PharmaSUG 2026 - Paper AI-352

Practical Lessons in AI-Assisted Metadata Conversion: From Database Specs to EDC, SDTM, and P21 — Successes, Pitfalls, and Regulatory Considerations

Jianlin Li and Andy Shen, EverMedix Inc.

ABSTRACT

Metadata-driven transformations (CDASH → SDTM → P21) remain a labor-intensive process across clinical programming teams due to differences in EDC implementations, sponsor conventions, and interpretation of the SDTM Implementation Guide (IG). This paper presents a practitioner's perspective on using AI assistance to accelerate metadata processing, workbook generation, and SDTM quality control, while maintaining traceability and regulatory alignment. Rather than focusing on high-level summaries of CDISC standards, this paper emphasizes practical experiences: dealing with Rave SDS inconsistencies, refining mapping rules, applying derivation logic, generating SDTM domain reference workbooks, and integrating AI-generated metadata into existing QC pipelines. The goal is to demonstrate how AI can reduce manual overhead while reinforcing—not replacing—controlled review and validation expected in regulated submissions.

INTRODUCTION

SDTM mapping continues to require significant programmer judgment, even when high-quality metadata is available. Differences in Rave SDS setups, heterogeneous EDC export formats, and sponsor-specific interpretations of the SDTM IG often limit the effectiveness of fully automated workflows.

Although several commercial AI-assisted SDTM conversion tools now exist, they typically require substantial effort to deploy, integrate, customize, and validate. Their outputs still need significant refinement because AI alone cannot resolve sponsor conventions, therapeutic-area nuances, or ambiguous metadata. As a result, many programming teams find that commercial tools accelerate certain tasks but do not eliminate the need for controlled logic and human oversight.

This paper describes a practical, hybrid in-house approach that combines AI acceleration with deterministic rules and programmer review. Rather than aiming for a fully autonomous SDTM engine, the focus is on using AI where it clearly helps—metadata normalization, SDTM domain workbook extraction, code generation—and preserving human control where clinical or regulatory reasoning is required. Our goal is to share real experiences, successes, and pitfalls to help programming teams adopt AI effectively within existing workflows.

AUTOMATED METADATA TRANSFORMATION FRAMEWORK

In this section we share some experiences in AI assisted metadata mappings including SDTM IG domain table extraction, proprietary database specification to EDC architect specification, and EDC architect or raw data specification to SDTM domain specification.

SDTM IG → SDTM DOMAIN WORKBOOK EXTRACTION

A clean, domain-level representation of the SDTM Implementation Guide (IG) is essential for meta-driven SDTM SAS programming. Although the SDTM IG can be exported to Excel, reshaping it into domain-specific tabs typically requires manually written scripts, which is time-consuming.

The recent availability of large language models (LLMs) enables AI to generate a standardized “SDTM Domain Workbook” directly from the SDTM Implementation Guide (IG). Within minutes, an AI model can produce a workbook that provides a consistent structure across domains (e.g., DM, AE, VS) and captures variable level metadata such as labels, types, lengths, roles, controlled terminology, and value level definitions where applicable. This workbook can serve as an input metadata layer for downstream processes, including raw to SDTM mapping spec, SDTM ddt, or P21 define specification which can then be used in macro-based code generation, and QC; it can also be used as a SDTM standard reference for SAS programmers.

