Paper ML18

Real-Time AI-Driven TLF Generation with GAutoTLF

Hao Xu, AstraZeneca, Mississauga, Canada
Tyler Rowsell, AstraZeneca, Mississauga, Canada

Abstract

Creating Tables, Listings, and Figures (TLFs) for clinical studies is a common and time-consuming process, particularly due to the requirements of double programming validation.

GAutoTLF (**G**uided **Auto**mation of **T**able **L**istings and **F**igures for Clinical Study Reports) is an AI-driven solution that automates this process by integrating generative AI with standard TLF shells, template code and study specific clinical metadata. Users submit analysis requests on standard TLFs through simple prompts and receive executable R code for rapid, consistent TLF generation within minutes. The adaptable and scalable design allows GAutoTLF to be leveraged across various therapeutic areas, providing biometrics teams with substantial improvements in productivity, cost efficiency, and quality. This presentation will also feature a brief demo of GAutoTLF's capabilities, showcasing its real-time analysis and transformative impact on clinical reporting workflows.

Introduction

In the pharmaceutical industry, clinical study reporting relies heavily on the generation of Tables, Listings, and Figures (TLFs). Traditionally, this process is manual, resource-intensive, and prone to human error. To work effectively on TLFs, programmers must be highly familiar with clinical study data, which is often complex and spans multiple data domains. Table shells may frequently change during the course of the study, increasing programming challenges as programmers must continually update specifications and code as necessary. Furthermore, programmers spend significant time writing and validating code for each output, often under tight timelines. These challenges are amplified in large-scale clinical study trials, where the volume and complexity of TLFs demand both speed and precision.

The need for automation in TLF generation has grown as organizations seek to improve operational efficiency, reduce costs, and accelerate decision-making. Existing solutions, such as template-based reference programming or interactive analysis tools (PowerBI™, R Shiny™, Spotfire™), offer partial relief but still require manual coding and study-specific customization. In addition, many critical exploratory analyses remain time-consuming, limiting the ability to respond quickly.

GAutoTLF (Guided Automation of Tables, Listings, and Figures) addresses these challenges by embedding Generative AI into the clinical reporting workflow. The solution integrates AI-driven R code generation with AstraZeneca's Standard TLF Shells, template code and study metadata, enabling rapid, consistent, and scalable TLF production. Users interact through a guided

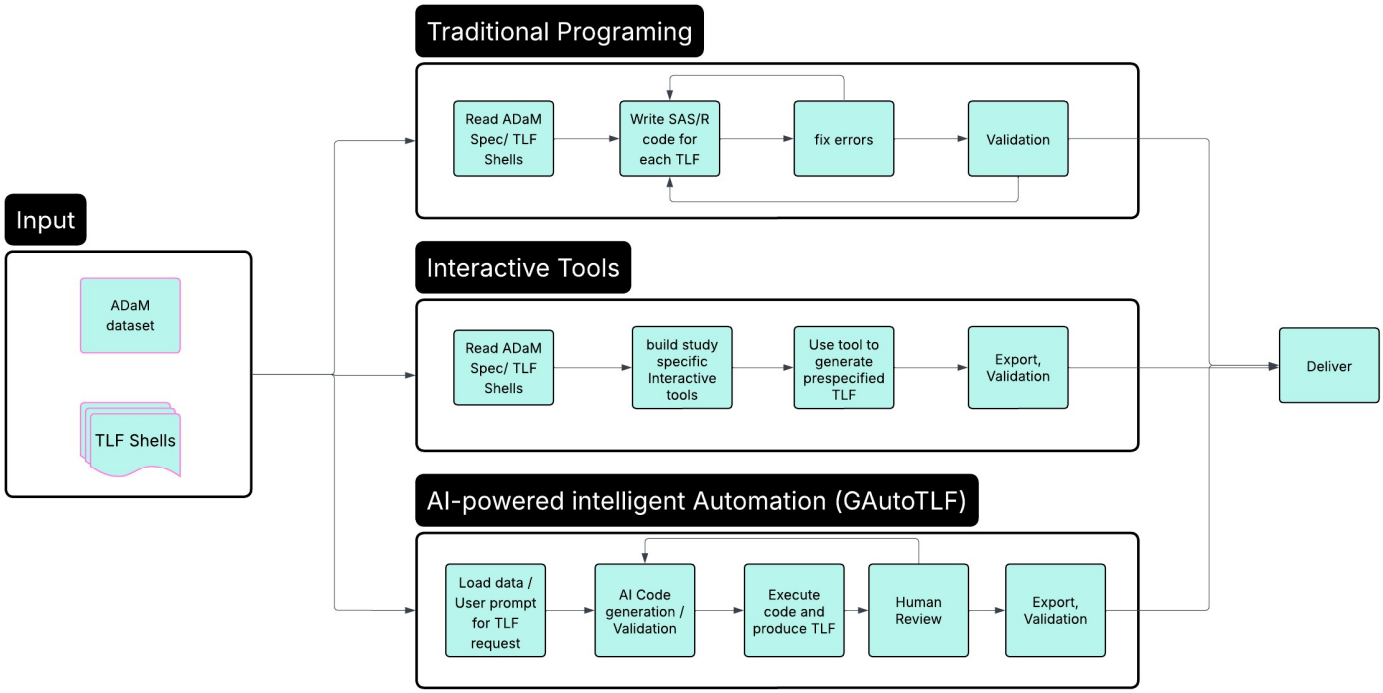
interface, submitting simple prompts to generate executable R code and then executing the code to instantly produce TLF outputs. Beyond standard outputs, GAutoTLF supports real-time exploratory analysis, empowering biometrics teams to derive insights on demand. GAutoTLF focuses on core safety outputs first, with all analysis performed within an R Shiny application.

