Paper ML19

SpecToSAS: AI-Powered SAS Code Generation from Study Specs

Gargee Vaidya, AstraZeneca, Toronto, Canada
Vitaliya Lysenko, AstraZeneca, Toronto, Canada
Lluis Gimenez, AstraZeneca, Barcelona, Spain

ABSTRACT

Creating regulatory-compliant ADaM (Analysis Data Model) datasets for clinical trials requires translating complex specifications into SAS programs. These specification files often include hundreds of variables with complex, multi-step derivation rules and dataset-specific logic. Manually translating them is time-consuming, repetitive, prone to inconsistency and error, and can lead to code that is difficult to read and maintain. This increases the risk of misinterpretation and complicates collaborative reviews, diverting statistical programmers from higher-value tasks.

To address this challenge, we introduce SpecToSAS, a hybrid AI-assisted system that automates the translation of Excel-based specifications into ready-to-review SAS code. By combining deterministic Python automation with a multi-agent AI architecture, SpecToSAS handles both straightforward variable mappings and complex derivation logic while maintaining transparency and human oversight. The system incorporates retrieval-augmented generation (RAG) to leverage existing institutional code repositories, ensuring generated programs align with established standards and reusable components.

Early implementation results across 14 safety ADAM datasets from four oncology studies show that 75-95% of variables were correctly programmed, with code generation taking 5-15 minutes and human review requiring 25-55 minutes for minor fixes to achieve 100% accuracy. While the generated code still requires validation, SpecToSAS significantly reduces the initial coding burden, allowing programmers to focus on quality review and complex problem-solving rather than repetitive tasks.

INTRODUCTION

In the pharmaceutical industry, clinical trials test investigational drugs with patients to evaluate their safety and efficacy. Throughout the trial, researchers meticulously track patient outcomes, adverse events, laboratory results, and other critical measurements. All this information is collected in Case Report Forms (CRFs) and delivered as raw data. Before this data can be analyzed to assess how well the drug performs, it must be standardized and transformed into structured formats that meet regulatory requirements [1]. This is where SDTM and ADaM datasets become essential. Creating these clinical analysis datasets using SAS programming language is both critical and complex, forming the backbone of regulatory submissions. SDTM standardizes the collected clinical trial data into a consistent format, while ADaM transforms it into analysis-ready datasets for statistical evaluation [2] [3].

The traditional process follows a two-stage pipeline, both requiring manual coding. In the first stage, statistical programmers receive SDTM specification files (typically complex Excel spreadsheets) and write SAS code to generate SDTM datasets. This involves mapping raw clinical data to standardized CDISC domains like Demographics, Adverse Events, and Laboratory Results. Once the SDTM datasets are validated, the second stage begins. Programmers receive separate ADaM specification files and write more SAS code to derive ADaM datasets from the existing SDTM data.

This workflow is highly complex and time-consuming. Programmers spend hours interpreting hundreds of variables spread across multiple specification documents, understanding complex derivation logics, translating it into working SAS code, and verifying that every step complies with strict regulatory standards [4]. Moreover, when study requirements shift or specifications are updated mid-project, keeping everything accurate and compliant becomes even more difficult because programmers will have to reuse SAS code written by colleagues. Understanding someone else's code is rarely straightforward and can take considerable time, especially when coding styles and documentation practices vary. This leads to inconsistencies as each programmer adapts the code to their needs. After several iterations of different programmers modifying the same program, the code becomes increasingly difficult to read, update, and maintain [5].

We present SpecToSAS, a hybrid AI-assisted system developed to address these challenges by automating the translation of Excel-based specifications into ready-to-review SAS code. Instead of relying on labor-intensive, error-prone manual coding, SpecToSAS receives as input the Excel specifications and can generate SAS code for both SDTM and ADaM datasets efficiently and consistently. While the system is designed to handle both dataset types, current validation has focused on ADaM generation, with SDTM capabilities yet to be formally tested. The tool produces ready-to-review SAS code that still requires validation by programmers to ensure accuracy and compliance, but dramatically reduces the initial coding burden.

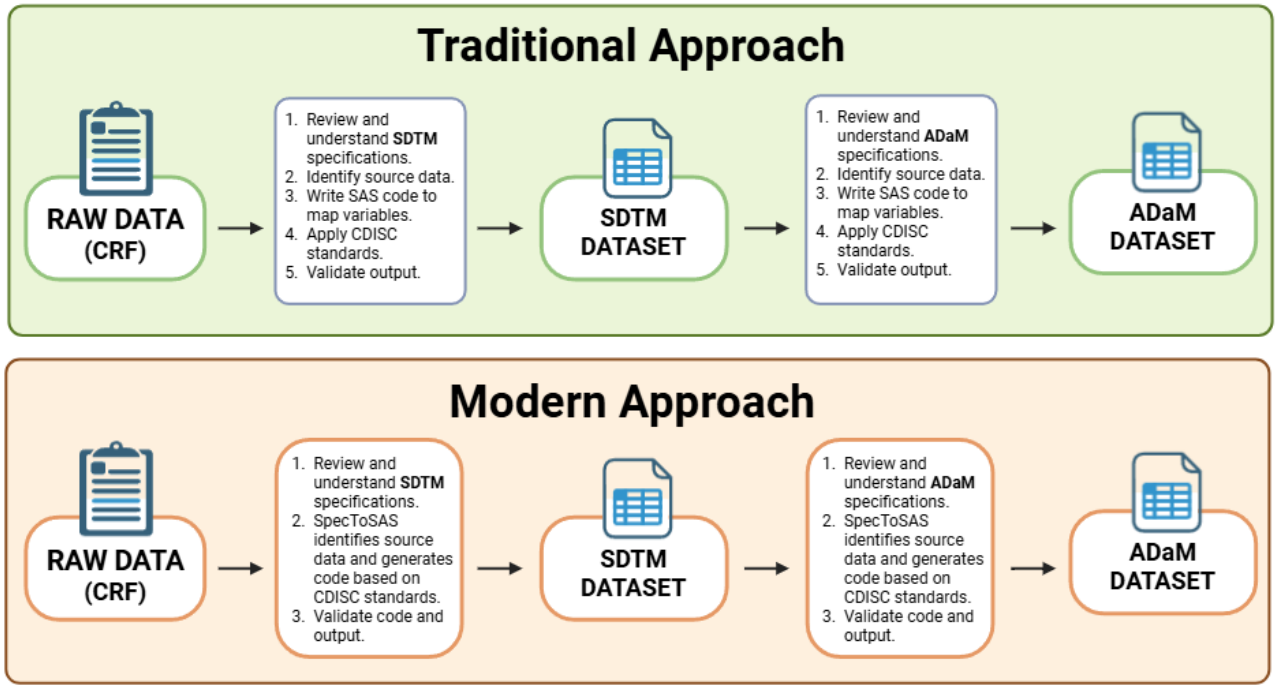

Figure 1. Traditional vs SpecToSAS-enabled workflow for clinical dataset generation.

As illustrated in Figure 1, the traditional approach requires five manual steps for both SDTM and ADaM: specification review, source data identification, SAS code writing, CDISC standards application, and output validation. SpecToSAS streamlines this to automated code generation followed by validation.

This paper introduces SpecToSAS, explaining both its architecture and practical benefits for clinical programming workflows. We describe the motivation behind its development, the core challenges in automating specification-to-code translation, and how our approach combines Python, LLMs, and intuitive interfaces to empower clinical programmers. We also explain how SpecToSAS adapts to the diverse specification formats commonly used in the industry while maintaining both technical accuracy and regulatory compliance.