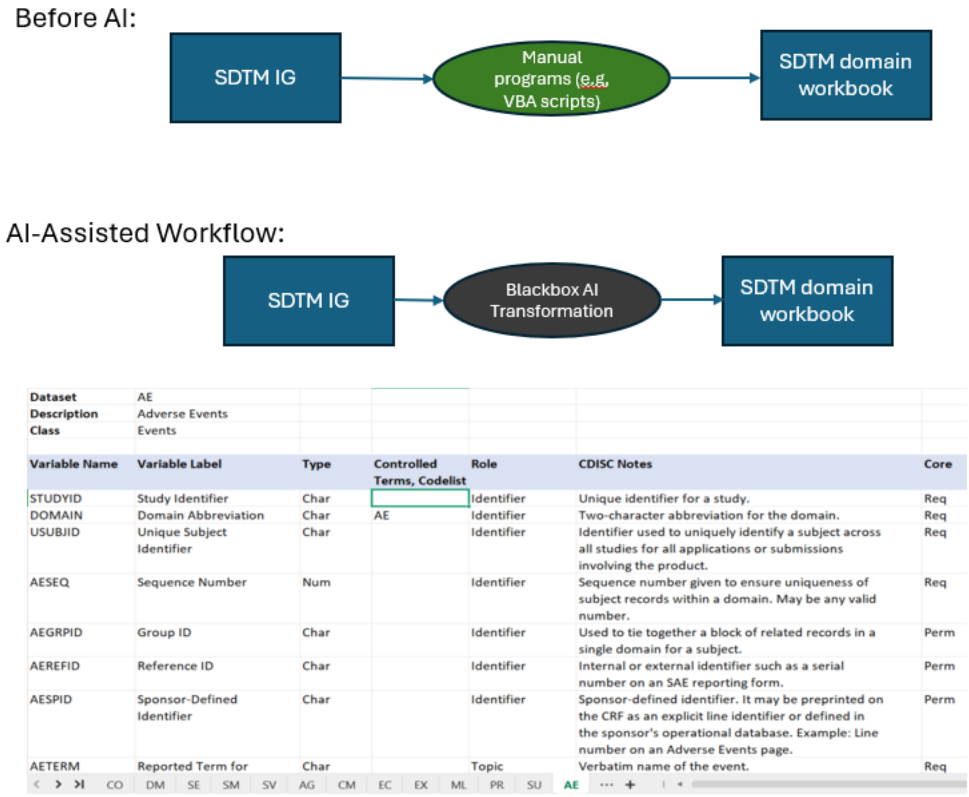

Figure 1 SDTM IG domain workbook extraction

Structural attributes—such as variable ordering, labels, formats, and controlled terminology can be carried over to the down stream metadata specifications. With the down stream metadata in place, SDTM generation becomes a programmatic process: derivation programs focus solely on data logic, while reusable templates and macros apply structural attributes directly from the metadata. QC is simplified because produced datasets can be compared against the metadata layer rather than re validating ad hoc program settings. For example, checks against Core designations (Req, Perm, Exp) can be performed during SDTM dataset creation. Variable labels and other attributes can be pulled automatically, eliminating the need to hard code them in each program. Issues surface earlier in the workflow and no longer depend on waiting for P21 validation step. This reduces manual setup, improves consistency across studies, and accelerates iteration. These capabilities can be implemented through SAS macros

that extract metadata—and even those macros can be initially drafted by AI, with final refinement performed manually.

What We Learned:

For metadata conversions governed by explicit, deterministic rules, AI performs extremely well—often producing accurate results within seconds—so long as the request is clearly and concisely defined. Usually there is no need for refinement, AI works as a black box and generates onetime result, which can be reused.

COMPANY-SPECIFIC DB SPECIFICATION -> EDC ARCHITECTURE SPECIFICATION

A second major transformation step occurs when converting a sponsor's proprietary database or CRF specification into an EDC-ready architecture. In traditional workflows, study builders rarely use a Rave SDS file. Instead, EDC designers manually configure the study **directly in Rave Architect**—creating visits, folders, forms, fields, and dictionaries through the user interface based on the company's DB specification.

This manual build process is labor-intensive. Even for mid-sized studies, designers may spend several days clicking through Architect to recreate the database structure, and the entire process must be repeated whenever the protocol is amended.

Pre-AI: A Python-Based Conversion Framework

To reduce this repetitive manual work, we developed a Python-based converter that transforms the company's proprietary DB specification into a **partial Rave SDS workbook**.
The SDS does not need to be complete; it only needs to contain the key tabs that Architect can import:

- CRFDraft
- Folders / Visits
- Forms
- Fields
- Data Dictionaries and Dictionary Entries

By producing these "pre-Matrices" tabs automatically, the converter enables the study team to:

1. **Upload the SDS into Rave**
2. **Create an initial study shell automatically**
3. **Eliminate the majority of form-by-form and field-by-field manual configuration**

This significantly reduces the initial study build time and gives designers a structured, repeatable baseline aligned with the sponsor's spec.

