PharmaSUG 2026 - Paper AI-101

How to Train Your Dragon – Embedding AI in Clinical Workflow.

Illustrated through Oncology Swimmer Plots

Sri Pavan Vemuri

Abstract

This paper presents a blueprint for successfully embedding AI into workflow using oncology swimmer plots as a working example. The framework is built on two pillars: structured prompt engineering and validation checkpoints. Prompts guide the AI step-by-step; checkpoints catch errors before they compound. The oncology visualization is the proving ground — the real contribution is a reusable methodology for any workflow where consistency and reproducibility are critical.

Introduction

Large Language Models (LLMs) have evolved into powerful, accessible tools. With foundation models and advances in natural language processing, AI capabilities—coding, analysis, creative problem-solving—are now available to professionals beyond machine learning specialists. This accessibility makes LLMs a natural fit for automating complex workflows.

However, a gap exists between capability and reliability. LLMs can hallucinate, produce inconsistent outputs, and lack domain-specific awareness. For workflows demanding precision—clinical trials, regulatory reporting, statistical programming—these limitations stall adoption.

The core insight: LLMs become reliable when you control how they're guided and how their outputs are verified.

This paper presents a framework built on two pillars:

1. **Structured Prompt Engineering —** The prompt acts as a communication bridge, conveying intent and constraining behavior. A well-designed prompt includes a clear role definition, explicit guardrails, and structured context to shape consistent, relevant outputs.

2. **Validation Checkpoints —** Layered verification catches errors before they propagate, ensuring outputs meet domain-specific standards.

Together, these pillars transform an LLM from a capable-but-unpredictable tool into a dependable automation engine. To demonstrate this approach, we apply the framework to generating oncology swimmer plots for clinical trials. But the visualization is just the example—the real contribution is the blueprint: a reusable methodology for embedding AI into any workflow where reliability is non-negotiable.

1. System Architecture

The system is built on a clear division of labour: Python does what Python is good at, and AI does what AI is good at. Python handles all structural validation directly — no API call, no latency, no hallucination risk. AI is called only where natural language understanding is genuinely needed. Every AI call is preceded by Python guardrails that validate inputs and enforce structure, so errors are caught before the model is ever invoked.

The workflow moves through five sequential steps:

1. **Step 1 — Web Interface:** A Dash web app (dash_app.py) loads CDISC datasets and presents them to the user. Python only.
2. **Step 2 — Data Manipulation:** The user describes changes in plain English. The app derives columns, filters rows, and merges ADSL if needed. AI is called here (data_customizer.py) to translate the instruction into pandas code.
3. **Step 3 — Graph Generation:** The user selects variables and graph options. Data is validated, then AI generates the swimmer plot code (graph_generator.py). A built-in debug loop fixes errors automatically.
4. **Step 4 — Interactive Customization:** The user refines the plot through natural language chat. AI applies each change while preserving the plot structure (graph_customizer.py).
5. **Step 5 — Code Conversion:** The final Python code is converted to R or SAS for downstream use (code_converter.py). AI called here.

1.1 File Structure

All files live in the same project folder. Python finds them automatically — no path configuration needed.

1. **dash_app.py** — Web interface. The only file the user interacts with directly. Python only.
2. **code_generator.py** — Orchestrator. Wires all modules together via the SwimmerPlotGenerator class. Python only.
3. **data_utils.py** — Loads ADSL and ADRS CSV files from disk at startup. Python only.
4. **data_validator.py** — Validates X, Y, and HBAR types and checks required variables are present. Python only.
5. **data_customizer.py** — Step 2. Translates plain-English instructions into pandas code. AI called here.
6. **graph_generator.py** — Step 3. Generates the swimmer plot code and handles debug. AI called here.
7. **graph_customizer.py** — Step 4. Applies iterative plot refinements via chat. AI called here.
8. **code_converter.py** — Step 5. Converts Python code to R or SAS. AI called here.
9. **utils.py** — Shared helpers used by all modules: calls the AI API, strips code fences, saves files. Python only.

1.1.1 Step 1 — Web Interface

File: dash_app.py. Python only.

