PharmaSUG 2026 - Paper ET-416

It's a Wonderful Lifecycle: Translating Statistical Programming into Modern Analytics Development

Steven Nicholas, Atorus Research

ABSTRACT

Emerging technologies such as Shiny and open-source analytics frameworks are increasingly being introduced into clinical programming environments that have long relied on established statistical programming processes. While the tools may be new, many of the principles required for successful adoption are already familiar.

This paper reframes modern analytics applications through a direct comparison of the software development lifecycle (SDLC) and traditional statistical programming workflows. Using a one-to-one mapping, it translates artifacts such as the Statistical Analysis Plan (SAP), mock shells, ADaM specifications, dry runs, and QC programming into analogous stages of analytics application development. User stories and feature requirements serve a role similar to the SAP, while wireframes function as the equivalent of mock shells, enabling early alignment on purpose, scope, and decision making before development begins. Iterative development cycles mirror dry runs and interim outputs, and testing phases align naturally with independent QC programming and validation practices long embedded in regulated clinical workflows.

Through real world examples of enterprise analytics applications, this paper demonstrates how applying a disciplined lifecycle-based approach leads to more sustainable, transparent, and scalable tools. By viewing pharma deliverables through a software lens, we can apply the same trusted practices to this new class of deliverable.

INTRODUCTION: THE PROCESS OVER THE TOOL

We are currently witnessing a renaissance in clinical analytics. The transition from static outputs to interactive applications is a significant shift in how we deliver data, but the why remains unchanged. As emerging technologies become standard, the success of these tools depends not just on the quality of the code, but on the discipline of the lifecycle used to build them. To build sustainable, enterprise-grade applications, we don't need to reinvent the wheel; we need to recognize that the rigorous standards we have honed for decades are exactly what software development requires.

Historically, the "finish line" for a clinical programmer was the delivery of static TFLs, reliable snapshots that fulfilled a fixed Statistical Analysis Plan (SAP). However, a static report is a closed system. In a recent safety review, a stakeholder identified an emerging trend and needed to instantly pivot from a high-level summary to a granular, patient-level view filtered by specific parameters. In a traditional workflow, this request would trigger a new programming cycle and a multi-day wait. By applying a lifecycle-driven model, rather than just a new tool, we provided an environment where that "drill-down" was built-in and validated from the start. This demonstrates that the shift to interactive analytics is not a departure from our roots, but an evolution in how we translate clinical requirements into actionable insights. The sections that follow walk through each stage of that lifecycle, one-to-one.

THE MAPPING: STATISTICAL PROGRAMMING AS SDLC

The Software Development Lifecycle (SDLC) is a structured process that guides the planning, design, development, testing, and delivery of software applications. The core of this transition lies in a simple recognition: the Software Development Lifecycle (SDLC) is the same discipline we have always practiced, just under a different name. This lifecycle applies end-to-end, from raw data through SDTM and ADaM to final outputs.

Traditional Statistical Programming	Modern Analytics Development (SDLC)	Core Philosophy
Statistical Analysis Plan (SAP)	User Stories & Requirements	Defining the who, what, and why
Mock Shells	Wireframes & Prototypes	Agreeing on the visual contract
SDTM & ADaM Specifications	Technical Specifications & Data Models	Ensuring data traceability and logic
Dry Runs / Interim Analysis	Iterative Review Cycles	Refining through stakeholder interaction
Independent QC Programming	Code Review, Automated Testing & Validation	Maintaining regulatory integrity

SAP TO USER STORIES: DEFINING THE WHO, WHAT, AND WHY

The Statistical Analysis Plan is the foundation of every clinical programming effort. It defines the scope, the endpoints, the populations, and the logic before a single line of code is written. In analytics application development, User Stories and Feature Requirements serve exactly the same purpose.

Where the SAP asks, "what analyses need to be produced and how," user stories ask, "who needs this information, what do they need to do with it, and why does it matter?" The form is different, but the discipline is identical: both force alignment on scope and intent before development begins, and both serve as the reference point when decisions need to be made mid-build.

This translation becomes concrete with a direct example. A standard SAP reads: "The number and percentage of subjects with TEAEs will be summarized by MedDRA-coded SOCs and PTs. Within each summary, a subject will only be counted once within each SOC and PT." The user story equivalent reads: "As a Medical Monitor, I want to filter adverse events by system organ class and onset date so that I can identify emerging safety signals without requesting a new TFL." Same clinical intent, different form. Both define the who, what, and why before a single line of code is written.

MOCK SHELLS TO WIREFRAMES: AGREEING ON THE WHAT

The mock shell is a staple of clinical programming because it provides a visual contract before development begins. In SDLC, this is the Wireframe. The underlying principle is identical: early visual agreement is critical to success.

Wireframing enables fast feedback cycles before any real investment has been made. Because nothing has been built yet, stakeholders and designers can freely say "I don't like this" without the psychological weight of sunk cost. A designer can present multiple layout options in the same session, the team can reject all of them, and no one has lost anything but an afternoon. That freedom disappears the moment development begins. Once a team has spent weeks building something, people become attached to it, not because it is the right solution, but because effort has been invested in it. The wireframe stage is specifically designed to prevent that attachment from forming before the right solution has been found.

