Paper ML12

Utilizing AI for Automated Conversion of ADaM™ Specifications to R Code

Alice Ehmann, GSK, Collegeville, PA, USA
Kjersten Offenbecker, GSK, Collegeville, PA, USA

ABSTRACT

The advent of artificial intelligence (AI) in clinical data sciences is paving the way for enhanced efficiency and accuracy in statistical programming. This paper delves into the application of AI technologies in automating the conversion of ADaM™ (Analysis Data Model) specifications into R code, a vital process in preparing analysis-ready datasets in clinical trials.

Our approach leverages AI models, equipped with natural language processing (NLP) and machine learning capabilities to interpret detailed ADaM™ specifications and semi-automate their conversion into R scripts. This not only accelerates the development of analysis datasets but also fosters greater consistency across projects.

A key feature of this system is its interactive, iterative process that involves data professionals working alongside AI-generated code. While AI can significantly reduce the initial manual coding effort by generating a robust framework, human expertise remains crucial for refining and validating the output to ensure it functions as intended. The AI produces code that serves as a strong foundation yet allows flexibility for human intervention to tailor and enhance the scripts based on specific project requirements.

Throughout the paper, we provide case studies demonstrating the efficacy of this AI-driven approach in producing reliable ADaM™ datasets. Our results reveal notable improvements in the speed of dataset preparation, ultimately facilitating more timely data analysis. Moreover, this collaborative AI-human model boosts productivity by enabling statisticians and programmers to focus on more complex analytical tasks and decision-making processes.

While the pursuit of fully error-free automated R code remains aspirational, this study highlights the substantial progress achievable with current AI technologies. Our findings underscore the symbiotic relationship between AI and human expertise, shaping a promising future for statistical programming in clinical research.

INTRODUCTION

ADaM™ (Analysis Data Model) datasets are the cornerstone of analysis ready data in clinical trials. For decades, programmers have translated ADaM™ specifications into production code (historically in SAS) to produce validated datasets for analysis and regulatory submission. As the pharmaceutical industry transitions more programming to open-source technology such as R, we face the practical challenge of enabling programmers with little or no R experience to produce high quality ADaM™ code quickly and consistently. Our existing ADaM™ specifications (largely language-agnostic) and extensive SAS codebase represent valuable accelerators in the adoption of R programming.

Artificial intelligence (AI), and in particular recent advances in large language models (LLMs) and natural language processing (NLP), offers a promising bridge from the SAS to R transition: these models can interpret specification text and recognize existing code patterns to generate initial, human-readable R code scaffolding. In other domains, code generation models have already accelerated software development, produced unit tests and assisted with refactoring; applying similar capabilities to clinical data programming creates an opportunity to translate ADaM™ specifications into executable R templates automatically. This approach can reduce repetitive manual work while preserving critical human oversight for domain-specific decisions, enabling programmers with limited to no R experience to produce consistent, auditable code without changing established specifications.

However, clinical programming differs from general-purpose coding in ways that raise unique challenges and constraints for AI assistance. ADaM™ derivations often include nuanced statistical rules, regulatory expectations and dataset-specific conventions. Any AI-generated code must therefore be auditable, reproducible and verifiable against the source specifications.

Furthermore, regulatory environments demand explicit traceability from specifications to implementation, making provenance of AI outputs and reviewer decisions a critical consideration for adoption in regulated settings.

In this paper we describe a practical, human-in-the-loop system for semi-automated conversion of ADaM™ specifications into R code. The system combines NLP-driven parsing of specification text, domain-aware code templates and controlled prompt engineering to produce R scripts that implement variable derivations and dataset assembly. Importantly, the workflow embeds human review and iterative refinement as mandatory steps to ensure correctness and regulatory compliance. We present synthetic case studies that mirror common ADaM™ tasks, report evaluation metrics for development speed and code quality and discuss governance, validation and limitations relevant to adoption in clinical programming organizations.

Our objectives are threefold: (1) demonstrate that AI can reliably produce useful initial code scaffolding for ADaM™ tasks, (2) quantify the productivity gains and types of edits typically require during human review and (3) outline practical governance and validation practices to ensure safe, auditable use of AI and regulated clinical data programming. By focusing on a programmatic balance between automation and human oversight, we aim to provide a road map for teams considering AI-assisted workflows for ADaM™ development.