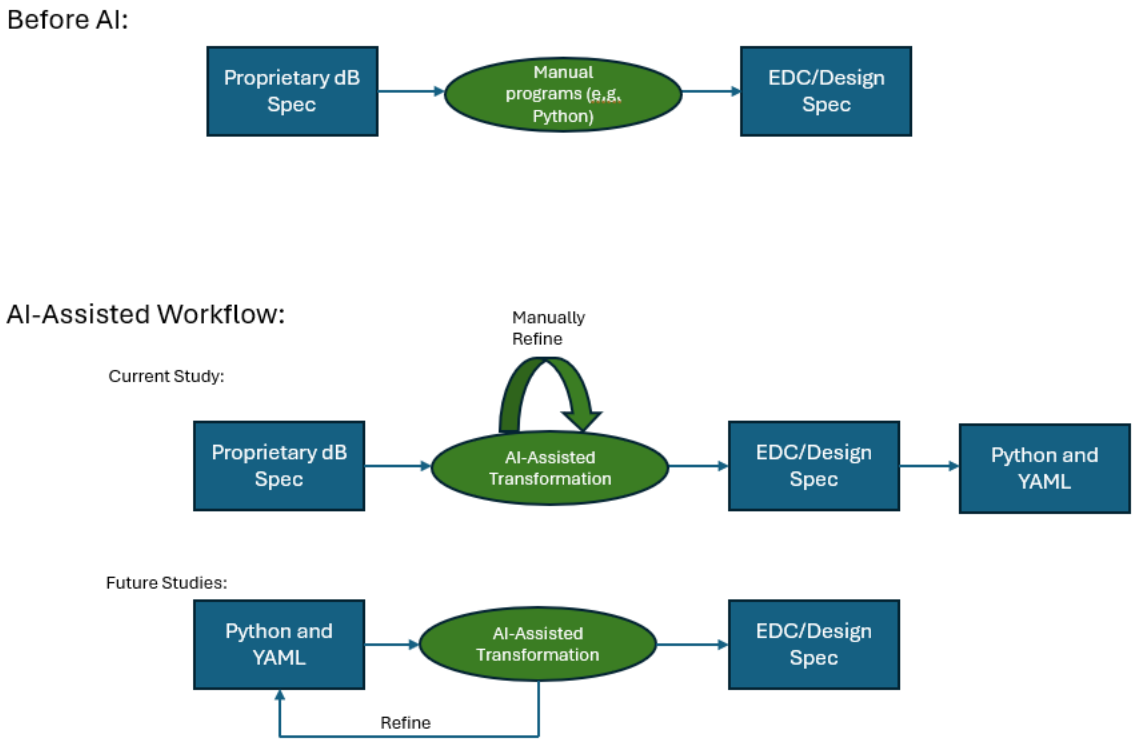

Figure 2 Company-Specific DB Specification to EDC Architect Specification

AI-Augmented Development: Making the Converter Easier to Adapt Across Companies

When AI tools became widely available, we started using them to help maintain and extend our converter. The goal was *not* to have AI generate the entire SDS. Instead, we used AI to speed up the parts that normally take time when a company changes its specification format.

To get dependable results, we learned that the instructions given to AI must be very clear. In our experience, the following points are essential:

- **The output must match the Medidata SDS format exactly.**
  If this is not stated directly, AI may produce a spreadsheet that looks convenient but cannot be loaded into Rave.
  We must explicitly ask AI to follow the official SDS layout and column names.
- **PreText must be handled carefully.**
  PreText controls what users see on the CRF screen.
  Our converter follows a fixed rule to populate PreText:
  eCRF Label → Field Label → Label
  If none exist, we leave it blank to avoid showing the wrong text in Rave.
  When AI is asked to help write code, we must restate this rule so nothing is overwritten.
- **Visit and folder structure must come from the company's Schedule sheet.**
  Each visit in the Schedule tab becomes a **FolderOID** in the SDS.
  This preserves the sponsor's exact visit flow.
  Forms are assigned based on a rule set. If a form appears in many visits, we place it in the first visit.
  This mirrors how most Rave designers would set up a study manually.

To keep the converter reliable across studies, we continue using:

- **Python** as the main engine (stable and repeatable)
- **YAML** to hold study-specific information (sheet names, column locations, etc.)

AI is used simply as a **helper**, not as the final authority. We use it to:

- adjust the converter when a new sponsor has a different specification layout
- help fix issues when column names or formats change
- suggest small improvements when the Rave loader reports errors

The important point is that the **Python/YAML converter remains the official version**, so results are always reproducible.
AI just helps us update the converter faster when something changes.