Our goal is to demonstrate how AI can transform standard practices in clinical programming. By streamlining the mapping, derivation, and code generation processes, SpecToSAS improves both efficiency and quality throughout the clinical trial lifecycle, allowing programmers to focus on validation, complex problem-solving, and ensuring regulatory standards rather than repetitive coding tasks.

PROBLEM STATEMENT

The manual creation of SDTM and ADaM datasets from clinical trial specifications presents significant challenges that impact both efficiency and quality in pharmaceutical companies. Statistical programmers face a repetitive, time-intensive process that begins with interpreting complex Excel-based specification documents and ends with writing hundreds or thousands of lines of SAS code. Additionally, as studies grow in size and scope, so does the risk of inconsistencies, omissions, and regulatory findings due to subtle errors in interpretation or data handling [4].

Overall, this traditional workflow suffers from several critical issues that can result in several pain points, including:

- Resource constraint & inefficiency: Most of the SDTM and ADaM coding process involve repetitive tasks that take hundreds of hours. Programmers spend valuable time on these mechanical tasks rather than focusing on complex problem-solving, quality review, or addressing unique study requirements. This inefficiency becomes particularly problematic when specifications change mid-project, requiring extensive code revisions.
- Manual interpretation and translation errors: Specification files often contain hundreds of variables, each with detailed derivation logic, controlled terminology requirements, and conditional rules. Programmers must manually parse these specifications and translate them into executable SAS code, a process highly susceptible to human error. Misinterpreting a single derivation rule or overlooking a conditional statement can lead to incorrect datasets, requiring costly rework and delaying study timelines.
- Inconsistent code quality and maintainability: When programmers create code from scratch or reuse existing programs, the resulting SAS code varies widely in style, structure, and documentation quality. Each programmer brings their own coding conventions, making it difficult for others to understand and modify the code later. As programs are passed between team members and modified over multiple iterations, they become increasingly convoluted and difficult to maintain. What begins as clean, logical code often deteriorates into a patchwork that is challenging to debug, update, or validate.

- Barrier to innovation and agility: The traditional workflow limits the organization's ability to innovate and adapt. When programmers are consumed by repetitive coding tasks, they have little capacity to explore new analytical approaches, implement emerging statistical methods, or leverage modern data science tools. This becomes critical as the industry moves toward more complex trial designs and adaptive trials requiring rapid data updates. Additionally, when regulatory guidance changes or new CDISC standards are released, updating existing code across multiple studies becomes a massive undertaking. Organizations struggle to quickly adopt new requirements because each change must be manually implemented and validated across numerous programs. The traditional approach also fails to scale, because s companies expand their trial portfolios or conduct larger global studies with increasing data volumes, teams cannot handle the growing workload without proportionally increasing the number of statistical programmers working on those projects, creating resource bottlenecks that delay drug development timelines.

There is a clear need for an automated, robust, and adaptive solution that can ingest real-world Excel specifications, interpret and validate complex derivations, and generate reliable SAS code with consistency and efficiency. SpecToSAS addresses these challenges by enabling teams to focus on scientific insight and quality validation rather than manual programming labor.

SYSTEM OVERVIEW

SpecToSAS is an AI-powered application developed to generate SAS programs for clinical trial datasets directly from study specification files. The system is a combination of Python-based automation as well as a multi-agentic AI architecture [6] that is responsible for transforming specifications into SAS code while maintaining transparency, traceability, and programmer oversight.

The architecture consists of a staged pipeline in which specification file preprocessing, dependency management, and variable classification are completed using automation and then generative AI components are used [7]. With this design, we restrict the scope of AI involvement, while making sure that generative AI components operate only on well-defined, context-rich inputs rather than raw or fragmented specifications. SpecToSAS produces SAS code snippets corresponding to, derived variable logic using AI and direct variable mappings and variable attributes (labels, formats, lengths) using automation. These snippets are validated, concatenated, and delivered as a single executable SAS program which is suitable for execution in standard SAS programming environments. To support the ‘human in the loop’ model aligned with regulated clinical development workflows; the final output is subject to programmer inspection and validation.

DESIGN PHILOSOHY

The design philosophy of SpecToSAS is based on the constraints and expectations of regulated clinical programming environments. Instead of treating code generation as a purely generative task, SpecToSAS is developed as a controlled decision-support system that bolsters programmer productivity while preserving traceability, predictability, and accountability [8]. This philosophy is guided by 3 main principles.

1. Determinism Before Generation: Wherever derivation logic is rule based and repeatable; automation is prioritized. Clinical study specifications often include variables that are direct mappings or follow standardized transformation patterns. Using generative models to program such variables would lead to unnecessary variability and cost. [7]. To address this, SpecToSAS performs early-stage deterministic processing to classify variables as direct or derived, resolve specification fragmentation and create dependency graphs and decide execution order. Once these steps are completed only then, Generative AI components come into action for variables that require semantic interpretation. This ensures that AI reasoning is applied purposefully and selectively.

2. Context Over Inference: Large language models work best with a structured and explicit context. Raw study specifications, in contrast, are often fragmented, ambiguous, and distributed across multiple tabs. SpecToSAS therefore transforms specifications into a unified, normalized internal representation before invoking any AI components. Each generative agent operates with explicit source domain information, dependency-aware variable ordering, consolidated derivation rules and retrieved in-house coding examples This approach minimizes reliance on implicit inference by the model and replaces it with context-aware reasoning grounded in study-specific and organization-specific knowledge. Because of this, the generated code is more consistent, interpretable, and in line with established programming standards.

3. Programmer Ownership and Human-in-the-Loop Control: SpecToSAS is intentionally designed as an AI assisted programming system. All generated SAS codes are human-readable, modular, editable, and traceable to the originating specification and retrieved examples. The system does not execute code, have access to study data, or replace programmer judgment. The app in turn accelerates routine and cognitively intensive steps while preserving human responsibility for final review, validation, and regulatory compliance. This design aligns with existing clinical development workflows, where accountability, auditability, and explainability are critical. By maintaining programmer ownership of outputs, SpecToSAS supports adoption without disrupting established validation and quality control processes.

COMPARATIVE DESIGN CONSIDERATIONS

There were 2 main design considerations for SpecToSAS:

1. SpectoSAS vs Automation: Traditional automation in clinical programming relies on predefined rules, templates, and macros. Such systems are effective for fully standardized and predictable logic, but with the variability inherent in real-world study specifications, including free-text derivations, study-specific exceptions, and cross-domain dependencies, only automation cannot suffice. Automation requires explicit encoding of all possible derivation patterns, leading to systems that are vulnerable, difficult to maintain, and costly to extend across studies with differing conventions. With the increase in specification complexity, there is also an increase in the effort required to maintain deterministic rule sets. SpecToSAS goes beyond traditional automation by implementing context-aware reasoning. This context-aware reasoning is made possible through RAG, based on which the system utilizes company specific legacy codes and standards instead of relying solely on predefined rules. Automation plays a role in establishing structure, enforce order, and handle repeatable logic, while AI agents are responsible for variables where complex interpretation is required. This hybrid approach allows the system to adapt to heterogeneous specification styles without rule expansion. SpecToSAS achieves a balance that neither pure automation nor unconstrained generative systems can provide. The result is a system that scales more effectively than rule-based automation while remaining predictable, explainable, and aligned with regulatory expectations.