METHODS

To evaluate and demonstrate AI-assisted ADaM™ code generation we implemented a three-layered workflow that mirrors typical programming practices while adding AI capabilities where they provide the greatest value.

The Methods section below describes:

1. ***ADaM™ Specification Creation***
   How ADaM™ specifications are created

2. ***Prompt Generation:***
   How the AI prompt is created leveraging standards (CDISC, company, etc.) as well as programming preferences

3. ***The Generation Engine:***
   The AI components, prompt engineering and template library used to produce R code scaffolding and

4. ***Human-in-the-Loop Review:***
   The interactive processes, validation checks and governance controls ensure outputs are auditable and suitable for regulatory use.

For each component we present design decisions, key implementation details and examples of how they worked together in practice.

ADaM™ Specification Creation

This step is the most critical stage of the process. This stage requires domain knowledge of CDISC, therapeutic area, as well as an understanding of the database (rawdata) design. To effectively use AI for code generation, the programmer should concentrate on ensuring specifications are accurate, clear, and concise.

In this presentation, specifications are created and formatted according to Pinnacle 21 (P21™). However, this process could be applied to any specification format. The programmer develops the specifications in a language agnostic manner, meaning any programming language specific functions or syntax should be described in a human readable manner (which can include mathematical or logical notations). In addition, the programmer should assume the AI has no previous experience programming ADaM™s. Anything the programmer knows from experience must be detailed in the specification (or included in the final prompt).

Effective AI-driven code generation depends on well-defined specifications. Investing effort in creating clear, complete specifications significantly reduces the need for deep language specific programming expertise.

High quality, unambiguous specifications are essential to the success of AI-assisted code generation. Because the input processing step relies exclusively on dataset spec texts, the AI can only reflect the intent that is clearly encoded in the specification. Consequently, the fidelity and usefulness of the AI-generated code are directly proportional to the clarity, completeness and consistency of the spec: well-written derivations produce accurate, actionable scaffolds with minimal manual edits, while vague or underspecified instructions require more human intervention and carry a greater risk of incorrect assumptions.

Creating a robust ADaM specification is therefore where much of the substance work should occur. Programmers, statisticians and spec authors should invest time to:

- Use precise language or operations (e.g. "last non-missing value prior to Day 1" rather than "baseline" when the meeting differs).
- Explicitly list those source variables and units and specify any required transformation, unit conversions, or decimal precision.
- Define edge cases (i.e. efficacy missing value imputation) and missing-value rules (for example, whether partial dates should be imputed and how).
- Include clear predecessor references
- Define any assumptions about joins or keys (e.g., USUBJID plus VISITNUM).

We recommend treating the spec authoring step as a collaborative, reviewable activity with the same rigor as code review. including version control, peer review and sign-off checkpoints. Doing so improves the quality of the AI scaffolds, reduces downstream programming and review effort and strengthens auditability for regulatory submissions.

Prompt Generation

Prompt generation should follow a standardized template that can be refined as needed. Using templates ensures consistency, reduces the burden on programmers to craft effective queries, and prevents frustration that often arises from poorly structured prompts. The prompt should introduce any programming language requirements or restrictions, keeping the specification language agnostic.

Prompt Structure:

1. Brief description of AI's role and the desired output:

> In R, I would like you to generate code in the R language to create a dataframe called **xxxx** using the specifications that are provided in 4 .txt tab delimited file format along with additional instructions I am providing. The 4 .txt files are variables.txt, codelists.txt, comments.txt and methods.txt. The column labels are in the first row of each file. The variables.txt should be filtered to only rows where the Dataset equals **xxxx**.
>
> Here is a brief description of each of the 4 .txt files:
>
> - variables.txt: You should only use rows in this sheet where the column Dataset equals **xxxx**. This file contains all variable definitions for the final **xxxx** dataset to be created. For example, all rows in the variables table for the **xxxx** dataset should be included in the final **xxxx** dataset. The label, data type, length and origin of each variable are also provided. Ensure the final variable is the same data type defined in the specification.
> - codelists.txt: This contains code (Term) and decode (Decoded Value) pairs identified with an ID. The codelists may be referenced in the variable definition or derivation.
> - comments.txt: This contains a derivation to compute a variable in the Description column. Each comment is identified with an ID.
> - methods.txt: Like the comments, each row contains a computation in the Description identified with an ID.