What We Learned

Through this work, we learned several lessons that may help others who want to automate similar conversions:

- **A partial SDS is already extremely valuable.**
  Even though our converter does not generate every SDS tab, producing the core tabs (Folders, Forms, Fields, Dictionaries) removes a large amount of manual setup in Rave.
  You do not need a "complete" SDS for automation to pay off.
- **Clear rules work better than clever code.**
  We found that having simple, explicit rules—such as how to populate PreText or how to choose the FolderOID—made the converter more reliable than trying to cover every edge case in a single pass.
- **AI is most helpful when the input format changes.**
  The biggest time saver was letting AI help rewrite small parts of the Python logic when a new sponsor used different column names, a different Schedule layout, or a different CRF structure.
  AI helps you adjust faster, but the rules still need to be clear.
- **The converter improves through real use.**
  Each new study brings small differences in how the sponsor formats its specification.
  Instead of redesigning the tool every time, we now treat each study as feedback that strengthens the converter for the next one.
- **AI does not replace human review.**
  We still check the SDS after loading it into Rave Architect.
  But because the structure is already generated, the review is much faster and focuses on correctness, not data entry.

EDC ARCHITECT SPECIFICATION / RAW DATA METADATA → SDTM SPECIFICATIONS

This conversion is where AI struggles within the streamlined framework, because the logic is not fully specified in the IG, varies across sponsor conventions, and often relies on common sense rather than strict, deterministic rules.

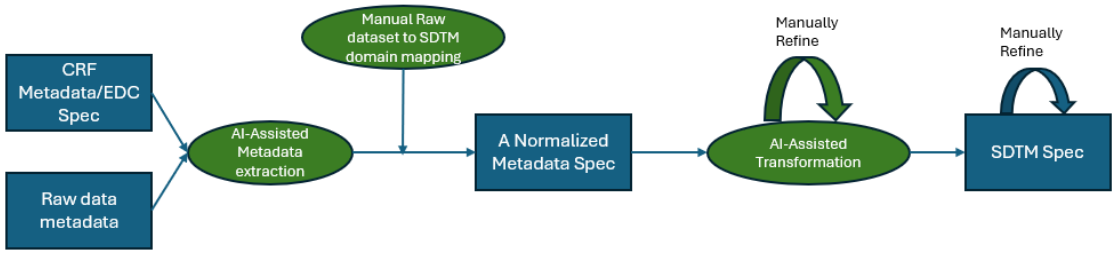

Figure 3 Raw Meta Data to SDTM Spec Conversion

The conversion is composed of two steps.

- Normalize heterogeneous EDC specifications into a standard structure containing six key variables: Dataset, Variable, Label, Type, Length, Format, and Free-text Label.
- Transform the normalized specification into an SDTM specification.

The first step is relatively straightforward: it is essentially a one to one mapping from the EDC specification or raw data specification. The free text label is optional and depends on the source; for example, it may correspond to the pretext field in a Rave EDC specification. AI handles this step well if the mapping instructions are stated clearly.

Dataset	Variable	Label	Type	Length	Format	Freetext
AE	AEACN	Action Taken with study treatment	Num	2		Action Taken with study treatment
AE	AEACNOTH	Other	Num	1		Other

The second step is considerably more challenging because it requires reasoning rather than simple mapping. This is not a straightforward schema conversion; it involves clinical reasoning, regulatory interpretation, and metadata-level judgment.

Key points of our mapping rules include:

- Mappings are driven primarily by semantic meaning, using variable names, labels, and free text descriptions.
- SDTM principles and regulatory expectations guide all mapping decisions.
- The Normalized Dataset offers a commonsense indication of the likely SDTM domain but never overrides clear semantic matches.
- Standard SDTM domains and variables are used whenever possible; no new SDTM domains are ever created.
- SUPPQUAL is used only when no appropriate standard SDTM variable exists within the correct parent domain.
- A normalized variable may map to multiple SDTM domains when the context justifies it.
- Only purely operational or system generated variables may be designated Not Submitted; meaningful clinical variables must map to SDTM or SUPPQUAL.

- Output is organized as one SDTM domain per worksheet, each containing the normalized metadata alongside the mapped SDTM variables.

Even after multiple rounds of refining the AI requirements, the resulting conversion quality remains below the level needed for reliable use.

One improvement is to introduce manual dataset level mapping. For each dataset in the normalized specification, we explicitly list the SDTM domains it maps to. This removes the need for the AI to guess the destination SDTM domains. With this addition, the resulting mappings improved substantially and became useful to a meaningful extent.