This innovation represents a paradigm shift in clinical reporting: from manual programming to intelligent automation. By reducing turnaround times from hours to minutes and ensuring consistency across studies, GAutoTLF accelerates evidence delivery, and positions biometrics teams at the forefront of AI adoption in pharmaceutical development.

The three common approaches differ in how specification, coding, and maintenance are handled:

- Traditional manual programming requires bespoke code for every TLF specified in shells; scope changes invariably trigger code updates and revalidation.
- Interactive tools (e.g., PowerBI™) accelerate delivery when predefined TLFs exist within the tool, particularly for subgroup or exploratory views; however, these tools are typically customized per study, so new studies or expanded scopes often means updating or building new tool.
- AI-powered intelligent automation (e.g., GAutoTLF) shifts effort to a prompt-driven pipeline: users provide ADaM data, select a template, and describe requirements in natural language. The system generates executable R code, executes it locally, and iteratively fixes errors. The result is faster turnaround and broad applicability across studies conforming to CDISC™ ADaM standards.

Figure 1. TLF Process



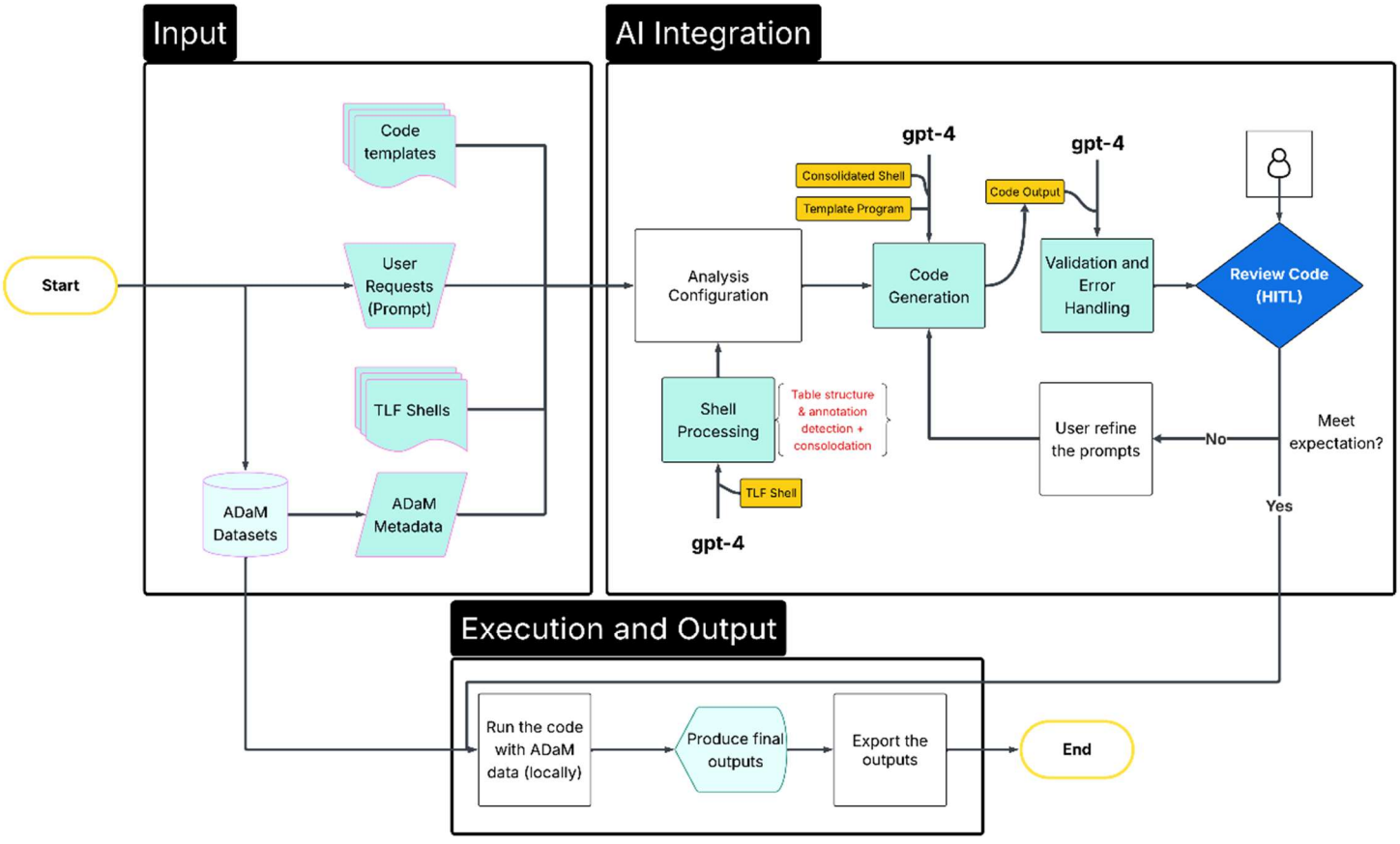
System Overview and Architecture

GAutoTLF's is an AI-enabled R Shiny application. It's user interface is a guided, Shiny-based workflow with two tabs: Introduction tab for orientation, Code Generation tab for the end-to-end analysis flow (data upload, dataset exploration, configuration, AI code generation, code editor with chat assistant, and output display). It uses clear step indicators, tooltips, and a modern layout to help users move from ADaM ingestion to prompt-driven code generation and local execution, with outputs rendered as standardized TLF and convenient export options.

GAutoTLF Workflow

The GAutoTLF workflow spans three integrated phases: Input, AI Implementation, and Execution & Output.

Figure 2. GAutoTLF workflow



Input phase

