Paper TT07

From Mock to Output: Using Generative AI to Write SAS Code for Standards-Based TFLs

Kala Shivali, Sycamore Informatics, North Carolina, United States
Bhuvana Venkatappa, Sycamore Informatics, Florida, United States

Abstract

Manual authoring of SAS programs for clinical Tables, Figures, and Listings (TFLs) is repetitive and difficult to standardize across studies. This paper describes a metadata-driven framework that uses large language models (LLMs), guided by CDISC SDTM/ADaM structures and Analysis Results Standard (ARS) principles, to generate reviewable SAS code with end-to-end traceability. The design links a study metadata repository, leveraging Analysis Results Metadata (ARM) and TFL mock shells, to a prompt builder. It constrains generation with institutional macro libraries and static lint rules, and enforces determinism through low-variance decoding and fixed seeds. Governance considerations include model versioning, prompt and artifact registries, PHI redaction, and validation documentation aligned with audit expectations. The framework is intended to augment existing programming workflows while maintaining reviewer oversight, quality control, and compliance.

Keywords

CDISC, SDTM, ADaM, SAS, TFL, LLM, Code Generation, Metadata, Traceability, Validation, GxP, Governance

---

1. Introduction

The industry's manual coding process adds substantial overhead to the clinical reporting process, translating to significant wasted hours across the pharmaceutical portfolio. In the pharmaceutical and life sciences industry, the generation of Tables, Figures, and Listings (TFLs) is a foundational, non-negotiable component of statistical reporting for clinical trials. TFLs, as defined by the Clinical Study Report (CSR), serve as the primary evidence for regulatory assessment of a drug's safety and efficacy.

Despite decades of standardization efforts through CDISC SDTM and ADaM models, the industry's TFL workflow remains primarily manual and resource-intensive. The current practice relies heavily on two parallel human efforts: a primary programmer manually interprets the TFL mock shell and writes the SAS code, followed by an independent QC programmer who performs dual-programming validation. This reliance on human

interpretation introduces inconsistencies, extends review cycles, and strains resources due to the sheer volume of required outputs.

Current trends, however, signal a profound shift: organizations are moving toward adopting CDISC Analysis Results Standards (ARS) to mandate prospective planning. This initiative emphasizes the creation of structured metadata—like the Analysis Results Metadata Technical Specification (ARM-TS)—*prior* to code development. This new focus on "planning first" establishes a machine-readable "single source of truth" that defines the analytical output, thereby setting the stage for true automation.

Generative AI offers the technical solution to fully capitalize on this trend. By leveraging Large Language Models (LLMs) guided by these structured ARM-TS and constrained by sponsor-defined rules, AI systems can automate the conversion of specifications into GxP-compliant SAS code, enabling organizations to transition from a manual, retrospective process to a reproducible, context-driven automation model.

2. Challenges in the Traditional TFL Workflow

The manual, programmer-driven process of generating Tables, Figures, and Listings (TFLs) for clinical trial reporting introduces numerous pain points that compromise efficiency, quality, and consistency. These challenges stem primarily from the reliance on human interpretation of structured specifications and manual efforts to ensure regulatory compliance.

Manual Translation

Converting TFL mock shells and Statistical Analysis Plan (SAP) specifications into executable SAS code is an inherently subjective process that depends on human interpretation. This introduces inconsistencies, especially across different studies, therapeutic areas, or programming teams.

Example 1: Conditional Footnotes. A TFL specification for a safety table might require a footnote stating, "Adverse events reported as related to study drug by the investigator". However, the exact ADaM filter logic (e.g., AEREL = 'Y' and AETRTEM = 'Y') might be interpreted and coded slightly differently by two programmers, leading to minor variations in the patient population counts reported.

Example 2: Missing Data Handling. The SAP may define complex rules for handling missing baseline data in a demographics table. One programmer might use a specific SAS function for imputation while another uses a complex IF-THEN/ELSE block in a Data Step. If not perfectly standardized via institutional macros, these variations introduce inconsistencies and complicate code review.

Long Validation Cycles

The industry practice relies on dual programming validation, where a primary programmer authors the code and an independent Quality Control (QC) programmer writes a second, parallel program to verify the output. This process directly doubles the programming effort and significantly extends timelines.

Example 3: Time and Resource Strain. For a typical Phase III study requiring 200-300 TFLs, the dual-programming model requires 400-600 programming artifacts (primary code and QC code). An error found by the QC programmer necessitates debugging and revision by both the primary programmer and the QC programmer, leading to extended timelines and strained resources.