2. Context-aware Generation vs Prompt Based LLM Usage: SpecToSAS implements context-aware code generation, rather than traditional prompt-based LLM usage [7]. In a prompt-only model, derivation logic is passed directly to a language model with minimal structural constraints, relying on the model's general training to infer intent. This method results in variability, reduces reproducibility, and hinders alignment with institutional standards. By contrast, SpecToSAS constructs a rich execution context before invoking any generative components. This execution context is constructed using a RAG framework [9]. Before code generation, relevant company specific examples, including legacy SAS programs, standardized derivations, in-house macros, and reusable templates, are retrieved from an internal code repository. Basing the code generation on the retrieved, company-specific examples, RAG constrains the language model and ensures alignment with established programming standards. Additionally, each AI agent receives features like, variable metadata, derivation rules, source domain information, Dependency based variable ordering and retrieved code examples. This context ensures that generation occurs within clearly defined boundaries. As a result, SpecToSAS behaves as a context-driven generation system, where the LLM works within a controlled workflow rather than as an unconstrained code author.

IN-HOUSE KNOWLEDGE REPOSITORY

At the core of SpecToSAS's context-aware generation capability is an internal, institution-specific knowledge repository which was developed to capture reusable clinical programming knowledge. This repository serves as the primary retrieval source for the system's RAG framework [9]. The repository is stored as a JSON file, consisting of mappings between variable-level derivation and its corresponding validated SAS code. The file is created using legacy programs, in-house templates, and macros that reflect established organizational practices.By storing derivations and codes at the variable level, the repository enables precise retrieval. With the help of this repository, the app promotes consistent reuse and avoids duplication of work.

TECHNOLOGY STACK

SpecToSAS is a Python Flask application [10] with the frontend designed using HTML and CSS. Python scripts handle specification preprocessing, standardization, dependency graph construction, and code generation for direct variables. Pandas [11] is used for data manipulation. An OpenAI [12] LLM (GPT-4-32K) is used for the AI agents to interpret derivation logic and generate SAS code. To constrain and guide model outputs, SpecToSAS also uses RAG based on an in-house knowledge repository. This repository serves as the retrieval source and includes, legacy SAS programs, reusable derivation patterns, In-house macros, Standardized templates. The retrieval mechanism supports both literal and semantic search. This approach significantly reduces reliance on unconstrained generation and promotes consistency with company specific programming standards.

MULTI-AGENT ARCHITECTURE

The heart of the SpecToSAS pipeline is a multi-agent AI architecture [6] which consists of multiple AI agents responsible for independent specialized tasks within the pipeline. Each agent operates independently within a clearly defined scope while exchanging structured intermediate outputs with other agents to achieve coordinated behavior as shown in Figure 2.

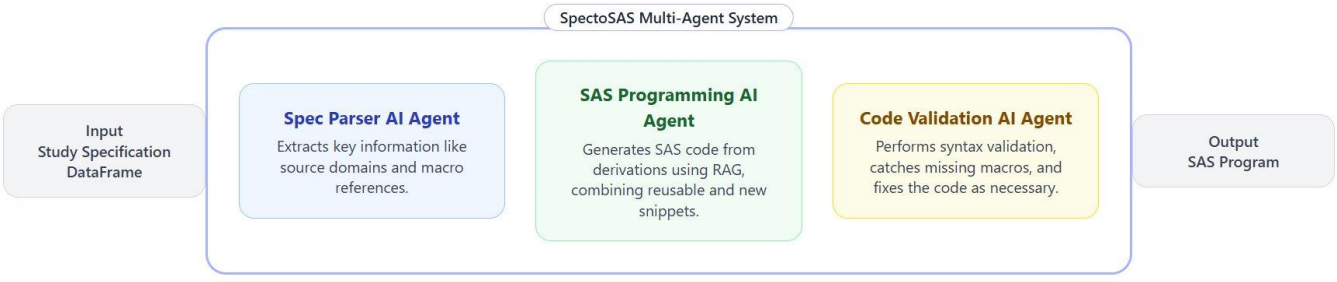

Figure 2. SpecToSAS Multi-Agent System

1. Specification Parsing Agent

The Specification Parsing Agent performs semantic interpretation of derivation rules beyond what can be inferred deterministically. Its responsibilities include, Identification of source domains, detection of macro references for reuse and classification of derivation complexity. As shown in Figure 2, this agent operates alongside Python automation that classifies variables as direct or derived. This separation ensures that rule-based classification is handled by automation, while semantic interpretation is delegated to the AI agent.

2. SAS Programming Agent

The SAS Programming Agent generates SAS code for derived variables. It takes context features generated by the parsing agent as well as derivation rules to generate code. The variables categorized as Derived (not directly mapped) are grouped by their source domain before code generation, for consistent handling of merges, joins, and data step logic. Direct variables are excluded from this agent and are handled entirely through python automation, ensuring efficient processing of straightforward mappings.

SAS Programming Agent's code generation is supported by a RAG mechanism [9], as shown in Figure 3, that leverages a curated, JSON-based in-house knowledge repository containing variable-level derivation and code pairs extracted from legacy studies and standardized templates. For each derived variable, the agent performs a hybrid semantic and literal retrieval against this in-house memory before initiating code synthesis. This retrieval strategy reduces unnecessary free-form generation, promotes reuse of existing code, and ensures that new code is generated only when no validated alternatives exist. By differentiating between reuse, adapt, and generate, SpecToSAS maintains both flexibility and control in AI-assisted code generation. The retrieval process could result in 3 possibilities which are as follows:

- Exact match: If an exact match between the requested derivation and an existing example in the repository is found, then the matching SAS code is reused without modification. This reuses existing validated logic and increases consistency with prior studies.
- Partial match: If no exact match is found but a similar example with the similarity score higher than 0.75 is found, the retrieved code is provided to the LLM as a reference example, along with the current derivation rule. The model adapts the example to the new context, modifying the code where required while retaining existing programming patterns.
- No match: If no match was found, the LLM generates SAS code based on the derivation rule and context features.

3. Code Validation Agent

The final validation check is performed by the Code Validation Agent on all the generated code snippets. The agent takes care of checking the syntax, structural inconsistency, and detection of missing or mis-ordered statements. Based on these checks, the agent also corrects the identified issues before providing the final code output.

All together, the 3 AI agents define the multi-agent AI system where each agent is performing and independent task with an individual context and role assigned, while exchanging information as they work towards a shared goal.