The **Input Phase** establishes the foundation for the GAutoTLF workflow by gathering all essential components needed for automated TLF generation. The process begins by connecting to the study database and extracting ADaM metadata. This metadata contains critical information about variables, data domains, and analysis populations that define the structure and content of the study data. Users submit analysis requests through natural language prompts, specifying the tables, listings, and figures they need. These requests describe the desired outputs in plain terms without requiring technical programming knowledge. The system incorporates standard TLF shells and pre-defined code templates to ensure all outputs meet regulatory and organizational requirements. These templates provide consistent structure and formatting across all generated deliverables. Rather than requiring programmers to manually read data specifications and interpret complex data structures, AI automates this critical step. The system analyzes study metadata and/or shell to identify the correct variables, appropriate data domains, and relevant analysis populations for each requested output. By consolidating metadata, user requirements, and template structures, the Input Phase creates a comprehensive configuration that guides the AI-driven code generation process. This foundation ensures that subsequent phases produce accurate, consistent, and compliant outputs from the very start.

AI Integration Phase

The **AI Integration Phase** is the core of the GAutoTLF workflow, where Generative AI transforms user requests into executable code. It begins with analysis configuration, combining user prompts, ADaM metadata, and standard TLF templates to define the scope of the output. The AI engine then generates R code tailored to the requested analysis, ensuring alignment with clinical standards and organizational specifications. This code undergoes automated testing and refinement to validate functionality. A human-in-the-loop review step allows users to inspect the generated code and provide feedback or update prompts if adjustments are needed. This iterative process ensures that the final code is both technically correct and meets study-specific requirements, laying the groundwork for efficient and reliable TLF production.

Data Security and Governance

GAutoTLF is designed with security and compliance by default. For data security, only privacy-safe metadata (variable names, labels, types, high-level patterns) leaves the environment; raw patient-level data remain local. Access to the AI service is governed by API endpoint controls and token management to prevent misuse and ensure reliability.