Dataset	SDTM domain 1	SDTM domain 2	SDTM domain 3	SDTM domain 4
AE	AE			
EC				
EX	EX	EC		

Normalized Datase	Normalized Variable	Normalized Label	SDTM Doma	SDTM Variable (or QNAM	SDTM Label (or QLABEL)
AE	AEACN	Study Drug Action Taken	AE	AEACN	Action Taken with Study Treatment
AE	AEOUT	Outcome	AE	AEOUT	Outcome of Adverse Event
AE	AEREL	Related to Study Drug	AE	AEREL	Causality

What We Learned:

LLMs perform poorly when tasks require multi-step reasoning, rely on implicit or unstated rules, or involve ambiguous or inconsistently defined metadata.

SUMMARY OF AI-ASSISTED METADATA MAPPING WORKFLOW

The overall workflow that emerged across multiple studies reflects a hybrid approach: use AI to accelerate metadata transformation, but maintain deterministic, controlled logic for all study-critical steps. The workflow consists of three major transformations that together produce a study-ready SDTM specification.

Proprietary Database Specification → EDC Architect Specification (e.g., Rave SDS)

A deterministic Python/YAML converter generates Rave Architect Loader SDS from proprietary metadata. AI assists in code refinement but does not drive the runtime. Key features:

- Strict Medidata SDS compliance
- Deterministic PreText derivation
- Visit/folder extraction from Schedule tab
- Stable, repeatable conversion used across studies.

Normalization of EDC / Raw Data metadata

A unified normalized schema allows downstream SDTM mapping. Normalization uses:

- Dataset
- Variable
- Label

- Type, Length, Format
- Freetext (e.g., mapped from Rave PreText or source)

AI performs this step with high accuracy.

Normalized Specification → SDTM Specification

- The most complex step, governed by semantic interpretation and controlled rules:
- Semantic-first mapping using Variable + Label + Freetext
- Preference for standard SDTM variables
- SUPPQUAL only when needed
- Operational fields → Not Submitted
- Manual override using Dataset to Domain mapping
- Multi-mapping for demographic anchors

PRACTICAL PROCEDURE FOR AI-ASSISTED METADATA CONVERSION

The previous sections described how we automated the transformation of a company-specific specification into a partial Rave SDS. To make the approach accessible to teams who may wish to try similar automation, this section provides a practical, repeatable procedure. Although the example here uses Rave SDS generation, the same workflow can be applied to other metadata conversion tasks.

1. Define the input and output formats clearly

Before using AI or writing code, specify:

- the exact spreadsheet that serves as the **input** (e.g., a proprietary DB or CRF specification),
- the expected **output format** (e.g., a partial Rave SDS with CRFDraft, Folders, Forms, Fields, and Dictionaries),
- and any important rules (e.g., how to derive PreText, FolderOID, FormOID, and schedule-based visits).

Clear requirements lead to predictable conversion behavior.

Domain	eCRF Name	eCRF Label	Variable Name	Code list Name	Variable SAS Label	Length
AE	Adverse Events	**eCCG Guidelines: Adverse Events**				
AE	Adverse Events	Adverse Event Term	AETERM		Adverse Event Term	$200
AE	Adverse Events	Serious?	AESER	NY_NY	Serious?	$1
AE	Adverse Events	CTCAE Toxicity Grade	AETOXGR	AETOXGR	Standard Toxicity Grade	$2
AE	Adverse Events	Start Date	AESTDAT		Start Date	dd-mmm-yyyy
AE	Adverse Events	Ongoing?	AEONGO	NY_NY	Ongoing?	$1

Figure xx Input Company DB Specification Example

2. Ask AI to draft the initial conversion logic

Provide the input format and desired output format to the AI tool.
Explain the rules the converter must follow.
Request an initial version of the conversion logic.

3. Interactively refine the logic with AI

Most conversions require several rounds of refinement.
During this step, we adjust:

- column mappings,
- visit/folder logic,
- PreText population rules,
- dictionary handling,
- and formatting so that Architect accepts the SDS.

This interactive phase lets users tune the converter without manually rewriting large portions of code.

4. Request the final Python and YAML files

Once satisfied with the output generated, request:

- a **Python script** containing the deterministic conversion logic (e.g., python rave_sds_builder.py), and
- a **YAML configuration file** (e.g., rave_sds_config.yml) capturing study-specific settings.

This separation allows the converter to be reused across studies.