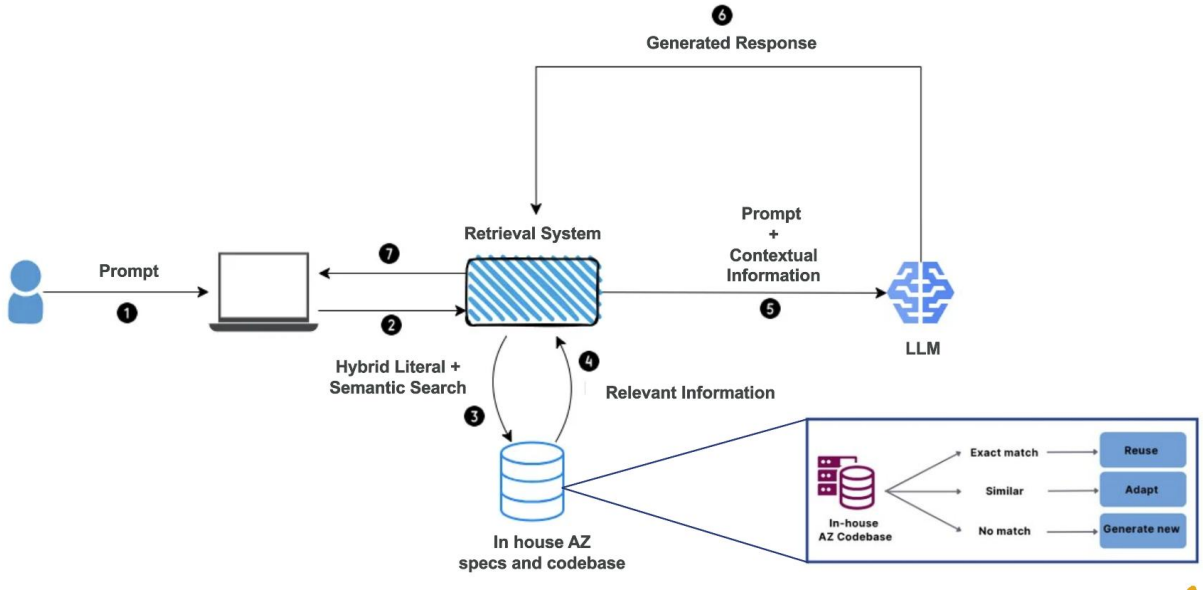

Figure 3. Retrieval Augmented Generation (RAG)

WORKFLOW

The entire pipeline, starting from uploading the study specification to the generated SAS code, is illustrated in Figure 6. Components highlighted in yellow represent automation steps, while components highlighted in blue correspond to AI steps. The following sections provide a detailed breakdown of each stage of the pipeline and further describe how these components interact to produce a SAS program.

1. Variables Attribute Code Snippet

The workflow (Figure 6) begins with an excel/csv based study specification file as input. The first step in the pipeline is to extract the attributes of each variable present in the specification file and create a corresponding PROC SQL SAS code for attribute setting. For example, Table 1 consists of a subset of variables for ADAE with respective attributes highlighted in yellow and the corresponding SAS code snippet generated by python automation is as follows.

Domain	Variable	Label	Type	Length	Origin
ADAE	STUDYID	Study Identifier	Char	200	ADSL.STUDYID
ADAE	USUBJID	Unique Subject Identifier	Char	200	ADSL.USUBJID
ADAE	SUBJID	Subject Identifier for the Study	Char	200	ADSL.SUBJID
ADAE	AESEQ	Sequence Number	Num	8	Sort by SUBJID AEDECOD ASTDT AETERM and assign sequential integer within each subject.
ADAE	ASTDT	Analysis Start Date	Num	8	1) If full date, convert date portion of AE.aestdat to numeric SAS date.
ADAE	AETERM	Reported Term for the Adverse Event	Char	200	Set to AE.aeterm value. Convert to upper case.

Table 1. Variable Attributes in Study 1 ADAE Specification.

```
proc sql;
create table output as
select
STUDYID length = 200 label = 'Study Identifier',
USUBJID length = 200 label = 'Unique Subject Identifier',
SUBJID length = 200 label = 'Study Identifier for the Study',
             AESEQ label = 'Sequence Number',
             AETERM length = 200 label = 'Reported Term for Adverse Events',
             ASTDT label = 'Analysis Start Date',
      from input;
quit;
```

2. Specification Normalization

The second step is responsible for handling the fragmentation in a specification file (if present) due to which content gets split across multiple tabs. Spec normalization converts a fragmented specification into a unified single tab. This step ensures there is no unnecessary back and forth across tabs by bringing all required information into 1 single tab. Figure 4 illustrates an example of specification normalization where PARAMCD level derivations are pulled from value level metadata to variable level metadata.

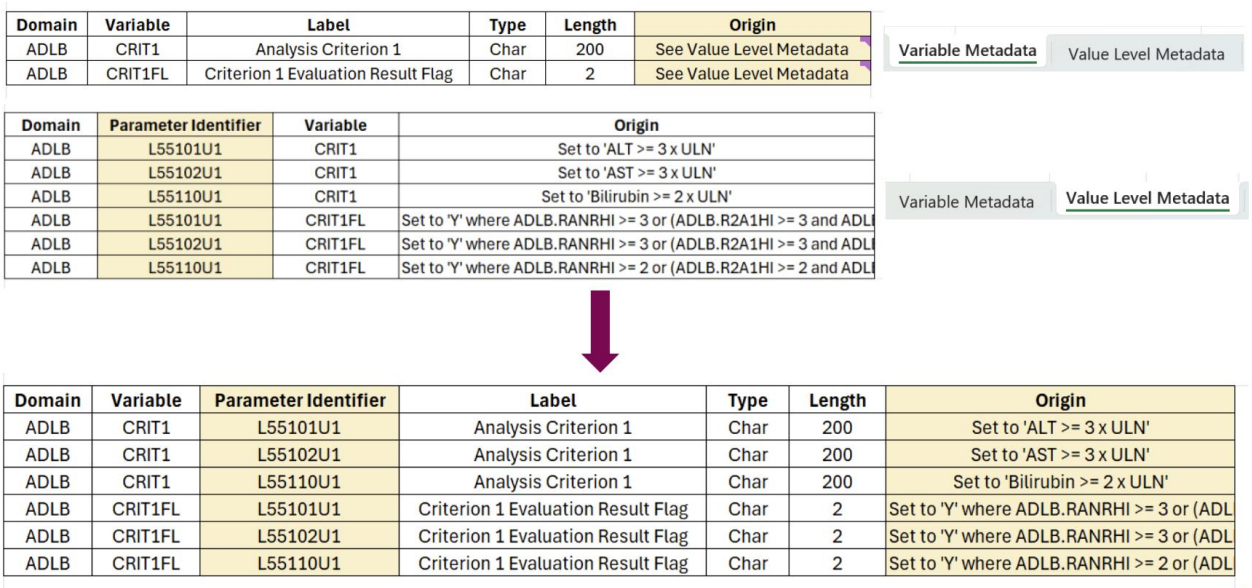

Figure 4. Specification Normalization for Study 1 ADAE dataset.

3. Dependency Management

The third step is to generate a dependency graph for the variables to identify variable relationships and decide execution order. An example of one such dependency graph for variables in the ADAE dataset is as shown in figure 5. Each node in this graph represents a variable, and the outward arrows indicate dependency. Using this dependency graph, variables are reordered such that independent variables are processed first, followed by variables with increasing dependency depth. This ensures that downstream code generation respects derivation dependencies and produces executable SAS programs without manual reordering.