Execution and Output Phase

The **Execution and Output Phase** is the final stage of the GAutoTLF workflow, where AI-generated code is applied to actual ADaM datasets to produce the requested outputs. Once the code passes review and refinement, it is executed locally to ensure compliance with data security requirements. These outputs are then formatted for clarity and consistency and made available for export in multiple formats, including HTML, excel, xpt (Figure 3). HTML provides a quick way to share the rendered output alongside the underlying R code for reproducibility. In addition, users can download the table as data (e.g., CSV/Excel/XPT), enabling programmatic comparisons

against SAS TLF datasets (PROC COMPARE–style checks) and seamless integration into QC pipelines.

This phase ensures that the automated workflow delivers ready-to-use outputs that meet both scientific and regulatory standards, significantly reducing turnaround time and enabling rapid decision-making.

Figure 3. Output example using CDISC Pilot Study

Analysis Results

Table 14.1.x

Demographics (Intent-To-Treat Population)

	Placebo	Xanomeline High Dose	Xanomeline Low Dose	Total
	(N=86)	(N=84)	(N=84)	(N=254)
Age				
n	86	84	84	254
Mean (SD)	75.2 (8.6)	74.4 (7.9)	75.7 (8.3)	75.1 (8.2)
Median	76.0	76.0	77.5	77.0
Min - Max	52.0 - 89.0	56.0 - 88.0	51.0 - 88.0	51.0 - 89.0
Pooled Age Group 1				
n	86	84	84	254
<65	14 (16.3%)	11 (13.1%)	8 (9.5%)	33 (13%)
>80	30 (34.9%)	18 (21.4%)	29 (34.5%)	77 (30.3%)
65-80	42 (48.8%)	55 (65.5%)	47 (56%)	144 (56.7%)
Sex				
n	86	84	84	254
F	53 (61.6%)	40 (47.6%)	50 (59.5%)	143 (56.3%)
M	33 (38.4%)	44 (52.4%)	34 (40.5%)	111 (43.7%)

AI Integration

GAutoTLF employs four specialized AI agents that work together to transform user specifications into interactive TLF code for standard or exploratory outputs based on ADaM data clinical trial data. Each agent handles a distinct phase of the code generation workflow, creating a seamless experience from initial specification to final deliverable.

The AI integration operates through a series of strategically crafted dynamic prompts sent to the LLM. These prompts combine consistent hardcoded instructions with dynamically embedded content, such as user inputs, configuration parameters, and context from previous AI-generated outputs, enabling adaptive and context-aware code generation throughout the workflow.

Shell Processor

The AI Shell Processor serves as a foundational layer of GAutoTLF, translating tabular shell specifications into structured metadata that guides subsequent code generation. This

preprocessing step establishes clear, unambiguous communication about which variables, filters, and structural table features should be incorporated into the generated code.

The shell processing workflow operates through a strategic two-stage approach. In the first stage, an initial API interaction analyzes the shell specification and assigns semantic labels to each component based on its purpose within the table structure. These labels, such as STRUCTURAL, DATASET, VARIABLE, and FILTER, etc. provide explicit context about each element's role. The AI returns this annotated information as a JSON array that preserves both the semantic meaning and the spatial coordinates (cell positions) of each component.

This JSON array is then converted into a structured data frame, creating a queryable object that can dynamically populate downstream prompts during code generation. By front-loading the shell interpretation in this manner, we effectively offload this complex parsing responsibility from the subsequent code generation stage. This architectural decision significantly reduces cognitive load on the code generation agent, minimizing the likelihood of errors, misinterpretations, or omissions that could arise from attempting to simultaneously parse shell structure and generate code.

Input Annotated Shell Example

ADSL (FASFL = Y)					
Demographics		**ARM**			
	Statistic	Arm A N = xxx	Arm B N = xxx	Total N = xxx	
Age (Years)	n	xxx	xxx	xxx	**AGE**
	Mean	xx.x	xx.x	xx.x	
	SD	x.x	x.x	x.x	
	Median	xx.x	xx.x	xx.x	
	Min	xx	xx	xx	
	Max	xxx	xxx	xxx	
Age group (Years)					**AGEGR**
<50	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
>= 50 - < 65	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
>= 65 - < 75	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
>= 75	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
Missing	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
Sex					**SEX**
Female	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	
Male	n (%)	xx (xx.x)	xx (xx.x)	xx (xx.x)	

Output Queryable Dataframe

row	*col*	*value*	*type*
1	1	ADSL	DATASET
1	1	FASFL = 'Y'	FILTER
...	...	...	...
3	1	Age (Years)	LABEL
3	5	AGE	VARIABLE
...	...	...	...