Dash was chosen for its native Plotly integration and callback-driven reactive model, which makes it ideal for a multi-step clinical workflow. Four tabs guide the user through the process in order, handing off data automatically between steps. At startup, ADSL and ADRS are loaded from disk (data_utils.py) and stored in memory. All data between steps is passed through in-memory browser stores (dcc.Store), so nothing is held on the server between callbacks.

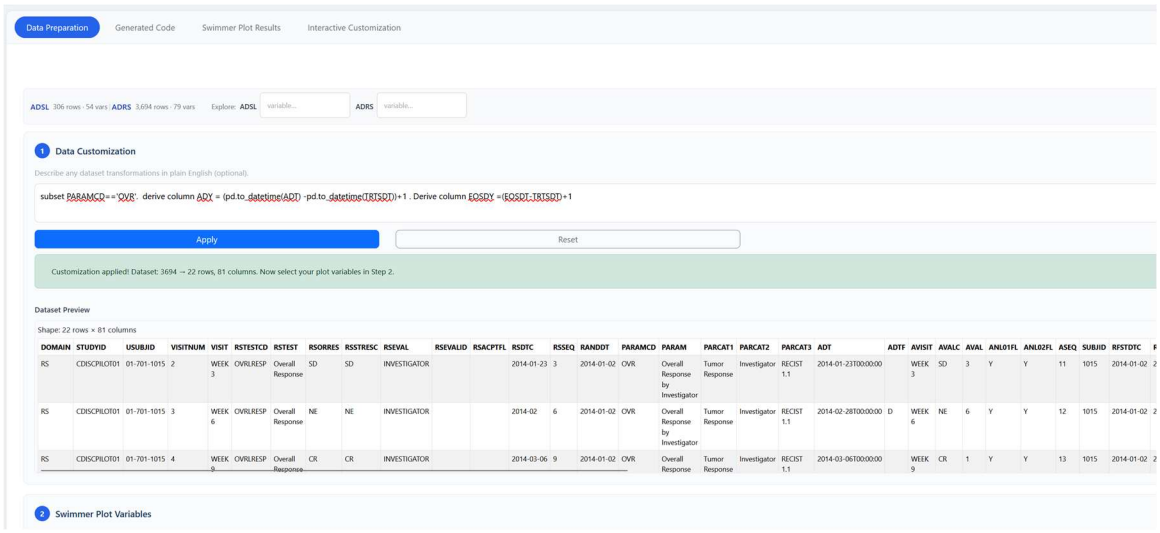

Fig 1. Landing page showing initial data load, customization, and tabbed navigation

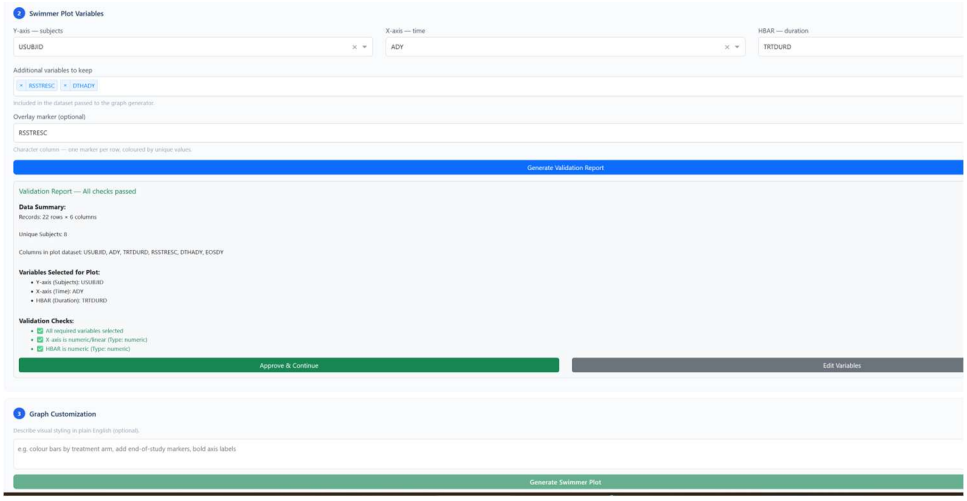

Fig 2. Variable selection and graph customization sections

1.1.2 Step 2 — Data Manipulation

File: data_customizer.py. AI called here.