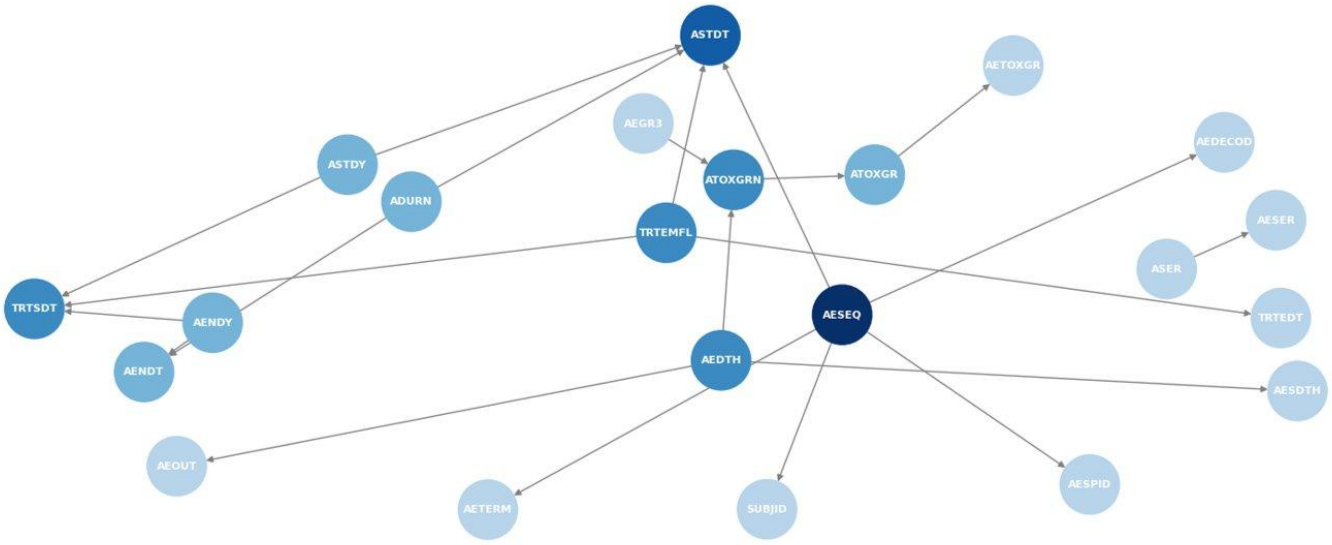

Figure 5. Variables dependency graph

4. Variable Classification and Processing

Variables are classified as direct or derived based on the derivation of rule complexity. Other features like source domain and macro reference, are also identified. An example of these 3 features is illustrated in Table 2. Direct variables are grouped by source domain and a python automation script generates the corresponding SAS code with the relevant mappings. Derived variables are grouped by the source domain and passed to the multi-agent AI system. The Specification Parsing Agent extracts semantic features, after which the SAS Programming Agent generates code using retrieved institutional knowledge where applicable as discussed previously.

Domain	Variable	Label	Type	Length	Origin	Direct Variable	Source Domain	Macro Reference
ADAE	STUDYID	Study Identifier	Char	200	ADSL.STUDYID	Y	ADSL	
ADAE	USUBJID	Unique Subject Identifier	Char	200	ADSL.USUBJID	Y	ADSL	
ADAE	SUBJID	Subject Identifier for the Study	Char	200	ADSL.SUBJID	Y	ADSL	
ADAE	ASTDT	Analysis Start Date	Num	8	1) If full date, convert date portion of AE.aestdat to numeric SAS date. 2) if partial date then do ; a)Missing day- Impute the 1st of the month unless month is the same as month of the first dose of study drug then impute first dose date b) Missing day and month -Impute 1st January unless year is the same as first dose date then impute first dose date c) Completely missing-Impute first dose date unless the end date suggests it could have started prior to this in which case impute the 1st January of the same year as the end date.	N	AE	
ADAE	AETERM	Reported Term for the Adverse Event	Char	200	Set to AE.aeterm value. Convert to upper case.	N	AE	
ADAE	AEDECOD	Dictionary Derived Term	Char	200	Set to AE.pt_name value	N	AE	
ADAE	AESDTH	Results in Death	Char	200	SERAE.AESDTH. Merge SERAE to AE by subject and aeno.'1' = 'Y' and '0' = 'N'	N	AE, SERAE	
ADAE	MEDDRAV	MedDRA Version	Char	200	SUPPAE.QVAL where SUPPAE.QNAM = "MEDDRAV"	N	SUPPAE	%m_a_addsupp (dsin=)
ADAE	AESEQ	Sequence Number	Num	8	Sort by SUBJID AEDECOD ASTDT AETERM AESPID and assign sequential integer within each subject.	N	ADAE	

Table 2. Variable Features in ADAE dataset.

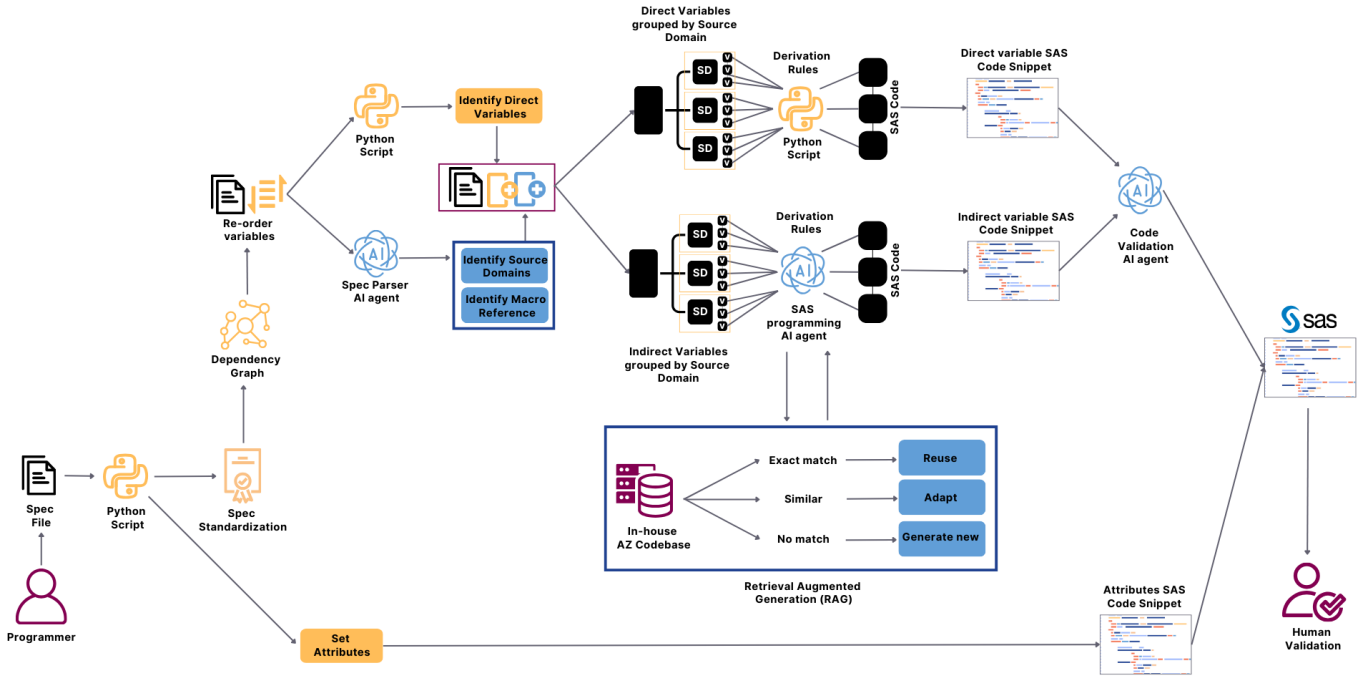

Figure 6: SpecToSAS Workflow

5. Code Assembly and Validation
Generated code snippets, including direct variables code, derived variables code, macro references, and attribute setting code, are all passed through the Code Validation Agent. Validated code snippets are then assembled into a complete SAS program. The final output is delivered to the programmer for review and can be copied and pasted to respective SAS environments where study data is stored for execution and dataset creation. This preserves human validation while substantially reducing manual programming effort.

APPLICATION INTERFACE AND USAGE OVERVIEW

To demonstrate the User Interface and experience (UI/UX) of the app, we have Figures 7–10 that provide a few screenshots showcasing an end-to-end code generation cycle. Programmers can create SAS code straight from study specification files using the SpecToSAS application's lightweight, browser-based interface, which eliminates the need for extra settings. Users start the process by uploading an Excel-based specification file via the interface, as illustrated in Figure 7.