5. Set up a Python environment (MiniConda)

Install MiniConda or Anaconda and create a clean environment with:

- Python
- pandas
- openpyxl
- pyyaml
- Jupyter Notebook

This ensures consistent execution across different machines and studies.

6. Run the converter from the Anacona command line

```
(psug26) C:\Users\everm\Documents\psug26-notebooks>python rave_sds_builder.py --config rave_sds_config.yml
OK: wrote C:\Users\everm\Documents\psug26-notebooks\Rave_SDS_CompanyDBspec1-1_preMatrices_fromBuilder.xlsx
CRFDraft: 1 row(s)
Folders: 29 row(s)
Forms: 45 row(s)
Fields: 438 row(s)
```

This should reproduce the SDS output exactly, confirming that the conversion logic is stable outside of the AI environment.

This can be also be accomplished by Shell command in Jupyter Notebook:

```
!python rave_sds_builder.py --config rave_sds_config.yml
```

```
OK: wrote C:\Users\everm\Documents\psug26-notebooks\Rave_SDS_Generated.xlsx
Fields: 249 rows, Forms: 30 rows
```

7. Break the script into Jupyter Notebook cells

For transparency and easier debugging, load the Python script into a Jupyter Notebook and divide it into logical sections such as:

- reading the input,
- building visit/folder structure,
- generating forms and fields,
- producing dictionaries.

E.g., rave_sds_builder.py is converted to debug_rave_sds_builder.ipynb

This allows users to inspect intermediate DataFrames and verify assumptions.

8. Debug interactively and validate output

Run each cell step-by-step in Jupyter to:

- examine intermediate results,
- adjust logic as needed,
- confirm that the output SDS loads properly into Rave Architect.

```
def main() -> int:
    ap = argparse.ArgumentParser(description="Build Rave SDS-style pre-Matrices workbook from CompanyDB spec.")
    ap.add_argument("--config", required=True, help="Path to YAML config.")
    ap.add_argument("--input", default=None, help="Override input Excel path.")
    ap.add_argument("--output", default=None, help="Override output Excel path.")
    args = ap.parse_args()

    crf_df, folders_df, forms_df, fields_df, dd_df, dde_df, dicts_df, ud_df, ude_df, ctx = run_from_config(
        config_path=args.config,
        input_override=args.input,
        output_override=args.output,
    )

    write_workbook(ctx, crf_df, folders_df, forms_df, fields_df, dd_df, dde_df, ud_df, ude_df)
    print(f"OK: wrote {ctx.output_path}")
    print(f"CRFDraft: {len(crf_df)} row(s)")
    print(f"Folders: {len(folders_df)} row(s)")
    print(f"Forms: {len(forms_df)} row(s)")
    print(f"Fields: {len(fields_df)} row(s)")
    print(f"DataDictionaries: {len(dd_df)} row(s), DataDictionaryEntries: {len(dde_df)} row(s)")
    print(f"Dictionaries: {len(dicts_df)} row(s)")
    print(f"UnitDictionaries: {len(ud_df)} row(s), UnitDictionaryEntries: {len(ude_df)} row(s)")
    return 0
```

This step builds trust in the converter and helps teams understand how each component is derived.

9. Update YAML (and Python if needed) for new studies

For future studies:

- adjust only the YAML file to reflect new sheet names or column mappings,
- update the Python script only when the sponsor changes the format significantly.

AI can assist with these updates, but the Python/YAML baseline remains the authoritative version for repeatability.

CONCLUSION

AI-assisted metadata transformation offers meaningful efficiencies across the SDTM preparation process, but it is not a substitute for controlled logic or clinical judgment. Our experience shows that the most effective approach is a hybrid one: use AI to accelerate well-defined, rule-based steps, while maintaining deterministic Python/YAML logic and human oversight for interpretation, derivation, and regulatory review.

Commercial AI-driven SDTM solutions can provide value, but they still demand significant deployment, customization, and validation effort, and their outputs are rarely perfect. For many organizations—especially those with unique metadata conventions or limited resources—an in-house, AI-assisted workflow provides a practical and cost-effective alternative. It allows teams to integrate AI where it reliably improves productivity while retaining full control over traceability, reproducibility, and sponsor-specific standards.

As industry continues exploring AI for clinical data transformation, we believe the greatest benefit will come from approaches that combine AI's speed with the rigor of metadata-driven programming. By sharing these lessons, learned—both successes and pitfalls—we hope to support programming teams seeking to adopt AI responsibly and effectively in their SDTM workflows.