Example 4: Code Divergence. Even when outputs match, the primary and QC code bases often use completely different logic or procedures (e.g., PROC FREQ vs. PROC REPORT), making the review process tedious and increasing the chance that a logic error in both programs, if present, goes undetected.

Limited Reusability

Despite standardization efforts like CDISC, code reuse across different clinical studies remains difficult because subtle variations exist in standards, specifications, and institutional practices.

Example 5: Variable Naming and Filtering. A standard Adverse Event (AE) listing is required in all studies. However, Study A might use the ADaM variable TRT01A for the analysis treatment arm, while Study B uses TRT01PN. Manually adapting the base code for every study involves find-and-replace efforts and manual validation of filter logic, making true reuse challenging.

Example 6: Study-Specific Specifications. While most demographics tables (ADSL) are similar, the inclusion of a study-specific randomization stratification factor or a unique historical control group requires custom code modifications. This forces programmers to maintain study-specific copies of every program, limiting portfolio-level automation.

Traceability and Auditability

Maintaining complete, clear links between the high-level specification (SAP/Mock), the low-level code, the underlying data (ADaM/SDTM), and the final output is laborious but critical for regulatory compliance.

Example 7: Manual Metadata Updates. When a protocol amendment occurs, the programmer must manually update not only the SAS code but also the associated metadata (e.g., Define-XML) and documentation explaining the code changes. This is labor-intensive and creates risk if the documentation lags behind the code.

Example 8: Regulatory Inspection Gap. During an audit, a reviewer might select a single data point in a TFL and demand the full provenance: the code that generated it, the data source, the specific derivation logic, and the corresponding line in the SAP. Manually assembling this documentation for thousands of TFLs is a major audit risk.

Skill Dependency

The manual TFL workflow requires highly specialized skills in SAS programming, CDISC standards, and biostatistical principles.

Example 9: Reliance on Senior Staff. Generating complex TFLs, such as time-to-event analyses (Kaplan-Meier plots) or repeated measures analyses, often requires deep

knowledge of specialized SAS procedures and biostatistics. This heavy reliance on senior programmers creates a staffing bottleneck, as junior staff cannot easily generate production-ready code without extensive supervision.

3. The Generative AI Approach

Generative AI models, such as those based on LLMs, can understand structured inputs (e.g., TFL mock specifications, metadata, and datasets) and produce reproducible, GxP-compliant SAS code. When coupled with a secure orchestration environment, AI can generate, validate, and reproduce outputs under full regulatory compliance. The feasibility of using LLMs for GxP-compliant SAS code generation relies on three technical pillars that constrain the AI's output into a predictable and auditable format:

- Grounded Inputs
- Deterministic Control
- Secure Orchestration

Core Capabilities Required of the LLM

To perform the complex translation from analytical specification to GxP-compliant code, the LLM must possess specific, advanced competencies beyond general code generation:

- **Syntax and Paradigm Specialization:** The LLM must be trained or fine-tuned not just on general programming languages, but specifically on SAS syntax, understanding key procedures like PROC REPORT, PROC FREQ, and PROC GLM, as well as the nuances of SAS Data Step logic.

- **CDISC Contextual Awareness:** The model must be proficient in recognizing and correctly utilizing CDISC ADaM variable names and standard derivation logic (e.g., how to derive analysis populations or treatment-emergent status) provided in the prompt context. This capability ensures the generated code aligns with industry standards.

- **Adherence to Institutional Logic:** The model must be able to recognize and correctly instantiate calls to sponsor-defined institutional macros and functions. By generating calls to these pre-validated logic modules, the framework avoids requiring validation for every line of AI-generated custom code.

The Necessity of Fine-Tuning

Relying on a general-purpose LLM is insufficient and introduces unacceptable regulatory risk. The model must be fine-tuned to specialize its output space:

- **Bias towards GxP Compliance:** Fine-tuning on a curated set of validated SAS code and associated CDISC metadata trains the model to prioritize code that is clean, secure, and easily auditable, conforming to the principles of GxP.

- **Precision over Creativity:** In clinical programming, creativity is a risk; precision is mandatory. Fine-tuning helps the model select the most statistically probable and standards-compliant sequence of code tokens (e.g., using PROC FREQ for a standard count table instead of an elaborate, custom Data Step).