The program automatically recognizes multiple ADaM or SDTM specification tabs in the uploaded file exist and displays the available dataset options for selection, as illustrated in Figure 8. This ensures clarity and avoids inadvertent code development across several domains by enabling users to specifically select the target dataset for which SAS code should be written. The application also has a column mapping feature, as shown in Figure 9, to account for variations in specifications' column names. Users can dynamically map the necessary fields (such as variable name, label, derivation rule, and source domain) to the appropriate column names in the uploaded specification file if it lacks the expected column names or uses a naming scheme unique to the study. Because of its adaptability, SpecToSAS may function with a variety of specification templates without the need for manual file reformatting before uploading.

The application runs the SpecToSAS workflow outlined in the previous sections after a dataset has been chosen. The entire SAS program is produced and shown in the interface in a little processing time, as shown in Figure 10. The generated code consists of macros calls, code for directly mapped variables using automation, code for derived variables using the multi-agent AI system and attribute setting code snippet to specifies labels, formats, and lengths. To maintain readability and conformity to common programming norms, the output is shown as a single, continuous SAS program in a code editor display. Before the resulting code is executed, users can examine particular sections by scrolling across it, as illustrated in Figure 10.

The interface does not run code or link to study data. It facilitates human-in-the-loop validation and preserves study data confidentiality. Alternatively, users can utilize a single operation to copy the entire created program and paste it into their local SAS environment for review, execution, and validation on study data.

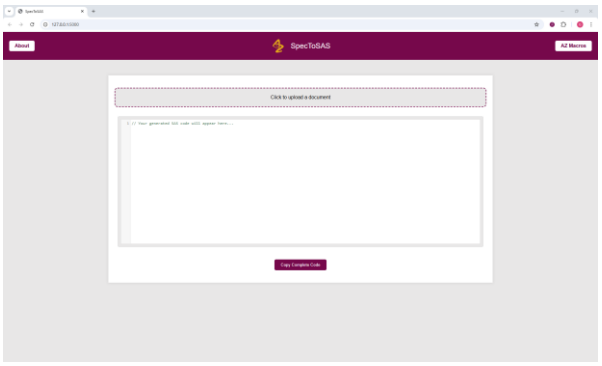

Figure 7. SpecToSAS User Interface

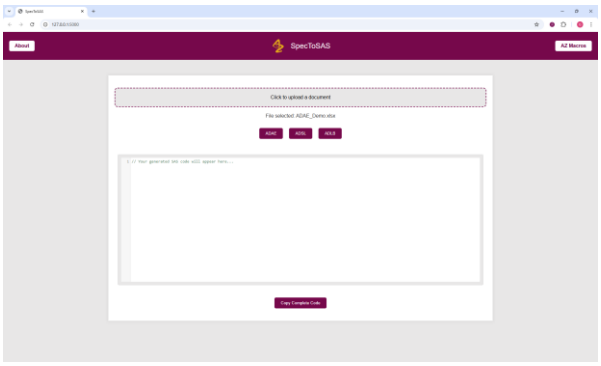

Figure 8. ADAM Dataset Options

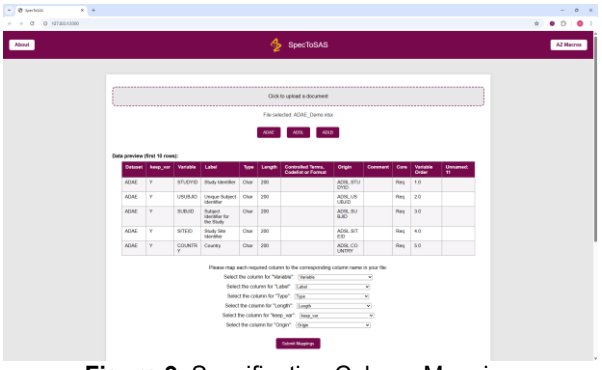

Figure 9. Specification Column Mapping

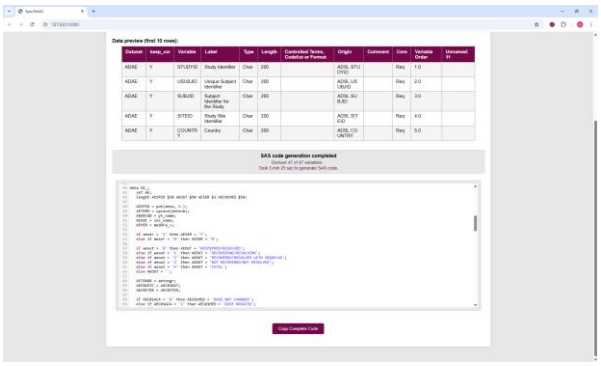

Figure 10. SpecToSAS Output

TESTING RESULTS AND IMPACT

To assess SpecToSAS's efficacy, the focus of the testing was on ADaM dataset development. ADaM programming involves more programming effort than SDTM due to complicated derivation logic, cross-domain relationships, and study-specific standards. SDTM, on the other hand, is more standardized and accessible to rule-based automation. Hence, we chose ADaMs as a more useful testbed for evaluating the effects of AI-assisted code creation.

Up to five safety ADaM datasets (ADSL, ADAE, ADCM, ADMH, and ADVS) were included in each of the four studies resulting in a total of 14 datasets. The generated code was compared against reference datasets that had been verified and were ready for production.

TESTING METHODOLOGY

As shown in Figure 11, a consistent, step-by-step testing process was used for all datasets. Using SpecToSAS, SAS code was produced for every dataset straight from the study specification. After being modified for the internal SAS environment, the produced code was run and compared to validated ADaM datasets. Next, the test programmer conducted incremental modifications until the comparison was 100% clean.

For every dataset, four system-level evaluation metrics were noted. These included how long it took SpecToSAS to produce the SAS code, how long it took to set up the initial environment and fix reference-related problems, how many variables were derived before any human intervention, and how much more time it took for human fixes to reach perfect agreement with the validated reference dataset. When combined, these measures offer a comprehensive picture of residual manual effort as well as automation efficiency.

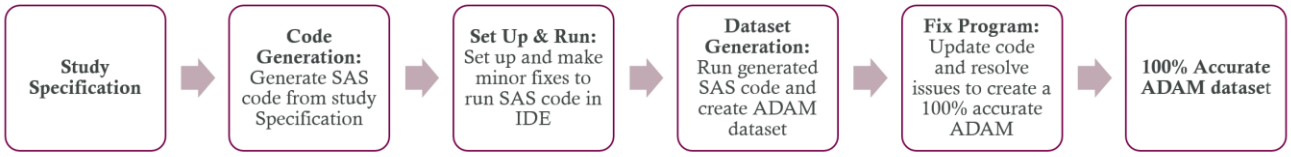

Figure 11. Step-by-step methodology to test SpecToSAS-generated code.

QUANTITATIVE RESULTS

Across the evaluated datasets, SpecToSAS consistently generated complete SAS programs within 5 to 15 minutes. The time required for initial setup and execution ranged between 15 and 40 minutes, depending on dataset complexity and the degree of alignment between generated code and the study-specific SAS environment. The time per dataset to make code fixes to reach 100% accuracy with respect to the validated dataset was 25 to 55 minutes. Initial derivation accuracy, measured as the proportion of eligible variables that derived prior to human intervention, ranged from 75% to 95%. Variability in this metric was primarily driven by the completeness and clarity of derivation rules in the specification files. Figure 12 presents the average time distribution by ADaM dataset, while Figure 13 summarizes overall performance across all evaluated datasets.