Code Generator

The AI Code Generator is the core engine of GAutoTLF, responsible for synthesizing executable R code from user specifications, shell structures, and ADaM metadata. This agent operates by integrating multiple information sources to produce context-aware, standards-compliant code. At the heart of the generation process is a backend template library which contains R code templates for standard TLF outputs commonly used across clinical study reports (CSRs).

***e.g., Demographics Table Template (via rtables package of Pharmaverse ecosystem)*:**

```
vars <- c("[analysis var 1]", "[analysis var 2]", "[analysis var 3]", "[analysis var 4]",
"[analysis var etc.]")

var_labels <- c(
  "[analysis label 1]",
  "[analysis label 2]",
  "[analysis label 3]",
  "[analysis label 4]",
  "[analysis label etc.]"
)

result <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by(var = "[treatment var]") %>%
  add_overall_col("Total") %>%
  analyze_vars(
    vars = vars,
    var_labels = var_labels
  ) %>%
  build_table(adsl)

result
```

Each template represents a proven implementation of a specific safety analysis type, such as demographic summaries, adverse event tables, etc., and serves as a starting point for customization. When a user selects a template and submits an analysis request, GAutoTLF retrieves the corresponding template code and combines it with three critical contextual inputs to be shared with the Code Generator: detailed metadata about the uploaded datasets (including variable names, labels, data types, and preview values), the processed shell structure from the AI Shell Processor, and any user-specified study-specific modifications or filtering criteria.

Example Compiled Dynamic Prompt sent to LLM

Black text: static prompt components; **Blue** text: dynamically populated components based on user inputs

The user has uploaded the following dataset(s): ADSL
Dataset Metadata:

Dataset	Variable	Label	Type	Preview
adsl	STUDYID	Study Identifier	char	CDISCPILOT01
adsl	USUBJID	Unique Subject Identifier	char	007SUBJ1234
adsl	SUBJID	Subject Identifier	char	SUBJ1234
adsl	SITEID	Study Site Identifier	char	701
adsl	SITEGR1	Pooled Site Group 1	char	701
adsl	ARM	Description of Planned Arm	char	Arm A
adsl	TRT01P	Planned Treatment for Period 01	char	Arm A
adsl	TRT01PN	Planned Treatment for Period 01 (N)	num	0

|adsl |TRT01A |Actual Treatment for Period 01 |char |Arm A |
|adsl |TRT01AN |Actual Treatment for Period 01 (N) |num |0 |
etc ...

Your objective is to generate a Demographics output code, use the following shell specification and template program as a reference point:

Shell specification:

from Shell Processor

```markdown
|                        | **Statistic** | **Arm A** | **Arm B ** | **Total ** |
|------------------------|---------------|-----------|------------|------------|
| **AGE**                |               |           |            |            |
|  Age (Years)           | n             | xxx       | xxx        | xxx        |
|                        | Mean          | xx.x      | xx.x       | xx.x       |
|                        | SD            | x.x       | xx.x       | xx.x       |
|                        | Median        | xx.x      | xx.x       | xx.x       |
|                        | Min           | x         | xx         | xx         |
|                        | Max           | xxx       | xxx        | xxx        |
| **AGEGR**              |               |           |            |            |
| Age group (Years)      | < 50          | xxx (xx.x)| xx(xx.x)   | xx (xx.x)  |
|                        | >= 50 - < 65  | x (x.x)   | xxx (x.x)  | xxx (xx.x) |
|                        | >= 65 - < 75  | x (x.x)   | xxx (x.x)  | xxx (xx.x) |
|                        | >= 75         | x (x.x)   | xxx (xx.x) | xxx (xx.x) |
|                        | Missing       | x (x.x)   | xx  (xx.x) | x (x.x)    |
| **SEX**                |               |           |            |            |
| Sex                    | Female        | xxx (xx.x)| xx (xx.x)  | xx (xx.x)  |
|                        | Male          | x (x.x)   | xxx (xx.x) | xxx (xx.x) |
|                        | Missing       | x (x.x)   | x (x.x)    | x (x.x)    |
```

=== SHELL ANNOTATIONS ===
Datasets referenced: ADSL
Table title: Demographics
Variables referenced: ARM, AGE, AGEGR, SEX
Labels referenced: Age (Years), Age Group (Years), Sex
Filters specified: FASFL == 'Y'
Statistics requested: n, Mean, SD, Min, Median, Min, Max, n (%)
=== END ANNOTATIONS ==

