target_variable> in <target_dataframe> has the derived study_day

    Examples:
      | source_dataframe | source_variable               | DM      | target_variable
| target_dataframe |
      | CM,LB,EX         | CMSTDTC,CMENDTC,LBDTC,EXSTDTC | RFSTDTC |
CMSTDY,CMENDY,LBDY,EXSTDY | CM,LB,EX         |

  Scenario Outline: RFSTDTC existence
    Given the <source_dataframe> exists
    And the <source_variable> exists
```

```
    And value within <source_variable> is a date
    And the RFSTDTC does not in <DM>
    And the <target_variable> exists in the <target_dataframe>
    When the call_study_day function executes
    Then the call_study_day function throws an error

    Examples: Source column contains date and RFSTDTC does not exists
      | source_dataframe | source_variable               | DM      | target_variable
| target_dataframe |
      | CM,LB,EX         | CMSTDTC,CMENDTC,LBDTC,EXSTDTC | RFSTDTC |
CMSTDY,CMENDY,LBDY,EXSTDY | CM,LB,EX         |

  Scenario Outline: Source variable does not contain a date
    Given the <source_dataframe> exists
    And the <source_variable> exists
    And value within <source_variable> is a not a date
    And the RFSTDTC exists in <DM>
    And the <target_variable> exists in the <target_dataframe>
    When the call_study_day function executes
    Then the call_study_day function returns an empty list

    Examples: Source column does not contain date and RFSTDTC exists
      | source_dataframe | source_variable               | DM      | target_variable
| target_dataframe |
      | CM,LB,EX         | CMSTDTC,CMENDTC,LBDTC,EXSTDTC | RFSTDTC |
CMSTDY,CMENDY,LBDY,EXSTDY | CM,LB,EX         |

  Scenario Outline: Target variable does not exist
    Given the <source_dataframe> exists
    And the <source_variable> exists
    And value within <source_variable> is a date
    And the RFSTDTC exists in <DM>
    And the <target_variable> does not exist in the <target_dataframe>
    When the call_study_day function executes
    Then the call_study_day function throws an error