Sometimes files can be formatted (i.e. some words cross-out, color highlights, links for other files), which AI would not be able to access or interpret. These variables were eliminated from testing. For example, the ADAE specification file had 70 variables with the derivation rule “Check SAP for derivation”, which SpecToSAS couldn’t interpret. As a result, the dataset took significantly longer to prepare, and the QC programmer had to code manually for those variables (Figure 13 c, d) and hence the impact was also seen on the end-to-end time effort on ADAE dataset.

During testing, two main categories of issues were identified: setup-related and code-related issues. Setup issues were associated with proper referencing within the internal SAS coding environment. For example, the tool-generated code was using incorrect or missing library references as well as incorrect macros and dataset names which resulted in errors like “Reference not resolved”. Another common error was related to referencing internal macros for merging the main SDTM dataset and the corresponding SUPP dataset. The macro code was part of the AI memory, and it was expected that the tool would use macro directly and produce MAIN+SUPP dataset. However, the tool has produced a code that generated both macro versions of MAIN+SUPP as well as a more manual approach for merging two datasets. As for the code issues, the following mistakes were encountered: multiple merge statements within single data step, “keep/drop” statements placed too early resulting in missing required variables, set and merge statements both present in data step, merging by wrong variables that don't exist in the dataset, rederiving variables that should have been already present from MAIN+SUPP merge, using “proc sort” in a wrong places or by incorrect variables and others. While it is easy to fix some of them (e.g. “keep/drop” statements, merging and sorting wrong variables), other errors related to code logic is what contributed to the Code fix time.

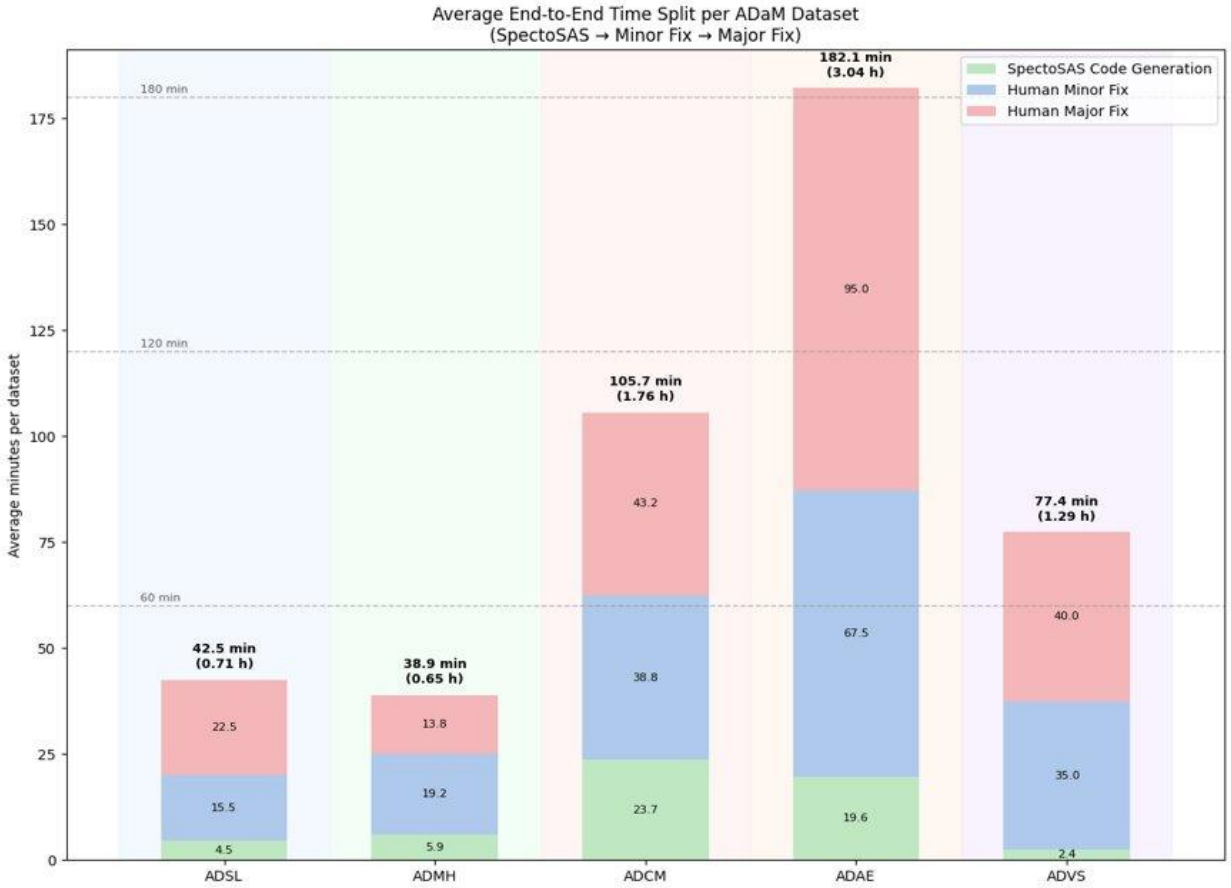

Figure 12. Average time split per ADaM dataset.