The user describes data changes in plain English — filter rows, derive a new column, merge a variable from ADSL. data_customizer.py sends the instruction to Claude and executes the returned pandas code against the dataset. All original, merged, and derived columns are passed to the next step so the user can select from them. Fig 3 shows this step.

After manipulation, data_validator.py (Python only) runs type checks — X-axis numeric or datetime, HBAR numeric, Y-axis a recognised subject identifier. Errors block progression; warnings allow it. The user must click Approve before moving to Step 3. Fig 3 illustrates this.

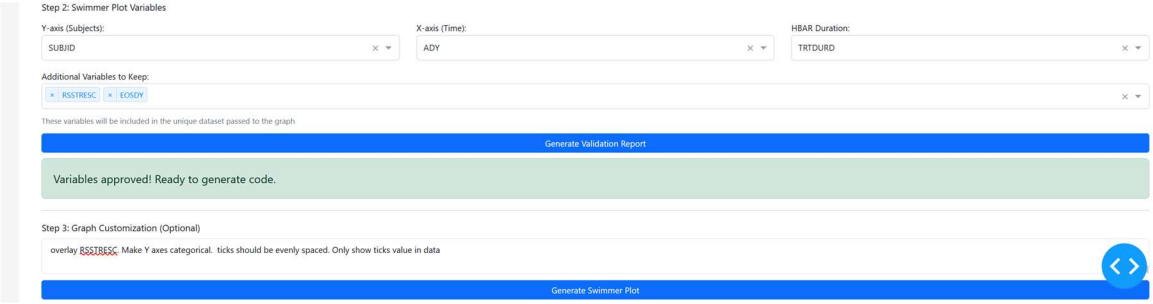

Fig 3. Data manipulation step showing variables selected and changes requested

Reactive State Management: The system tracks data transformations through reactive dcc.Store components that automatically update dependent UI elements. The original ADRS data is held in a read-only store; all customization writes to a separate customized-data store. Resetting restores the original state without reloading from disk.

ADSL Merge Strategy: The data customizer always begins by performing a LEFT JOIN of ADSL onto the full ADRS dataset using USUBJID as the join key. This is hardcoded in the starter skeleton injected into every AI prompt, not left to the AI to decide. The skeleton deduplicates ADSL to one row per USUBJID before merging, selects only the columns needed, and asserts that the row count cannot increase after the join. The AI fills in only the derivations, filters, and calculations within this safe structure.

Data Structure Validation: Before proceeding to code generation, the system validates that swimmer plot requirements are met. Errors are shown in red with the Approve button disabled; warnings in amber with the button still enabled. Code generation is blocked until the user clicks Approve.

1.1.3 Step 3 — Graph Generation

File: graph_generator.py. AI called here.

Once data is validated and approved, graph_generator.py builds a structured prompt and sends it to Claude. The prompt injects a fixed starter skeleton — horizontal bar trace, axis types, sort order — before the user's customization request. Claude can only add to this skeleton; it cannot remove the core structure. If the generated code fails when executed, Python captures the full error and sends it back to Claude with the same invariant-enforcing prompt. The corrected code replaces the original automatically. The debug button only appears when there is an error. Fig 4 and 5 show these steps.