Human-in-the-Loop (HITL) Integration

The generative AI approach is an augmentation tool, not a replacement for human oversight. The system must explicitly incorporate Human-in-the-Loop (HITL) controls to maintain quality and reviewer oversight:

- **Validation Gateway:** Programmers act as the ultimate Validation Gateway. After the LLM generates the code and the static validation checks are complete, a programmer must review the code and the validation log to ensure the analytical intent was correctly translated.

- **Feedback Mechanism:** The framework should allow programmers to efficiently provide feedback, corrections or alternative compliant code, to the system. This human feedback is crucial for future reinforcement learning from human feedback (RLHF) to continuously improve the model's accuracy and adherence to specific, evolving sponsor standards.

4. Architecture Overview

The TFL generation framework is built upon three integrated, interdependent layers designed to maintain GxP compliance, end-to-end traceability, reproducibility, and deterministic output generation.

Data and Metadata Context Layer

This layer serves as the authoritative single source of truth and the boundary for data security. It consolidates all prospective planning metadata, which includes CDISC specifications and Analysis Results Metadata Technical Specification (ARM-TS) analytical plans. This layer provides access to the necessary CDISC-compliant datasets (ADaM, SDTM) along with study metadata defining variables, formats, and derivations. By enforcing the Model Context Protocol (MCP), it ensures that only non-sensitive contextual data is prepared for the LLM, thereby achieving Protected Health Information (PHI) segregation and abstracting the analytical intent from the programming process.

Generative AI Layer

This layer executes the core intelligence function, translating the structured metadata received from the Context Layer into executable code. It utilizes Large Language Models (LLMs) that have been trained or fine-tuned specifically for SAS syntax, statistical concepts, and sponsor-defined standards. The Prompt Builder Agent and the LLM reside here, where algorithmic controls like Fixed Seed injection are applied to enforce output determinism. This layer relies on the Model Context Protocol (MCP) to enable secure access to contextual data during the code generation process and performs a crucial Static Validation check immediately prior to job submission.

Execution and Orchestration Layer

Acting as the Compliance Gateway, this layer manages the secure execution of the generated code and finalizes the audit trail. It provides secure and scalable runtime environments for SAS and R, typically hosting the Containerized SAS Runtime within the Statistical Computing Environment (SCE), which guarantees the environment's immutability. This layer ensures reproducibility through automated versioning, audit trails, and environment snapshots, and is responsible for applying cryptographic hashing (SHA-256) to all output artifacts for non-repudiable audit logs. Its functions support collaboration while rigorously maintaining GxP and regulatory compliance.

Illustrative Example: Generating an ADSL Demographics Table

To clarify the function of each layer, consider the automated generation of an ADSL (Analysis Data Subject Level) Demographics table for a regulatory submission.

Layer	Action	Programmer's Input/Control	Output and Compliance Artifacts
Data & Metadata Context Layer	Prepares the Prompt Payload.	Programmer approves the ARM-TS (e.g., specifying the exact population filter, SAFFL='Y') and the ADSL variable list (e.g., AGE, SEX, RACE) from the Study Metadata Repository (SMR).	Machine-readable Structured Mock Shell derived from ARM-TS. Non-sensitive metadata package for the MCP.
Generative AI Layer	Writes and Validates the Code.	The programmer provides the SPEC_ID. The Prompt Builder Agent injects the Fixed Seed (e.g., seed=789) and whitelisted macro signatures (e.g., %TLF_DEMO).	Generated SAS Code (e.g., a PROC REPORT call with correct WHERE clause and macro parameters). Static Linting Log (Pass/Fail) confirming GxP markers are present.
Execution and Orchestration Layer	Executes, Finalizes, and Logs.	Programmer authorizes the job submission. The SCE uses the validated code in the Containerized SAS Runtime.	Final TFL Output (e.g., RTF table). Cryptographic Hash (SHA-256) applied to the output and log files. Audit Trail Record linking SPEC_ID, code, output, and hash.

5. Technical Details: Algorithmic Constraints for Output Determinism

5.1 Data and Metadata Context Layer

The Data and Metadata Context Layer is the foundational input stage of the Generative AI framework, ensuring the LLM has access to secure, standardized information needed to generate accurate code. It consolidates all analytical planning and data specifications, explicitly shifting the workflow from retrospective reporting to prospective planning, as defined by analysis results standards.