2. Describe input data: Provide details including location and packages to utilize when reading data

You are given SDTM datasets **sdtm1, sdtmsupp1, sdtm2, sdtm3, sdtmsupp3, sdtm4, sdtm*n*, sdtmsupp*n***, which can be found in the directory referred to as “sdtm”. These files are stored as SAS transport files and have the extension.xpt. For example, you read in SDTM datasets like this:
dm <- haven::read_xpt(paste(sdtm, "dm.xpt", "/")). The **sdtm1, sdtm2, sdtm*n*** domains have SUPP domains that can be joined to the parent domain like dm <- metatools::combine_supp(dm, suppdm).

You are also given ADaM datasets **yyyy, zzzz** which can be found in the directory referred to as “adam”. These files are stored as SAS transport files and have the extension.xpt. For example, you read in ADaM datasets like this:
adsl <- haven::read_xpt(paste(adam, "adsl.xpt", "/")).

You should also run the function admiral::convert_blanks_to_na() immediately after reading in any .xpt file. For example, dm <- haven::read_xpt(paste(sdtm, "dm.xpt", "/")) %>%
admiral::convert_blanks_to_na().

3. Describe the specifications: Detail what specification tables are provided and how each table is used. Additionally, provide information on how the metadata tables relate to each other if not described in the specification. This description of the specifications should be standard across all ADaM™s.

> For variables where Origin = Predecessor, there is a column called “Predecessor” which will tell you from which SDTM or ADaM dataset you can just pull the variable from. For example, if the Variable we are trying to derive is STUDYID, the origin is Predecessor, and the Predecessor column states “DM.STUDYID”, the STUDYID is equal to the STUDYID variable stored in the DM dataset.
>
> For variables where Origin = Assigned, you would look at the "comments" table to find the description of how to set the variable. For example, if the Variable is AAGEU, and the origin is "Assigned", you then look at the Comments column to find the name of the comment ID which is “COM.AAGEU”. You then go to the comments table and find the row where the ID column equals COM.AAGEU. The description column in the comments table will have instructions on how to derive the column. For example, if the description states to "Set to Years", this means that the variable AAGEU will be set to “Years”.
>
> For variables where Origin = Derived, you look at the methods column to find out the name of the Method. You then go to the "Methods" table and find the ID of the method. In the methods table, the description column will describe how to derive the variable. For example, if the Variable is AGE, in the Methods table you will find a description such as “Years between BRTHDT and TRTSDT rounded to 1 decimal place". You can see that in a case like this the specification assumes that you already have the TRTSDT and BRTHDT variables available. This means in our task the variables are sometimes dependent on each other being built already and therefore it is important to establish a starting point for yourself.
>
> Additional metadata descriptions can be developed for the prompt including how to handle codelists, value level metadata, dictionaries, variable level metadata (mandatory, significant digits, etc.) etc. can be developed.
>
> For variables where Mandatory = Yes, the value cannot be missing. Please include a check that the column is populated when Mandatory = Yes.
>
> Please ensure when deriving a variable, the variable type of the variable equals the type defined for the variable.
>
> CDISC naming conventions and rules could also be incorporated into the prompt. For example, it may be advantageous to let the AI know this:
>
> Variables ending in "DT" are numeric date variables. Variables ending in "DTM" are numeric datetime variables.

Company standard rules could be incorporated as well to ensure the AI is conforming.

4. Code Organization: Describe how code provided by AI should be structured and what packages to use. This will assist with AI producing similar code across ADaM™s and ensure consistency across the code base. This can also force the AI to use desired packages, pandas, procedures, etc.