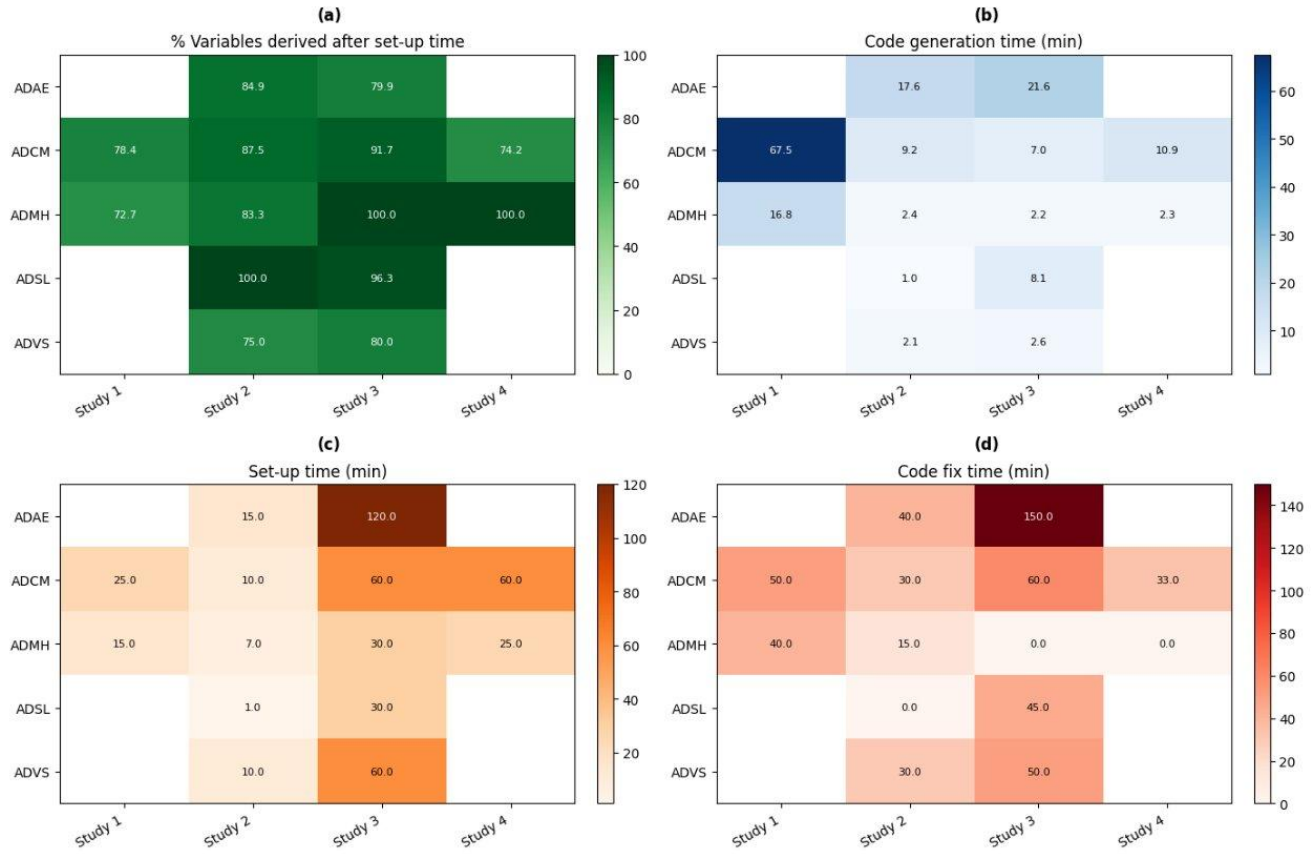

Figure 13. The summary of the results of running the SpecToSAS on a total of 14 datasets across 4 clinical studies.

In addition to saving efforts and time, it is also worth noting that testing programmers were at different career levels (junior and mid-levels of programmers) and were not familiar with any study level information. They were still able to generate code and easily perform programming activities. This indicates that SpecToSAS not only accelerates programming tasks but also lowers the barrier to contribution, enabling broader team participation while maintaining quality and oversight.

Overall, the results demonstrate that SpecToSAS functions as an effective AI-assisted programming system, reducing manual effort and development time while preserving the human judgment and accountability required in regulated clinical programming environments.

FUTURE DEVELOPMENT

Future development of SpecToSAS will focus on reducing residual manual effort and expanding system coverage. Key priorities include progressing toward zero setup time by resolving identified environment and reference-related issues that will minimize number of errors and hence minimal time is required from programmers to update the code. We aim to extend support to additional ADaM datasets beyond the safety domains evaluated in this study, expand to SDTM code generation and potentially other therapeutic areas as well.

Further work will strengthen context construction through closer alignment with standardized specification formats and expansion of the in-house knowledge repository to capture recurring derivation rules, dataset patterns, and validated implementations. In parallel, in-house macro coverage will be extended to support additional automated derivations. To enable continuous improvement, a structured feedback mechanism will be introduced to track setup and code-level issues. Exploratory work will also assess lightweight supervised fine-tuning approaches, such as LoRA and QLoRA [13], to further optimize domain-specific code generation.

CONCLUSION

SpecToSAS is an innovative tool that leverages Python-based automation and AI capabilities to translate variable derivations into SAS code. This approach substantially reduces effort for de novo programming and as well as use and updating the reference programs. While the output requires human review and validation, generating specification-tailored code meaningfully shortens end-to-end development time.

Testing across 14 safety datasets across 4 different oncological studies demonstrated that 75–95% of variables were correctly programmed, with a range of 5 -15 minutes for code generation and up to 1 hour for human review. Integration of reference codes and existing internal macros substantially increased correctness, indicating that reuse of validated components enhances

fidelity to specifications and reduces defect rates. The code produced by the tool can generate 75-95% of the dataset before human intervention. With less than 1 hour of human review, a programmer can generate a 100% accurate ADAM dataset. The human review time was usually related to the company-specific programming environment and set-up programming mistakes that could be easily fixed. Finally, programmers at different levels of career and knowledge about the study were able to use the tool and update the programs easily.

While deployment to date has focused on validation of safety ADAM datasets in Oncology studies, the methodology is scalable to code generation across all therapeutic areas and to a broader dataset scope, including efficacy-related datasets as well as SDTMs.

REFERENCES

[1]. U.S. Food and Drug Administration. (2018). E6(R2) Good Clinical Practice: Integrated Addendum to ICH E6(R1) Guidance for Industry. FDA. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/e6r2-good-clinical-practice-integrated-addendum-ich-e6r1

[2]. Kush, R. D., Warzel, D., Kush, M. A., Sherman, A., Navarro, E. A., Fitzmartin, R., ... & Houstoun, M. (2012). SHARE: A proposed metadata repository to facilitate sharing of clinical research data. Trials, 13(1), 122. https://doi.org/10.1186/1745-6215-13-122

[3]. CDISC. (2021). Analysis Data Model (ADaM) Implementation Guide Version 1.3. Clinical Data Interchange Standards Consortium. https://www.cdisc.org/standards/foundational/adam

[4]. Huser, V., Sastry, C., Breymaier, M., Idriss, A., & Cimino, J. J. (2015). Standardizing data exchange for clinical research protocols and case report forms: An assessment of the suitability of the Clinical Data Interchange Standards Consortium (CDISC) Operational Data Model (ODM). Drug Information Journal, 49(5), 578-590. https://doi.org/10.1177/2168479015591952

[5]. Spinellis, D. (2003). Code Reading: The Open Source Perspective. Addison-Wesley Professional.

[6] Wu Q, Wang J, Shen Y, et al. AutoGen: Enabling next-generation large language model applications via multi-agent conversation. arXiv preprint. 2023. https://arxiv.org/abs/2308.08155

[7] Zaharia M, Chen A, Guo A, et al. From prompt engineering to context engineering. Stanford DAWN Project Technical Report. 2024. https://dawn.cs.stanford.edu/2024/01/15/context-engineering/

[8] Amershi S, Weld D, Vorvoreanu M, et al. Guidelines for human–AI interaction. Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. 2019. https://dl.acm.org/doi/10.1145/3290605.3300233

[9] Lewis P, Perez E, Piktus A, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems (NeurIPS). 2020. https://arxiv.org/abs/2005.11401

[10] Grinberg M. Flask Web Development: Developing Web Applications with Python. 2nd ed. O'Reilly Media; 2018. https://www.oreilly.com/library/view/flask-web-development/9781491991725/

[11] McKinney W. Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference (SciPy). 2010. https://conference.scipy.org/proceedings/scipy2010/mckinney.html

[12] OpenAI. GPT-4 Technical Report. arXiv preprint. 2023. https://arxiv.org/abs/2303.08774

[13] Hu EJ; Shen Y; Wallis P; Allen-Zhu Z; Li Y; Wang L; Chen W. Low-Rank Adaptation of Large Language Models. Proceedings of the International Conference on Learning Representations (ICLR); 2022. https://arxiv.org/abs/2106.09685

ACKNOWLEDGMENTS

The authors would like to thank Avinash Reddy Pati for his guidance, support, and valuable insights throughout the development of SpecToSAS. His encouragement and strategic perspective were instrumental in shaping the system's direction and ensuring alignment with organizational and regulatory expectations.

CONTACT INFORMATION

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

Author Name: Gargee Vaidya
Company: AstraZeneca
Address: 1004 Middlegate Road, Mississauga, Ontario, L4Y 1M4
Work Phone: 1-800-565-5877
Email: gargee.vaidya1@astrazeneca.com
Website: AstraZeneca Canada