      Examples: Target column does not exist in target dataframe
      | source_dataframe | source_variable               | DM      | target_variable
| target_dataframe |
      | CM,LB,EX         | CMSTDTC,CMENDTC,LBDTC,EXSTDTC | RFSTDTC |
CMSTDY,CMENDY,LBDY,EXSTDY | CM,LB,EX         |
```

PUBLICATION WORKFLOW

An automated pipeline for continuous integration and deployment (GitHub Actions) takes programmer-authored functions through validation, deployment, and registration so they can be discovered and used within the execution engine. When code is committed to the BYOF GitHub repository, the pipeline packages the functions as reusable layers, assigns a new version, and automatically updates the shared BYOF execution function to use the latest version. This event-driven workflow ensures that new or updated functions become available without manual deployment steps.

At the same time, function and derivation metadata (for example, name, version, scope, and associated study) is uploaded and registered in the search index (OpenSearch) used by the graphical user interface (GUI). This enables users to easily discover and enable appropriate study-specific or global functions for their study. The automated pipeline also builds the compute resources, and performs consistency checks including:

- Metadata validation
- Version conflict prevention
- Visibility enforcement (typically study-scoped for new versions pending roadmap review)

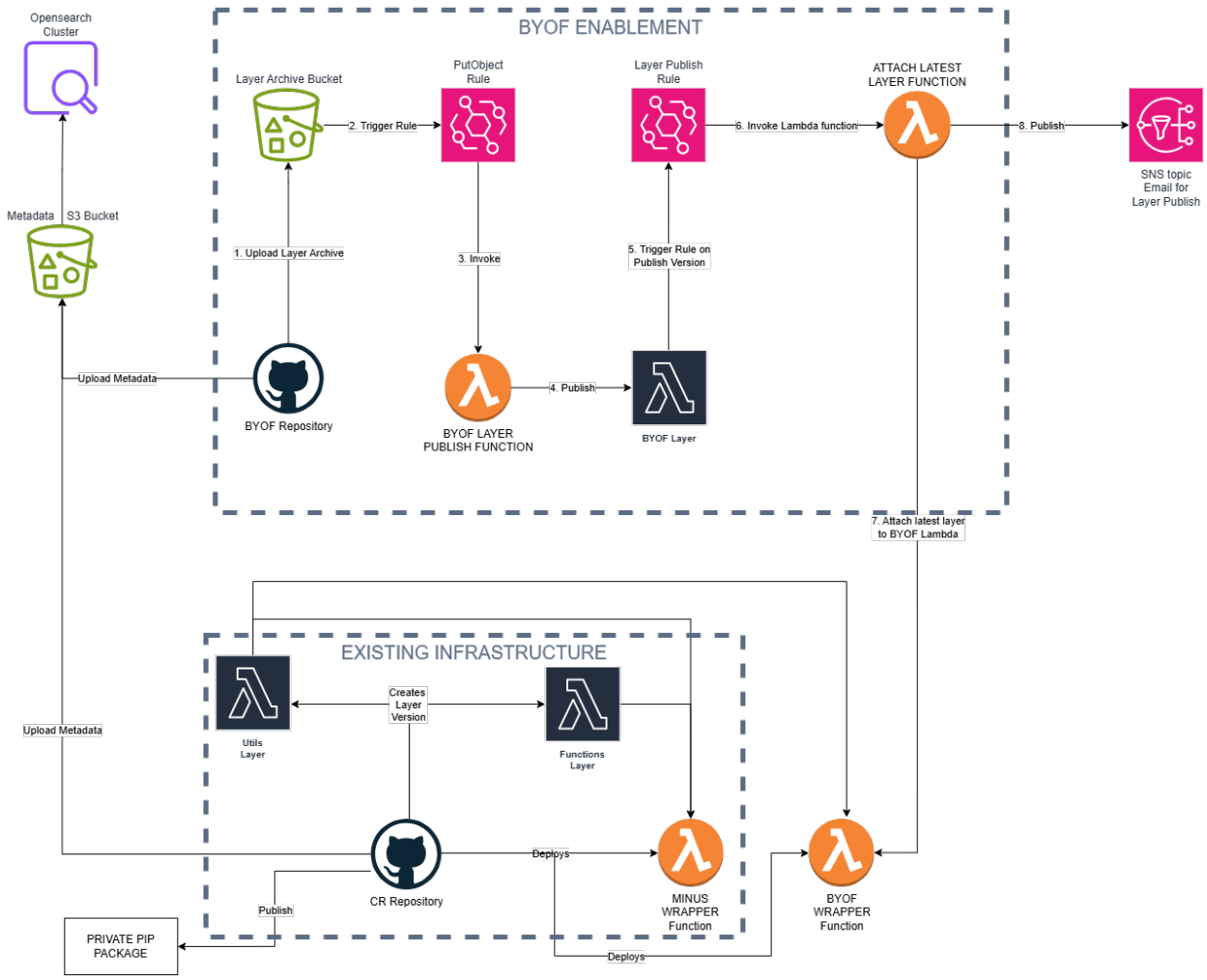

Figure 2. BYOF publication workflow

FUNCTIONS REGISTRY

The web GUI provides a searchable registry of both global and study-specific functions, allowing users to quickly find the right derivations based on name, purpose, domain, or study context. Users can hover over function names to reveal a concise description of the underlying code, supporting quick comprehension without leaving the screen. Only functions authorized for a given study are surfaced and executable, ensuring controlled reuse of logic while preserving study-level isolation and governance.

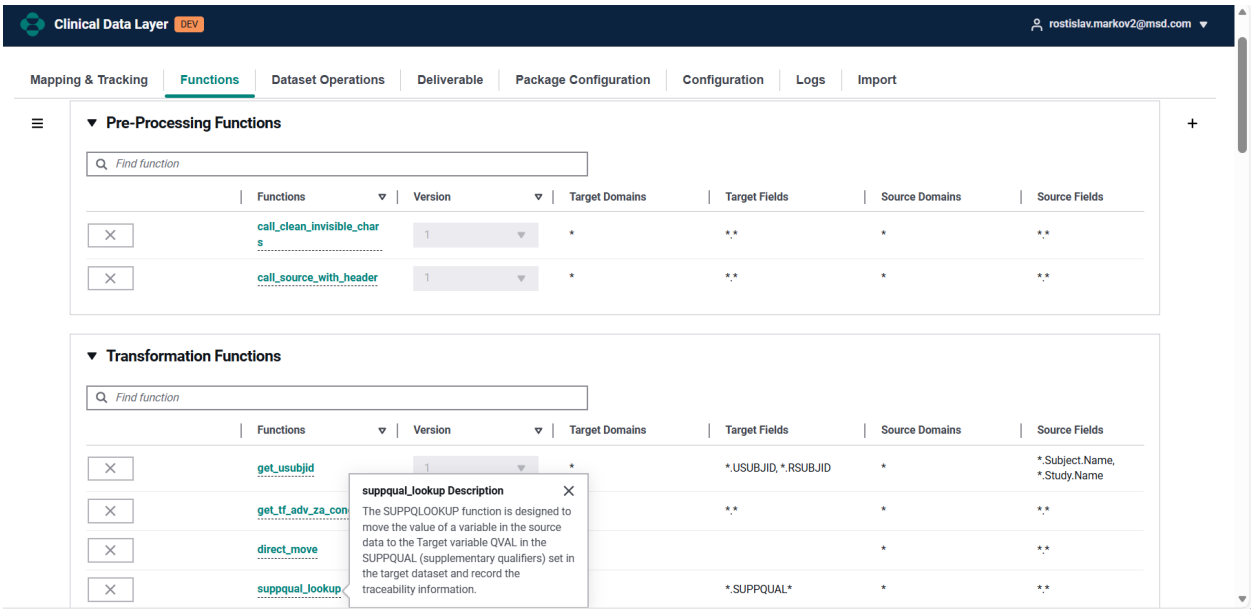

Figure 3. Graphical user interface of the BYOF repository

PROGRAMMATIC CODE EXECUTION VIA MAPPING SPECIFICATIONS

Mapping specifications use an extensible JSON syntax that allows straightforward referencing of study-specific functions as published business objects. A correctly formulated mapping specification acts as the “contract” between the execution engine and the underlying function code, ensuring that the intended logic is executed in a consistent and traceable way. The example below illustrates how variables and associated transformation and derivation functions are referenced for execution:

```
{
  "mappings": [
    {
      "SourceTable": "AE",
      "SourceVariable": "AEENDTC_RAW",
      "TargetList": [
        {
          "FunctionName": "ISO8601 (v1)",
          "SDTMTransformationFunction": "functionId",
          "SDTMVariable": "AEENDTC",
          "Group": 1,
          "SDTMDomain": "AE"
        }
      ]
    }
  ],
  "derivations": [
    {
      "Function": "derivationId",
      "FunctionName": "get_informed_consent",
      "Version": 1,
      "Sequence": 0
    }
  ],
  "transformationFunctions": [
    {
      "Function": "functionId",
      "FunctionName": "std_fn_split_text",
      "Version": 1
    }
  ]
```

```
}
```

TRACEABILITY SYSTEM

Automated logging captures complete audit trail for compliance and debugging. Each function reads an input dataframe, performs transformations, and emits detailed JSON-based traceability logs including study metadata, mapping specification versions, function name, timestamps, and complete source/target domain, record, and variable mappings. A data pipeline publishes these traces into the search cluster to enable real-time, text-based search and analysis of source-to-target mappings at the record and variable level. To keep transformation code clean, Python decorators are used to encapsulate traceability logging concerns outside the core business logic.

WORKFLOW CONTROLS

New guardrails enable separation of duties and allow business users to introduce new functions and derivations safely, so they can be run reliably.

1. **Automated code scans**. GitHub Actions perform security scans and compliance checks through standard CI/CD tooling.
2. **Study-specific scoping**. Publication guardrails restrict unauthorized reuse through visibility controls.
3. **Enforced traceability**. Complete logging of actor, action and date/time for function registration, use in mapping specifications, executions, and outcomes.
4. **Periodic reviews**. Manual assessment of repository additions informs function promotion decisions and broader organizational visibility.

AI-ASSISTED FUNCTIONS DEVELOPMENT

We illustrate how AI assistants integrate with the BYOF repository to support end-to-end function development. Rather than using AI purely as ad-hoc code generator, our approach leverages the BYOF framework assets including function versioning, metadata, BDD specifications, and publication workflow to enable an **agentic workflow**. The AI assistant, for example an agentic integrated development environment (IDE), can interpret requirements, create specifications, generate and refine code, and validate behavior in a governed, reproducible manner as follows.

The BYOF repository is intentionally designed to be machine- and human-readable. The repository structure makes extensive use of formal syntax such as BDD and Pytest, which naturally supports AI agents as **knowledge base**. This allows the AI assistant to play several roles:

1. **Requirements interpreter** turning transformation needs expressed in natural language into BDD feature files and metadata content
2. **Function recommender** searching the repository to suggest existing functions/versions that fit the programmer's need
3. **Code assistant** generating new function implementations or refactoring/extending existing functions
4. **Validation and QA assistant** proposing new test scenarios for edge cases, test data variations in tests/data/ and interpreting failing tests.

We describe this workflow using adaptations of the study-day derivation function, call_study_day, as a running example. Study-day derivations are widely used and frequently require study-specific logic, making them a natural candidate for versioned BYOF functions.

FROM NATURAL LANGUAGE TO BDD SPECIFICATION

A typical interaction begins with a study programmer expressing a need in natural language. For example:

"For Study SID.123, we need a modified call_study_day derivation. Instead of using RFSTDTC as Day 1 for all records, this protocol defines Day 1 by treatment period (different reference dates per EPOCH), and requires specific handling for partial dates (no imputation; leave day missing but still record traceability). Other studies must continue to use the original call_study_day behavior."

The AI assistant can then discover existing files and draft BDD requirements, converting the description into a Gherkin syntax:

```
Feature: call_study_day (v2) function

    The CALL_STUDY_DAY (v2) function calculates study day relative to a
    period-specific reference date (per EPOCH) and applies study-specific
    rules for partial dates, while recording full traceability.

    Scenario Outline: Derive study day using period-specific reference dates
        Given the <source_dataframe> exists
        And the <source_variable> exists
        And value within <source_variable> is a date
        And the period-specific reference dates exist in <DM> (e.g., RFSTDTC by EPOCH)
        And the <target_variable> exists in the <target_dataframe>
        When the call_study_day v2 function executes
        Then the <target_variable> in <target_dataframe> has the derived study_day
```

```
            based on the correct period-specific reference date

        Examples:
          | source_dataframe | source_variable | DM   | target_variable |
target_dataframe |
          | EX               | EXSTDTC         | DM   | EXSTDY          | EX
|

    Scenario Outline: Partial dates are not imputed
        Given the <source_dataframe> exists
        And the <source_variable> exists
        And value within <source_variable> is a partial date
        And the period-specific reference dates exist in <DM>
        And the <target_variable> exists in the <target_dataframe>
        When the call_study_day v2 function executes
        Then the <target_variable> in <target_dataframe> is left missing
        And traceability records that the study day could not be derived due to a
partial date
```

The programmer then reviews and extends these scenarios by interacting with the agent—adding cases for missing period reference dates, mixed EPOCH values, or negative study days—rather than writing the feature file from scratch.

INITIAL SCAFFOLDING

The BYOF repository includes a BYOF sub-module generator (scripts/initialize_module.py) that creates a new module skeleton: versioned directory structure, metadata.json, and a Python function scaffold. An agentic IDE can assist by proposing appropriate values for the generator prompts, based on the requirement and existing patterns. For call_study_day (v2), for example:

- subModuleName = "call_study_day", version = "2"
- studyIds = ["SID.123"]
- validSourceDomains = ["AE", "CM", "EX", "LB", "VS"] (or a subset)

Using these inputs, the generator produces a metadata.json file such as:

```
{
  "subModuleName": "call_study_day",
  "version": "2",
  "name": "call_study_day (v2)",
  "displayName": "Study Day Derivation (Period-Specific)",
  "description": "Derive study day using period-specific reference dates with study-
specific handling of partial dates.",
  "studyIds": ["SID.123"],
  "validSourceDomains": ["EX"],
  "validTargetDomains": ["EX"],
  "validSourceFields": ["EXSTDTC"],
  "validTargetFields": ["EXSTDY"],
  "functionType": "derivation"
}
```

This metadata immediately makes call_study_day (v2) discoverable and clearly scoped to the study SID.123, while leaving call_study_day (v1) available for other studies that use the simpler RFSTDTC-based logic

IMPLEMENTATION AND QUALITY ASSURANCE

With the scaffolding in place, the agent can support the implementation and quality assurance:

- **Generate initial Python code** – create or update app.py for call_study_day (v2), using patterns from call_study_day (v1), the new BDD scenarios, and shared SDTM utility packages to:
  - Select the correct reference date per EPOCH or period.
  - Compute study day relative to that date.
  - Apply the study's rules for partial dates (e.g., no imputation, leave missing).
  - Record detailed traceability for derived and non-derived values.
- **Explain and refine behavior** – provide natural-language explanations of how the code satisfies each scenario, and suggest refactoring to improve clarity, reuse, or performance.
- **Extend scenarios and test data** – propose additional BDD scenarios and input datasets to strengthen coverage, such as:
  - Records with missing EPOCH or missing reference dates.

  - Mixed complete and partial dates within the same domain.
  - Negative study days and pre-baseline events, if allowed.
- **Run metadata consistency checks** – identify mismatches between code, BDD scenarios, and metadata.json (for example, fields or domains referenced in tests but not declared in metadata).
- **Suggest linting and formatting fixes** – provide or apply fixes for Pylint, Flake8, and Black findings.
- **Interpret test failures** – read pytest output, map failures back to the relevant Gherkin scenarios, and recommend specific code or scenario changes.

Statistical programmers remain in control: they decide which suggestions to accept, run invoke test, and iterate with the AI assistant's help until all tests pass and the implementation meets the study's requirements.

PUBLICATION

Once tests and quality assurance pass, the AI assistant can help finalize integration into the BYOF framework:

- **Pull request support** – draft a PR summary that clearly describes the purpose and behavior of call_study_day (v2), differences from call_study_day (v1), and the studies and domains affected.
- **Mapping specification support** – propose JSON mapping fragments that use the new function version in the study's mapping specification, for example:

```
{
  "derivations": [
    {
      "Function": "call_study_day_v2_id",
      "FunctionName": "call_study_day (v2)",
      "Version": 2,
      "Sequence": 0
    }
  ]
}
```

- **Post-deployment checks** verifying that the new version is indexed and runs successfully.

Study teams gain the agility to adapt core derivations such as call_study_day to their specific protocol definitions, while governance, traceability, and reuse are preserved through versioning, BDD, metadata, and automated publication workflows.

DISCUSSION

The BYOF framework deployment at Merck provides insights into practical AI integration within regulated statistical programming environments. Our experience highlights both successes and considerations for organizations pursuing similar approaches.

BALANCING COMPETING OBJECTIVES

Traditional SDTM transformation systems optimize for control and consistency, while study teams require flexibility and speed. Our framework demonstrates these objectives need not conflict when proper governance structures exist.

1. **Speed without sacrificing quality**. Study-specific functions move from conception to production deployment in minutes rather than months. Comprehensive templates, automated testing, and AI-guided development ensure code quality matches or exceeds centrally developed alternatives. In internal experiments we observed that providing the AI assistant with curated, organization-specific examples produced substantially higher-quality functions than using generic prompts.
2. **Innovation with governance**. Study-scoped deployment and visibility controls enable experimentation without system-wide risk. Functions prove their value through actual study use before consideration for broader organizational adoption. This creates a natural quality filter and paves the way for community development from study-specific needs rather than anticipatory central development.
3. **AI augmentation with human oversight**. Positioning AI as a code generation assistant rather than autonomous decision-maker preserves traceability and architectural coherence. Statistical programmers report spending less time on implementation mechanics such as debugging centrally governed programs. Also, they invest more time in requirements analysis and system design, both activities yielding greater long-term value.

THE IMPORTANCE OF CURATED CONTEXT

Generic large language models trained on public codebases lack understanding of SDTM standards, organizational conventions, and regulatory requirements. Our approach of providing AI with validated organization-specific examples controls output quality via:

- **Pattern consistency**: AI-generated functions match existing architectural patterns from the repository.
- **Standard compliance**: Metadata structures and testing approaches follow established templates.
- **Regulatory awareness**: Traceability requirements and validation standards are implicitly encoded in examples.

This finding suggests successful AI integration in regulated environments depends more on context curation than model selection. Organizations should invest in building high-quality function registries before deploying AI assistance.

CODE AS BUSINESS OBJECT

Treating functions as configurable business entities rather than hard-coded software components fundamentally changes system adaptability. Traditional approaches require software releases to modify transformation logic; our framework enables authorized users to modify behavior through configuration:

- **Reduced deployment friction**: No software builds or environment promotions required
- **Faster feedback loops**: Test in preview, deploy to production within same day
- **Natural versioning**: Functions evolve alongside study needs with complete audit trails

This architecture proved essential during protocol amendments when transformation logic required rapid updates across multiple domains.

BUSINESS OBJECT GOVERNANCE

Moving to a BYOF framework requires thoughtful approach to change management and business governance. Consider the following:

1. **Learning curve**. Statistical programmers accustomed to centralized function libraries require training on Git workflows, metadata structures, and testing frameworks. For best outcomes, budget sufficient time for team onboarding and learning on the job.
2. **Repository maintenance**. As the function count grows, maintaining consistency and preventing duplication requires ongoing curation. We recommend periodic reviews (quarterly) and automated similarity detection to identify consolidation opportunities.
3. **Human oversight**. While curated context improves AI output quality, responses remain non-deterministic. Human review of AI-generated code is non-negotiable. Our workflow enforces this through pull request requirements and preview environment testing.
4. **Scope creep**. Without clear guidelines, teams may create overly specific functions with limited reusability. We address this through code review feedback emphasizing parameterization and generalization.

IMPLICATIONS FOR THE STATISTICAL PROGRAMMER ROLE

The statistical programming discipline faces increasing complexity: more diverse data sources, faster study timelines, and evolving regulatory expectations. Traditional centralized development models cannot scale to meet these demands. Our framework demonstrates that AI-assisted distributed development, guided by robust governance and architectural principles, offers a sustainable path forward.

Our framework shifts statistical programmer activities from mechanical implementation toward higher-value contributions:

- **Architecture and design**: More time defining optimal function structures and interfaces
- **Quality assessment**: Evaluating AI suggestions and enforcing reusability standards
- **System optimization**: Analyzing usage patterns and refining the function ecosystem

This evolution aligns with broader industry discussions about the future of statistical programming as technology automates routine tasks. The measure of success is not the volume of code generated or the speed of AI responses, but rather the quality, reusability, and maintainability of the function ecosystem that emerges. Early production results suggest this approach enables statistical programmers to spend less time on mechanical tasks and more time architecting solutions to scientific data challenges. By democratizing function development while maintaining compliance standards, we enable the statistical programming community to respond with the agility demands of modern clinical research.

FUTURE DIRECTIONS

The framework's modular architecture supports several enhancement opportunities we plan to evaluate next.

1. **Expanded AI capabilities**. Beyond code generation, AI agents could suggest function reuse opportunities by analyzing study protocols, recommend test case coverage improvements, or identify optimization opportunities across the function registry.
2. **Cross-study analytics**. Aggregated execution logs and reuse patterns could inform function development priorities, identify common customization needs, and quantify the business value of specific functions.
3. **Community contributions**. The governance model could extend beyond single organizations, enabling industry-wide collaboration on common transformation patterns while preserving organization-specific customizations.

CONCLUSION

We have demonstrated that AI-assisted function development can be successfully integrated into regulated SDTM transformation workflows through careful governance and architectural design. The BYOF framework enables statistical programmers to develop and deploy study-specific functions rapidly while maintaining compliance standards and data quality.

The path forward for statistical programming lies not in choosing between centralized control and distributed innovation, but in creating governance structures that enable both. Our framework demonstrates this balance is achievable through study-scoped deployment, comprehensive traceability, and AI positioned as assistant rather than

autonomous agent. As clinical research complexity increases, the ability to adapt transformation logic quickly becomes competitive advantage.

Our BYOF framework validates several principles applicable beyond SDTM transformations:

1. **Curated context transforms AI capability**. Providing AI models with organization-specific validated examples yields dramatically better results than relying on pre-training data outside the clinical research domain.
2. **Business objects enable software flexibility**. Treating functions as configurable business entities rather than hard-coded software components creates adaptive systems that evolve with study needs.
3. **Traceability and transparency enable trust**. Comprehensive logging and version control satisfy regulatory requirements while building confidence in AI-assisted workflows.

ACKNOWLEDGMENTS

The authors would like to thank the BARDS Statistical Programming Leadership and the Management team at Merck & Co., Inc. including Amy Gillespie, Janakiraman Gopinath, and Janet Low for their support. The authors would also like to thank Principal Scientists Donna Hyatt, Proma Debnath, Sangeeta Sama, and Srinivas Malipeddi for their contributions to the presented framework. We extend our gratitude to AWS Professional Services for their invaluable collaboration, which has significantly enhanced the development of the BYOF framework and its implementation.

CONTACT INFORMATION

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

Author Name: Jacques Lanoue
Company: Merck & Co., Inc.
Address: 351 N Sumneytown Pike, North Wales, PA 19454
Email: jacques.lanoue@merck.com
Website: https://jobs.merck.com/us/en/mrl-locations