Study Metadata Repository (SMR)

The Study Metadata Repository (SMR) functions as the single source of truth for all study-specific and institutional technical specifications. This repository stores and manages core specifications, including ADaM/SDTM variable definitions, derivations, and institutional standards for all analysis data structures. Critically, the SMR stores the Analysis Results Metadata Technical Specification (ARM-TS), which is generated prior to programming. This document contains the necessary, structured metadata on statistical methods, data sources, and display requirements. Storing the ARM-TS centrally ensures that all subsequent programming is aligned with a centrally managed, single analytical plan, enforcing consistency across all TFLs.

Key elements:

- Core Specifications
- Prospective Analysis Planning (ARM-TS Integration)
- Standards Consistency

Structured Input and Data Model

This component ensures the inputs fed to the Prompt Agent are machine-readable and traceable to the official analysis plan. TFL mock specifications are ingested and stored in a machine-readable, structured format (e.g., JSON or XML), which is now directly derived from the ARM-TS to ensure complete synchronization with the analytical plan. Furthermore, this layer acknowledges the Analysis Results Dataset (ARD) as the final, machine-readable repository for the analysis results data. The ARD captures both the derived results and associated metadata in a standardized format, serving as the definitive, reusable input for generating multiple displays.

Secure Data Control: Enforcing the Model Context Protocol (MCP)

Mechanisms are enforced to ensure that sensitive data is managed securely while context is provided to the generative model. This control provides secure, access-controlled connectivity to the underlying ADaM/SDTM and ARD datasets. The Model Context Protocol (MCP) is key to this enforcement, as it ensures that the generative model only receives non-sensitive metadata for context, while actual patient data (PHI) remains isolated within the secure Statistical Computing Environment (SCE).

5.2 Generative AI and Constraint Mechanisms

5.2.1 Prompt Builder Agent

The Agent is responsible for transforming semi-structured human input into a cohesive, highly constrained instructional payload. The process involves Contextual Assembly, where the Agent ingests the TFL Mock Shell and retrieves all associated ADaM/SDTM Metadata and Institutional Macro Signatures from the Study Metadata Repository, integrating them into the prompt body as explicit technical constraints. To enforce GxP reproducibility, the Agent performs Deterministic Injection by incorporating control parameters into the API payload, specifically a Fixed Seed (e.g., seed=789) and ultra-low Low-Variance Decoding (temperature ≈ 0.0). Additionally, through SAS Function Guardrails, the Agent programmatically restricts the LLM's output space to only include whitelisted SAS procedures and authorized institutional macros, preventing the generation of insecure or non-standard code.

Example: Enforcing Code Security

A key function of the SAS Function Guardrails is to prevent the LLM from generating code that could breach the SCE security or violate GxP principles. For instance, the Prompt Builder Agent ensures that the instruction payload contains an explicit negative constraint prohibiting high-risk commands.

Forbidden Command	Risk Aversion
X command or CALL SYSTEM	Prevents the generated code from executing unauthorized operating system commands, which could compromise the SCE security and data integrity.
FILENAME statement to external path	Restricts the SAS code from attempting to read or write data outside the secure, validated SCE file paths.
Non-whitelisted encryption functions	Ensures all data handling functions rely only on pre-validated, authorized institutional macros or procedures.

Key Elements:

1. **Contextual Assembly:** The Agent ingests the TFL Mock Shell and retrieves all associated ADaM/SDTM Metadata and Institutional Macro Signatures from the Study Metadata Repository.

2. **Deterministic Injection:** The Agent enforces GxP reproducibility by injecting control parameters into the API payload.
3. **SAS Function Guardrails:** The Agent programmatically restricts the LLM's output space to only include whitelisted SAS procedures and authorized institutional macros.

5.2.2 Large Language Model (LLM) and Constrained Generation

The LLM performs the token generation under the controls set by the Agent.

The process begins with Seed-Based Initialization, where the LLM's internal state is initialized by the received Fixed Seed, which guarantees that the code generation process starts from an identical point on every call, forming the basis of deterministic output. Operating under low-variance decoding, the LLM performs Constrained Token Sampling to generate the SAS code token-by-token. This forced sampling ensures that the model selects the most statistically probable and standards-compliant sequence, accurately reflecting the ADaM variable names and filtering logic provided in the prompt. Finally, the LLM performs Macro Instantiation by generating the exact calls to institutional macros (e.g., %ADAE_LIST) with the correct parameter structure defined by the Agent, ensuring the generated code relies on pre-validated logic rather than custom code.