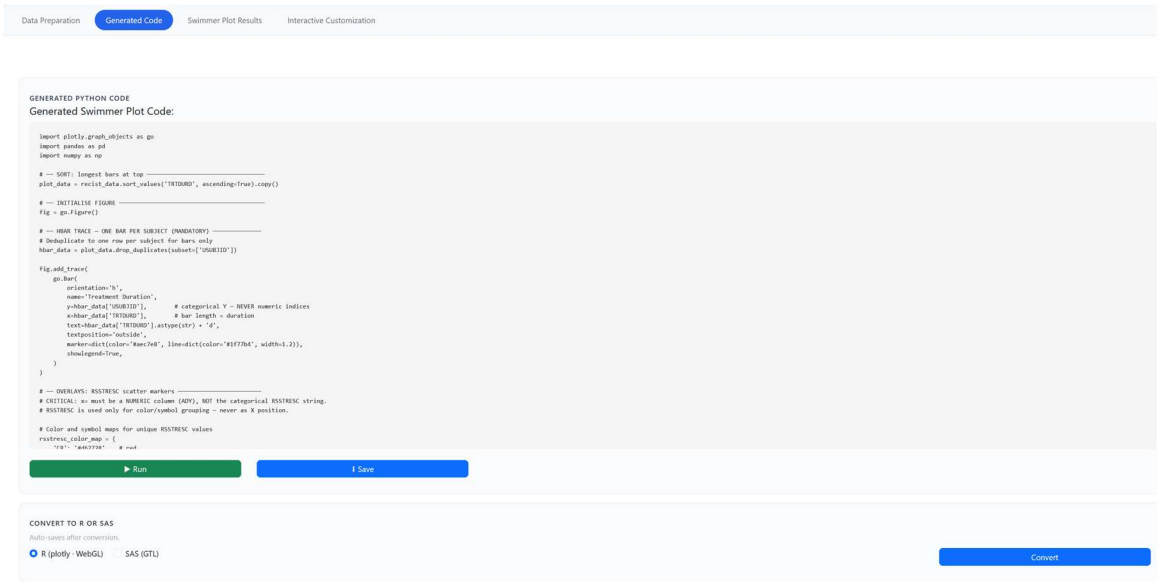

Fig 4. Code generation with built-in debug feature (debug button only appears on error)

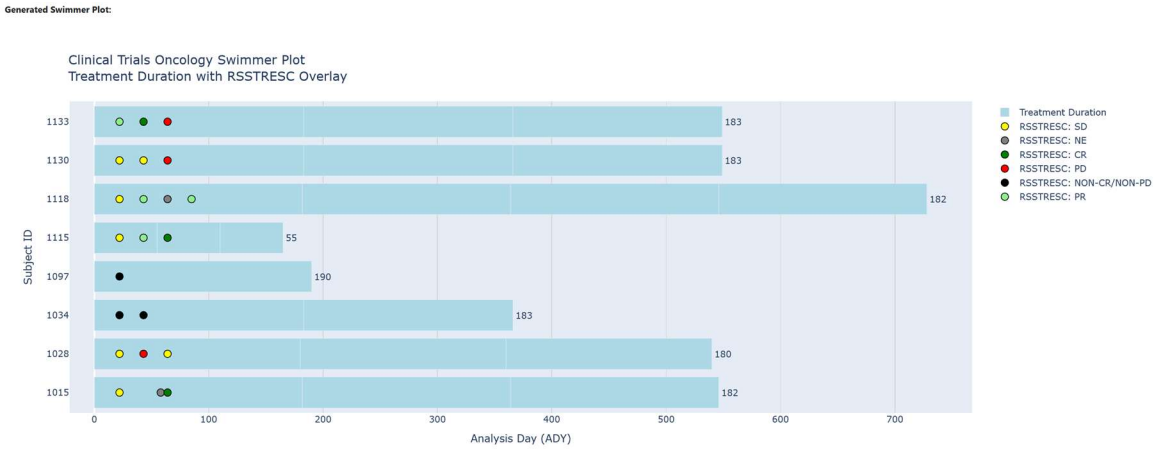

Fig 5. Initial plot after successful code generation

1.1.4 Step 4 — Interactive Customization

File: graph_customizer.py. AI called here.

After the plot is generated, the user can refine it through natural language chat. Each message is sent to Claude along with the current plot code and the last six messages as context. Claude modifies only what was requested and returns the full updated code. The code state is saved after every exchange. After Claude returns modified plot code, a Python regex check scans the code for any column references and validates them against the dataset metadata before accepting the change. If an unknown column is found, the code is discarded, the plot is restored

to its previous state, and the user is prompted to verify the column name.

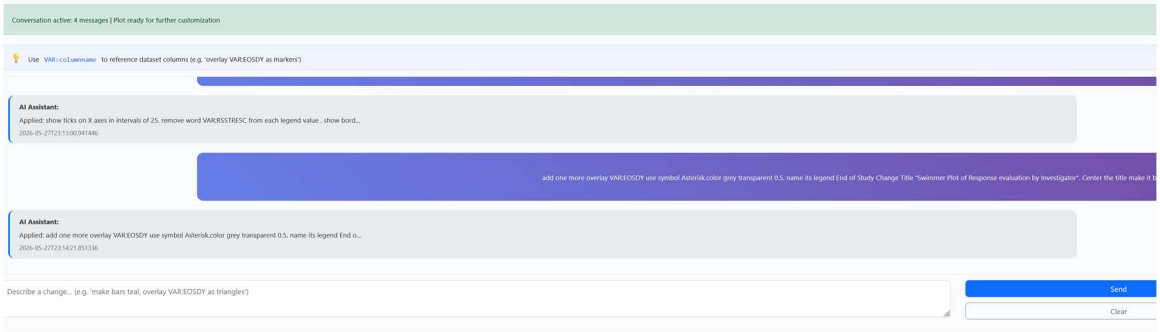

Fig 6. Interactive customization chat. Each request updates and preserves the code state.