Template program:

=== TEMPLATE START ===

```
vars <- c("[analysis var 1]", "[analysis var 2]",
"[analysis var 3]", "[analysis var 4]", "[analysis var etc.]")
var_labels <- c(
  "[analysis label 1]",
  "[analysis label 2]",
  "[analysis label 3]",
  "[analysis label 4]",
  "[analysis label etc.]"
)
result <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by(var = "[treatment var]") %>%
  add_overall_col("Total") %>%
  analyze_vars(
    vars = vars,
    var_labels = var_labels
  ) %>%
  build_table(adsl)
result
```

=== TEMPLATE END ===

GAutoTLF constructs a comprehensive backend prompt that instructs the AI model to adapt the template code according to the provided context. For example, if the shell specifies a demographic table stratified by treatment arm with age statistics, and the user requests filtering to a specific subject population, the generator ensures that the template code is modified to incorporate the correct population flag, apply appropriate grouping variables, and calculate the requested statistics using variables confirmed to exist in the uploaded data. This enables the AI model to make intelligent decisions about variable selection, statistical methods, and output formatting based on standard clinical trial data conventions.

Importantly, the Code Generator does not operate in isolation, it produces code that is immediately executable within GAutoTLF's environment, allowing for rapid visual review and iteration. This integration of template-based structure with AI-driven customization strikes a balance between consistency and flexibility, enabling users to generate tailored analyses without sacrificing adherence to organizational template standards.

Code Assistant

The AI Code Assistant provides an interactive, conversational interface for refining generated code, offering users the ability to request modifications or seek explanations without manually editing code. This agent enhances accessibility by allowing users with varying levels of R programming expertise to iteratively improve their analyses through natural language dialogue. When a user submits a message through the chat interface, the Code Assistant first performs request classification to determine the intent: is the user asking for a code modification, requesting an explanation of how the code works, or posing a general question about the analysis? This classification step is crucial, as it dictates the assistant's response strategy and ensures that the appropriate level of technical detail is provided.

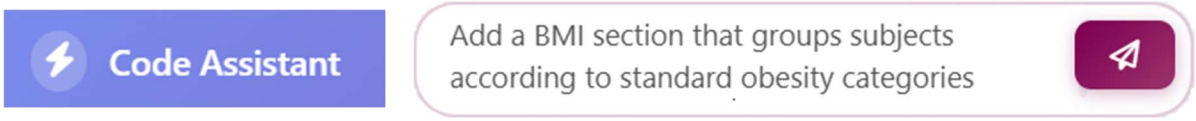
For conversational requests, such as "Can you explain this code?" or "Why are we filtering by SAFFL?", the assistant generates a clear, context-aware explanation that references the specific code in the editor. These explanations help users understand the logic behind the generated code and build confidence in the output.

When a code modification is requested, such as "Add a filter for patients over 65 years old" or "Change the grouping variable to ACTARM", the Code Assistant initiates a more sophisticated workflow. It first generates modified code based on the user's request, incorporating the current code state, dataset metadata, and conversation history to ensure continuity. Critically, the assistant does not simply return the modified code to the user; instead, it automatically executes the new code in an isolated backend environment to verify that it runs without errors.

If the code executes successfully, the assistant updates the code editor and notifies the user that the modification is complete. However, if errors are detected, the assistant enters an iterative refinement loop. It analyzes the error message, classifies the error type (such as missing variables, incorrect data types, or syntax issues), and generates a corrected version of the code. This process repeats for up to three iterations, with each attempt informed by the errors encountered in previous iterations. If the code still fails after three attempts, the assistant provides a detailed diagnostic report, including the error history and suggested next steps, empowering the user to resolve the issue manually or rephrase their request.

```r
adsl <- adsl

vars <- c("AGE", "AGEGR", "SEX”)

var_labels <- c(
  "Age (Years),”
  "Age group (Years)",
  "Sex”
)

result <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by(var = "ARM") %>%
  add_overall_col("Total") %>%
  analyze_vars(
    vars = vars,
    var_labels = var_labels
  ) %>%
  build_table(adsl %>% filter(FASFL == "Y")
)

result
```