Summarized Steps:

1. Seed-Based Initialization
2. Constrained Token Sampling
3. Macro Instantiation

5.2.3 Static Code Validation

Immediately post-generation, the Prompt Agent executes a rules-based check before releasing the code. This involves Static Linting, where a dedicated engine scans the raw SAS code block for adherence to the sponsor's coding standards, checking indentation, commenting practices, and variable naming conventions. The rules-based check also includes a Security and Compliance Check, which confirms the presence of mandatory GxP markers (e.g., the %let seed=789; statement and the SPEC_ID) and verifies that no forbidden functions or system commands were generated.

5.3 The Execution and Orchestration Layer (SCE)

The SCE is primarily responsible for the execution, reproducibility, and final quality control of the generated code. The SCE's governance framework ensures that all necessary records are created and secured.

5.3.1 Validation & Execution

The execution process incorporates multiple quality controls to ensure compliance. Quality Control (QC) ensures only standards-compliant code is executed, meeting GxP requirements. The code runs in a secure, versioned SCE environment with validated SAS installations via the Containerized SAS Runtime. This guarantees identical execution every time, supporting full auditability and Reproducibility. The final stage involves TFL Output Generation, which executes validated code on ADaM data to produce final TFL outputs (e.g., demographics, AE listings). The SCE ensures Governed Execution by automatically capturing and securing all run records.

5.3.2 Governance & Traceability

This final stage secures the audit trail. Traceability links each program to its corresponding output, supporting regulatory submissions. The SCE finalizes Audit Trails by applying cryptographic hashes (SHA-256) to code and outputs, ensuring integrity and tamper-proof audit logs.

6. Benefits of AI-Driven TFL Generation

Dimension	Traditional Approach	AI-Enabled Approach
Speed	Manual coding and review cycles	Automated code generation and validation
Quality	Subjective interpretation	Consistent adherence to standards
Compliance	Manual traceability	Built-in audit trails, GxP compliance
Reproducibility	Study-specific manual execution	Fully reproducible runs across studies
Scalability	Resource-intensive	Parallel generation across multiple TFLs

7. Future Directions:

The Generative AI framework presented is foundational, but its full potential lies in its strategic evolution toward a self-improving, portfolio-wide reporting platform. These future directions leverage advanced AI methodologies to solidify the system's role as the central engine for GxP-compliant statistical reporting.

Self-Correction and Continuous Improvement via RLHF

The immediate next step involves the rigorous integration of Reinforcement Learning from Human Feedback (RLHF) to enhance code accuracy and adherence to evolving sponsor standards.

The current system relies on programmers as the Validation Gateway, manually auditing the generated code. RLHF formalizes this human oversight into a continuous feedback loop. When a programmer makes a manual correction to AI-generated code, that correction is captured, verified through the quality control (QC) process, and then used to fine-tune the Large Language Model (LLM). This guarantees the model learns directly from validated, GxP-compliant best practices, reducing the need for manual intervention over time and ensuring the AI's internal logic remains aligned with the latest regulatory and statistical requirements.

Strategic Expansion: Portfolio-Level Automation

Moving beyond the single-study context, the framework must expand its cross-study standardization to achieve true portfolio-level automation.

This involves formalizing a Cross-Study Metadata Repository that houses reusable, abstracted specifications for common TFLs (e.g., standard Demographics, Exposure, and Adverse Event tables). By standardizing the inputs across different studies, the system can ensure consistency even when trial protocols vary slightly. This capability is critical for accelerating integrated summaries of safety (ISS) and integrated summaries of efficacy (ISE), where data must be pooled and reported uniformly across multiple trials to support New Drug Applications (NDAs). This strategic expansion shifts the cost savings from task-level efficiency to organization-wide portfolio acceleration.

Advanced Orchestration: Multi-Agent Systems

The evolution into a multi-agent system represents the highest level of automation. Instead of a single LLM generating a single TFL, a fleet of specialized AI agents will be deployed to collaborate on complex, end-to-end study reporting tasks.

- **Task Delegation:** One agent specializes in data harmonization (SDTM-to-ADaM translation), another handles ADaM derivation logic, and a third handles the TFL rendering and formatting.
- **Collaborative Validation:** These agents would communicate via secure APIs, allowing the Validation Agent to audit the Derivation Agent’s output before the TFL Generation Agent begins work. This orchestration ensures that the entire reporting process—from data sourcing to final output—is seamlessly managed within a secure, GxP-compliant workflow. This move towards autonomous collaboration further minimizes human intervention while maintaining rigorous quality control

8. The Paradigm Shift to Auditable Automation

This paper provides an outline of a robust, three-layered Generative AI framework designed to fundamentally shift the clinical trial reporting paradigm from a manual, error-prone effort to an auditable, machine-driven process. The core innovation lies in leveraging regulatory-mandated specifications (specifically the CDISC ARM-TS) as the machine-readable input to automatically generate GxP-compliant SAS code.