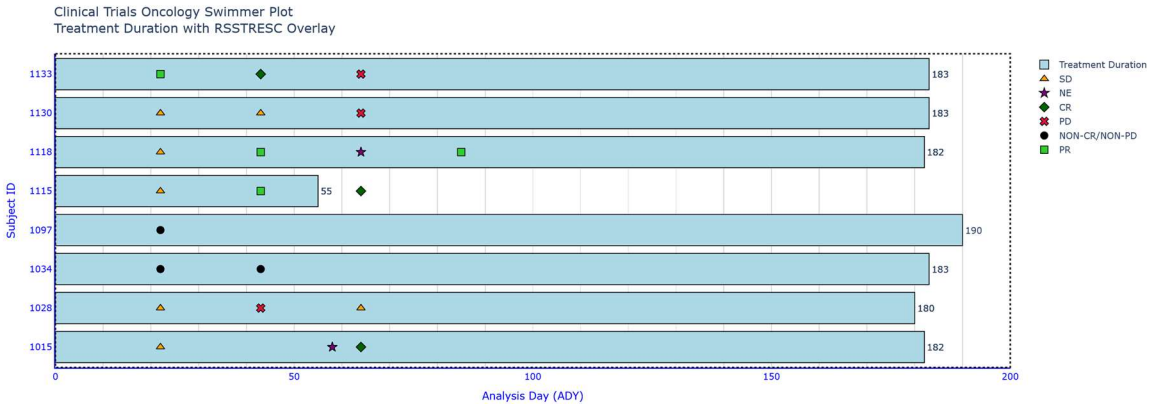

Fig 7. Final graph after AI plot customization

1.1.5 Step 5 — Code Conversion

File: code_converter.py. AI called here.

Once the plot is finalized, the Python code can be converted to R or SAS. code_converter.py sends the Python code to Claude with language-specific requirements and the converted code is returned and auto-saved.

R output: Uses plotly directly with plot_ly() and add_trace(). Horizontal bars use type='bar' with orientation='h'. Scatter overlays use scattergl for WebGL performance.

SAS output: Uses GTL (Graph Template Language) with PROC TEMPLATE to define the layout and PROC SGRENDER to render against the data.

2. Appendix

2.1. Technology Stack

Dash: Component-based reactive web framework. Multi-tab layout, callback-driven state management, native Plotly integration.

Anthropic Claude AI: Large language model used for all AI steps. Model: claude-sonnet-4-6.

Plotly: Renders the swimmer plot as interactive HTML embedded directly in the browser tab.

Python: Handles all validation, state management via dcc.Store, and error capture around every AI call.

2.1.2 Github_code and References

- https://github.com/sripavanv/Coding_agent_PharmaSUG2026
- https://research.google/pubs/attention-is-all-you-need/
- https://www.coursera.org/learn/develop-generative-ai-applications-get-started
- https://www.edx.org/learn/python/harvard-university-cs50-s-introduction-to-programming-with-python
- https://www.anthropic.com/learn

Conclusion

The reliability of this system comes from a clear division of labour: Python defines the boundaries through direct validation and structured prompts, and AI operates creatively within them. Neither alone would be sufficient — Python cannot interpret free-text instructions, and AI alone cannot guarantee structural consistency. Together, they produce results neither could achieve independently.

Future work will expand to additional oncology visualization types and extend compatibility to more clinical data standards. The success of this implementation demonstrates the potential for AI-assisted tools to enhance productivity and accessibility in pharmaceutical data analysis.

Contact Information

Your comments and questions are valued and encouraged:

Sri Pavan Vemuri, sripavanv@gmail.com.