from Code Generator



Red text: newly embedded code within code editor in response to user’s request

```r
# Create BMI category variable based on WHO classifications
adsl <- adsl %>%
  mutate(
    BMICAT = case_when(
      BMI < 18.5 ~ "Underweight",
      BMI >= 18.5 & BMI < 25 ~ "Normal weight",
      BMI >= 25 & BMI < 30 ~ "Overweight",
      BMI >= 30 & BMI < 35 ~ "Obese Class I",
      BMI >= 35 & BMI < 40 ~ "Obese Class II",
      BMI >= 40 ~ "Obese Class III",
      TRUE ~ NA_character_
    ),
    BMICAT = factor(BMICAT, levels = c(
      "Underweight",
      "Normal weight",
      "Overweight",
      "Obese Class I",
      "Obese Class II",
      "Obese Class III"
    ))
  )

# Updated variable lists
vars <- c("AGE", "AGEGR", "SEX", "BMI", "BMICAT")

var_labels <- c(
  "Age (Years)",
  "Age group (Years)",
  "Sex",
  "Body Mass Index (kg/m²)",
  "BMI Category"
```

```
)

# Build table
result <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by(var = "ARM") %>%
  add_overall_col("Total") %>%
  analyze_vars(
    vars = vars,
    var_labels = var_labels
  ) %>%
  build_table(adsl %>% filter(FASFL == "Y")

result
```

This backend testing and iterative refinement capability distinguishes GAutoTLF's Code Assistant from traditional chatbots. By proactively validating code changes before presenting them to the user, the assistant significantly reduces the likelihood of runtime errors and accelerates the path to a working solution. The conversational interface, combined with intelligent error handling, makes GAutoTLF accessible to users who may not be expert R programmers, while still providing the flexibility and control that experienced users require.

Data Explorer

To support user data exploration during analysis configuration and facilitate rapid biometrics-related insights, GAutoTLF includes an exploratory analysis and visualization capability. Unlike the aforementioned standard clinical TLF generation workflow, which relies on shell uploads and structured analysis configuration, the exploratory component enables users to query their data using natural language. Users can ask questions or request specific analyses, and the system generates executable R code to visualize or summarize the results according to the inquiry.

This feature is particularly valuable when users need to quickly answer ad-hoc questions about ADaM dataset contents without writing manual code. For example, users can rapidly determine the number of subjects with missing adverse event start dates, identify the count of serious adverse events by treatment arm, or explore other data characteristics that inform subsequent analysis decisions. By streamlining these exploratory tasks, GAutoTLF reduces the time between data receipt and actionable insights, supporting more efficient study analysis and reporting.

Integration Across Agents

Together, these AI agents form a cohesive system that addresses the lifecycle of clinical trial data analysis and TLF generation. The AI Shell Processor ensures that user specifications are accurately captured and structured, transforming tabular shell layouts into queryable metadata that guides code generation. The AI Code Generator leverages organizational standards and clinical metadata to produce tailored, standards-compliant R code that adheres to proven template structures while accommodating study-specific requirements. The AI Code Assistant enables iterative refinement through natural language interaction with built-in debugging, automatically testing code modifications and providing intelligent error resolution to ensure executable outputs. Finally, the Data Explorer empowers users to rapidly investigate their data through conversational queries, generating ad-hoc visualizations and summaries that inform analysis decisions without requiring manual coding.

Prompt Engineering Overview

In the context of GAutoTLF, the aforementioned 'Agents' are made possible through separate API calls with strategic prompt engineering and backend response parsing/ processing. More specifically, we use a combination of system prompts to coach each agent into its designated role in the context of the app, request each response to be made with a specific structure, and treat the responses as structured data or objects that can be embedded within the user interface or used as backend intermediary processing steps. This approach allows us to embed specialized, purpose-built components of the system that leverage LLM capabilities to perform specific, well-defined tasks within the clinical trial TLF generation workflow.

Results and Benefits

Pilot Performance Insights

To evaluate GAutoTLF's performance, we conducted pilot testing across four oncology studies using ADaM datasets. When all required variables were available, GAutoTLF successfully generated 94% of core outputs within approximately 3 minutes per output

The tool achieved high success rates in generating 40 core safety outputs across key domains such as study population, exposure, adverse events, laboratory results, and vital signs, as well as comprehensive listings for protocol deviations, exposure, adverse events, and individual lab measurements. The app significantly reduced turnaround time compared to traditional programming. Table 1 summarizes key performance metrics observed during validation.