> The code generated should be structured as follows. First, please source the script setup.R. This will define references for the data locations. Next, the following packages can be used and should be included in library statements: tidyverse, admiral, metacore, metatools, haven and xportr. After the library statements, please read all data using haven::read_xpt. The final dataframe can be written out to the adam directory as a .xpt file using the xportr package. The variable labels, lengths can be found in the variables.txt data.

5. Describe any additional joins/derivations needing special handling. Some of these items may need customization for a dataset but most could be generalized for a standard prompt.

- Joins: It is critical the AI is aware of any joins. For example, when creating an ADaM such as ADAE, the programmer will utilize ADSL. A programmer is aware that ADSL has one record per subject and can be joined to any ADaM using the key STUDYID, USUBJID. It is critical the AI is aware of this rule. In the ADAE prompt, a statement such as:
  For variables that are derived from the dataset ADSL, ADSL has a unique key of STUDYID, USUBJID and can be joined to other datasets by this key.
  may be added to the prompt.
- Instructing AI to use particular package: If particular packages should be used for deriving fields, this should be included in the prompt. For example, if you would like all code to convert –DTC date to numeric –DT dates using the admiral function derive_vars_dt, this should be included in the prompt
  For derivations that require a character date to be converted to a numeric date the function admiral::derived_vars_dt should be used.
- Additional instructions for derived records: Derived records (e.g DTYPE = AVERAGE) have proven challenging for AI to handle due to incomplete specifications and variable dependencies.

The Generation Engine

The Generation Engine transforms the canonical structure representation produced by the Prompt Generation into an initial, human-readable piece of code. Its goal is to produce code that

a. implements the derivation described in the P21™ specs as conservatively as possible,
b. follows agreed coding standards and idioms,
c. is traceable to the original spec and
d. surfaces any assumptions or unresolved ambiguities to the programmer.

The Generation Engine allows the AI model (LLM) to consume the AI prompt as well as any additional input files (e.g. P21™ specification) to meet these goals.

By combining templates and controlled prompt engineering with an LLM, the Generation Engine produces safe, auditable code that accelerates programming while ensuring human oversight.

Human-in-the-Loop Review (HITL)

Human oversight is essential. The AI Generation Engine provides conservative, well-documented scaffolds but these outputs are starting points not final deliverables. The human in the loop HITL process is mandatory in our workflow: AI produces conservative, well-documented code and programmers (often the original spec authors or experienced ADaM™ programmers) inspect, test and refine the code before it becomes part of the codebase. Programmers remain responsible for interpreting the specifications, testing the code against data, resolving ambiguities and making judgment calls where domain expertise is required. Because LLMs can produce plausible looking but incorrect logic, hallucinate unstated assumptions or mishandle edge cases, the HITL process is a mandatory safety and quality control step; it ensures correctness, preserves regulatory traceability and captures the rationale for any deviations from the original specification. In short, AI accelerates and augments programming work but human expertise, review and sign-off are nonnegotiable.

Ownership and roles

- Production programmer:
  Both initiates code generation (by ingesting the P21™ spec into the Generation Engine and submitting or approving prompts) and is the single responsible owner who implements, tests and finalizes the resulting R code for the ADaM™ dataset. This end-to-end ownership; from scaffold generation through implementation and logging; ensures clear accountability for interpreting the specs, applying study conventions and producing the final validated implementation.
- QC programmer:
  Performs an independent quality control of all ADaM™ datasets. The QC programmer is explicitly separate from the production programmer and must not reuse the production programmer's scaffold, code artifacts or prompt/response history prior to performing QC. This separation preserves independence and reduces confirmation bias in the QC process.
- Subject matter experts:
  Statisticians and spec authors are available to resolve unspecified rules or confirm interpretations of analysis intent when escalations are required.

The Production Programmer typically initiates specification creation and carries the end-to-end responsibility for delivering production ADaM™ code so in this process below we will use the Production Programmer as the main user. However, the same checklist and practices apply to any user of the generation engineer, including a QC programmer performing independent verification or an SME prototyping code; with the caveat that the QC programmer must maintain formal independence from the Production Programmer's artifacts when performing QC. In short, the steps outline a general, repeatable approach for anyone refining AI-generated scaffolds into validated R code, while governance rules determine who may reuse which artifacts in the independent QC context.