The architecture addresses the crippling inefficiencies of dual-programming validation and subjective human interpretation through algorithmic constraint. The Data and Metadata Context Layer establishes the single source of truth and enforces strict PHI segregation using the Model Context Protocol (MCP). The Generative AI Layer ensures code reproducibility by employing Deterministic Injection (Fixed Seed) and SAS Function Guardrails. Finally, the Execution and Orchestration Layer acts as the Compliance Gateway, securing the workflow by running code in a validated Containerized SAS Runtime and applying cryptographic hashing (SHA-256) to all outputs and logs for non-repudiable audit trails. This end-to-end framework transforms the programmer's role from author to auditor, establishing a scalable and consistent approach to TFL generation.

9. Securing Quality and Accelerating Insight

The adoption of this Generative AI framework is not merely an efficiency upgrade; it is a mandatory strategic investment necessary to secure data quality and accelerate the delivery of clinical insight in the face of rising trial complexity.

By automating the creation and validation of statistical programming code, this system directly eliminates the 100% overhead imposed by traditional dual-programming and subjective manual translation. The framework ensures that every line of code generated, and every data point reported, is explicitly traceable back to the approved ARM-TS specification, thereby minimizing regulatory inspection risk.

The ultimate value delivered is twofold: first, achieving unprecedented, deterministic reproducibility across entire study portfolios ; and second, freeing specialized biostatistics and programming teams to focus on complex scientific questions, such as estimands and sensitivity analysis, rather than repetitive code generation.

The path forward requires robust governance, particularly the integration of Reinforcement Learning from Human Feedback (RLHF) to continually refine the model's compliance and accuracy. By fully embracing this automated, GxP-compliant workflow, the pharmaceutical industry can finally transition its clinical reporting process into a truly modernized, scalable, and error-resistant component of drug development. This evolution secures quality at every step, allowing organizations to achieve reproducibility, compliance, and speed simultaneously.

10. Conclusion

Generative AI represents a transformative step in clinical programming. By automating the transition from mock TFL specifications to validated SAS outputs, organizations can achieve reproducibility, compliance, and speed. Combined with Secure SCE, this approach allows biostatistics teams to focus on scientific insight rather than manual programming, while maintaining GxP and regulatory compliance at every step.

Appendix A: List Of Abbreviations

Abbreviation	Full Form
ADaM	Analysis Data Model
ADAE	Analysis Data for Adverse Events
ADSL	Analysis Data Subject Level
API	Application Programming Interface
ARD	Analysis Results Dataset
ARM-TS	Analysis Results Metadata Technical Specification
ARS	Analysis Results Standard
CDISC	Clinical Data Interchange Standards Consortium
CSR	Clinical Study Report
GenAI	Generative Artificial Intelligence
GxP	Good Practices
HITL	Human-in-the-Loop
LLM	Large Language Model
PHI	Protected Health Information
QC	Quality Control
RLHF	Reinforcement Learning from Human Feedback
SAS	Statistical Analysis System
SCE	Statistical Computing Environment
SDTM	Study Data Tabulation Model
SMR	Study Metadata Repository
SOP	Standard Operating Procedure
SPEC_ID	Specification Identifier
TFL	Table, Figure, and Listing

TMF	Trial Master File

References

1. CDISC (Clinical Data Interchange Standards Consortium). "Analysis Results Standard (ARS) v1." GitHub. Last modified [Current Year]. Accessed October 23, 2025. https://github.com/cdisc-org/analysis-results-standard/tree/main/workfiles/examples/ARS%20v1.
2. CDISC (Clinical Data Interchange Standards Consortium). ADaM Implementation Guide (ADaMIG) Version 1.3. Raleigh, NC: CDISC, 2020.
3. https://www.lexjansen.com/pharmasug/2023/MM/PharmaSUG-2023-MM-327.pdf.

CONTACT INFORMATION

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

Author Name: Kala Shivali, Bhuvana Venkatappa
Company: Sycamore Informatics
Address: North Carolina, United States
Email: kala.shivali@sycamoreinformatics.com, bhuvana.venkatappa@sycamoreinformatics.com
Website: www.sycamoreinformatics.com