Table 1. GAutoTLF Pilot Performance Metrics

Data Source	Applicable Outputs	Success Rate	Average Attempts	Average Time per output (mins)
CDISC Dummy data	24	96% (23)	1.2	3.3
Study 1	37	95% (35)	1.5	2.2
Study 2	31	94% (29)	1.5	4
Pooled Safety data	25	92% (23)	1.2	1.5
Overall		94%	1	3

Benefits

GAutoTLF delivers substantial benefits by transforming the traditional manual TLF generation process into an automated, AI-driven workflow. It significantly reduces turnaround time to minutes, improving operational efficiency and enabling rapid decision-making. By leveraging standard TLF shells, code templates, and study metadata, the tool ensures consistency and

compliance across studies while minimizing human error. Additionally, its ability to support real-time exploratory analysis empowers biometrics teams to generate insights on demand, enhancing flexibility and responsiveness. These advantages collectively position GAutoTLF as a scalable, cost-effective solution that accelerates evidence delivery and drives innovation in clinical reporting.

Future Directions and Opportunities

To further enhance GAutoTLF and expand its capabilities, several strategic improvements are planned:

1. Broaden Output Coverage
The current implementation focuses primarily on standard safety outputs. Future development will extend coverage to include **efficacy outputs** and additional domains, enabling comprehensive support for both regulatory and exploratory analyses.

2. Integration with Internal Systems
Seamless integration with key platforms will be prioritized, including Clinical Analytical and Clinical Standards Systems etc.

3. Regulatory Compliance Enablement
Transitioning toward GxP-compliant architecture will ensure adherence to regulatory standards. This includes implementing audit trails, version control, and validation frameworks to support inspection readiness and compliance.

4. Modular and Parameterized Architecture
Moving from rigid template-based generation to a parameterized function-driven approach will improve flexibility and scalability. This architecture will enable dynamic handling of complex logic and reduce dependency on hardcoded templates, and support non-standard or exploratory analyses with greater efficiency.

5. Scalable Backend and API Optimization
Performance improvements will focus on upgrading to modern large language model (LLM) APIs—such as Claude™, GPT-5™, and Gemini 3™—for better token handling, reliability, and speed.

6. Extensible Plugin System
Introducing a plugin-based architecture will allow developers to incorporate custom analysis modules or domain-specific logic without modifying the core codebase. This extensibility will foster collaboration across programming teams and ensure adaptability for diverse study requirements, ultimately enhancing scalability and innovation.

Conclusion

GAutoTLF demonstrates that AI-enabled, privacy-aware automation can meaningfully improve clinical TLF production by integrating ADaM metadata, standard shells/templates, and prompt-driven code generation within a governed R Shiny workflow. Across core safety outputs, the system reduces turnaround from hours to minutes, increases consistency via template guardrails, and preserves analytical rigor through automated validation process and

human-in-the-loop review process. While current scope centers on core safety analyses and AstraZeneca's TLF templates, the pathway to scale is clear: expand coverage, evolve toward parameterized function libraries, and evaluate next-generation models. In sum, GAutoTLF advances biometrics operations from manual programming to intelligent automation, accelerating evidence delivery while upholding quality and governance.

Acknowledgement

We would like to thank Avinash Reddy Pati for his strategic insights and guidance, which have been instrumental in shaping the direction and success of this work. We are especially grateful to Nandini Thampi, Dishank Jani, Jason Yang, and Piotr Bugajski for their technical expertise and collaborative contributions within the GAutoTLF development team. Additionally, we extend our gratitude to the AstraZeneca Oncology Biometrics leadership team, and especially to Mark Reynolds, for their sponsorship and support throughout the development of GAutoTLF. The collective commitment, innovation, and cross-functional collaboration made this AI-driven solution possible.

Contact information

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

Hao Xu, AstraZeneca Oncology Biometrics, Email: hao.xu5@astrazeneca.com
Tyler Rowsell, AstraZeneca Oncology Biometrics Email: tyler.rowsell@astrazeneca.com

References

Becker, G., Waddell, A., & Roche (2024). rtables: Reporting tables with flexible layout and formatting. R package version 0.6.7. https://CRAN.R-project.org/package=rtables

CDISC™ (Clinical Data Interchange Standards Consortium) (n.d.). SDTM/ADaM Pilot Project. GitHub™ repository. https://github.com/cdisc-org/sdtm-adam-pilot-project

OpenAI™ (2024). ChatGPT™ API: Large language model interface. https://platform.openai.com/docs/api-reference

Pharmaverse Community (2024). Pharmaverse: Curated R packages for clinical reporting. https://pharmaverse.org