Scaffold Validation and Implementation Process

- Import and run the code to identify high-level issues (syntax errors, use of unapproved packages, runtime errors in the log).
- Resolve immediate runtime and package issues to the code
- Iteratively validate piece by piece:
  - Execute and verify each logical block (e.g., run the ASTDT derivation and confirm it is populated as expected).
  - Fix or refined the code for that block as needed, documenting changes.
- Run the full pipeline and validate the data set histologically (counts, joins, derived variable distributions, metadata).
- If a derivation requires a spec change or scaffold update:
  - Update the P21™ spec or prepare clarified instructions,
  - Regenerate the code (entirely or for the affected fragment) or manually replace the block in the code,
  - Rerun the relevant tests and full pipeline.
- Once satisfied with the implementation, commit the finalized code, update the audit log and change log, and hand off the dataset to the next person in the chain.

This process is inherently iterative: the AI is intended to get the code close to a correct implementation, but it is the Production Programmer's iterative cycle of run → inspect → refine that ensures the final results are correct. Often a few short edit-and-test cycles are required to resolve edge cases, clarify ambiguous spec language or tighten performance and error handling. Where necessary the programmer may update the P21™ specs or request a targeted regeneration of a scaffold fragment, retest the change and repeat until the outputs fully reflect the intent of the specifications. This human-led refinement is the clinical quality control that turns an AI-generated starting point into a validated, auditable atom implementation.

Best Practices and Governance

- Trade HITL as a formal step in the study design life cycle for ADaM™ code: require sign-off and version control for any AI-generated artifact.
- Ensure reviewers are trained to recognize common LLM failure modes (hallucinations, over generalization, incorrect assumptions).
- Maintain a living FAQ or prompt guidance library documenting prompt phrasing and common fixes for repeated issues.
- Limit model permissions (both technical and process) so that only approved content is used in prompts and generated code is always reviewed before merging.

Limitations and Human Factors

- Review quality depends on reviewer expertise; junior programmers may need additional oversight when working with AI outputs.
- The HITL process introduces its own overhead; however, properly measured, the combination of AI scaffold generation plus HITL review should reduce net programmer effort by shifting repetitive work to the model and reserving human time for higher-value decisions.

CONCLUSION

AI-assisted generation of R scaffolds from Pinnacle 21 ADaM™ specifications can materially accelerate ADaM™ development while preserving the traceability and auditability required for regulatory submissions. By using a spec-first approach, a Generation Engine that combines domain aware templates with controlled prompt engineering and a mandatory human-in-the-loop review, teams can obtain high-quality initial code that reduces repetitive work and standardizes implementation patterns across studies.

Crucially, the Production Programmer remains responsible for converting this scaffold into validated, production-ready code. The iterative cycle of run → inspect → refine is where domain expertise migrates mitigates model limitations (such as hallucinations or under specified logic) and ensures outputs faithfully realize the specifications. Independent QC by a separate QC programmer, performed without reuse of the production artifacts, provides a further, essential layer of verification and preserves confidence for downstream analysis and regulatory review period

For practical adoption we recommend investing in high quality P21™ specifications (the better the spec the better the scaffold); maintain a curated, versioned template library and prompt guidance; enforce prompt/response and artifact audit logging; and train programmers and QC staff on model failure modes and prompt design. These governance and workflow measures help realize productivity gains while limiting risk.

Future work should evaluate the approach across a broader set of real studies, expand template coverage for more complex derivations and explore controlled model grounding with domain knowledge to reduce ambiguity and improve first-pass correctness. With appropriate controls and human oversight, AI is a powerful tool to augment clinical programming productivity, not a replacement for expert judgment.

CONTACT INFORMATION

Alice Ehmann
Associate Director
Infectious Disease - Clinical Programming (US)
RD Projects Clinical Platforms & Sciences
GSK
Alice.x.ehmann@gsk.com
gsk.com

Kjersten Offenbecker
Programming Leader
Infectious Disease - Clinical Programming (US)
RD Projects Clinical Platforms & Sciences
GSK
Kjersten.x.offenbecker@gsk.com
gsk.com