The cost of skipping this step is real. In one data quality review application, a development team spent three weeks building out core functionality before key stakeholders communicated their actual requirements: a specific landing page layout, precise placement and shape of menu filters, particular visual styling of selection controls, and, critically, data listings surfaced on hover rather than housed on a separate panel (screen). None of these were minor cosmetic preferences; each had downstream implications for data architecture and navigation logic. A single wireframe session before development began would have surfaced all of it. It is the bridge between a requirement and a reality, ensuring we agree on the "what" before we invest heavily in the "how."

ADAM SPECS TO TECHNICAL SPECIFICATIONS AND DATA MODELS: TRACEABILITY AS A THREAD

Traceability is a fundamental pillar of clinical data science. In traditional programming, the ADaM Specification serves as the vital link between raw data and the final analysis. This concept translates directly to Technical Specifications and Data Models in application development.

Just as we wouldn't program a complex ADaM dataset without a spec that defines derivations and metadata, we cannot build a robust application without documentation that maps how data flows from its source to the user interface. Modern analytics applications often draw from multiple data sources simultaneously, with transformations required at each stage before data is ever surfaced to the end user. That entire pipeline must be mapped explicitly. Define.xml, which combines dataset specifications and metadata into a single submission artifact, maps naturally to the data dictionaries and schemas that serve the same purpose in application development.

The stakes are higher still when an application has write access, sending data back to an EDC system for example, because the flow is no longer unidirectional. Every transformation, every decision point, and every system interaction becomes part of the audit trail. Traceability here is not just good documentation practice; it is an auditable process.

The regulatory reviewer who expects a clear audit trail from raw data to TFL output will expect the same from an interactive application. A well-maintained technical specification is what makes that possible, and it is the thread that keeps these tools compliant.

DRY RUNS TO ITERATIVE DELIVERY

Software thrives on iterative cycles, which is a natural extension of the Dry Run and Interim Analysis mindset. Just as interim analyses are scheduled checkpoints that allow a trial team to assess whether the data is telling the right story before locking, iterative review cycles give a development team structured opportunities to assess whether the

application is solving the right problem before release. The principle is identical: don't wait until the end to find out something needs to change.

The value of this becomes concrete in practice. During a mid-cycle review of a safety monitoring application, a stakeholder flagged that outlier values were stretching the x-axis of a key plot to the point where the bulk of the data was compressed and nearly unreadable. Adding axis filter controls and a zoom capability was a manageable sprint (a defined iteration cycle in Agile development) of work when caught at that stage. Caught at final delivery, it would have required rearchitecting the visualization layer.

This discovery also illustrates a governance reality: not every finding mid-cycle belongs in the current release. This is where the mock shell earns its place as a living document throughout the development lifecycle, not just at the start. Revisiting the wireframe when new requirements surface creates a documented record of what was deferred and what belongs in a future phase, preventing scope creep without losing good ideas.

INDEPENDENT QC PROGRAMMING TO AUTOMATED TESTING: MAINTAINING REGULATORY INTEGRITY

Independent QC programming is one of the most deeply ingrained practices in clinical programming. The principle is straightforward: a second programmer, working independently from the original, verifies that the output is correct. It is not a formality; it is the mechanism by which we catch what familiarity blinds us to.

Automated testing applies the same principle to application development. Rather than a second programmer validating a final output, automated tests continuously verify that the application behaves exactly as the user stories said it would. Every time a new feature is added, or an existing one is modified, the test suite runs and confirms that nothing previously validated has broken. It is independent verification, applied earlier and continuously rather than once at the end.

Code review extends this principle further. Just as a second programmer works independently to verify output, a second developer reviews code before it is merged into the application, catching errors, enforcing standards, and ensuring the logic matches the intent of the technical specification. It is the double-programming instinct applied at the source rather than the output.

This matters especially as applications evolve across phases. The mid-cycle discovery described in the previous section, a zoom capability added to an existing visualization, represents a change to validated logic. Without automated testing, there is no systematic way to confirm that adding a feature did not introduce an error somewhere else in the application. With it, the double-programming instinct we have always relied on becomes a built-in safeguard rather than a manual step.

CONCLUSION: EXPERIENCE IN AN ACCELERATED ERA

The workflows emerging around analytics tools may come with new documentation names, new acronyms, and new technology, but the lifecycle underneath them is one we have been practicing for years. The SDLC is not foreign to clinical programmers; it is already in our DNA.

As interactive analytics tools become standard, the boundaries between traditional statistical programming and software development are increasingly blurred. Roles are converging, and the people filling them need to speak both languages. The parallels outlined in this paper are not just conceptual; they are practical tools for navigating that convergence.

By embracing the commonalities between clinical programming and software development, we can navigate this transition with confidence. By sticking to our roots, emphasizing design intent, traceability, and rigorous validation, we ensure that as we move faster, we continue to deliver the quality that clinical trials demand. Whatever our backgrounds, we share the same goal: to improve our tools and processes so we can bring treatments to patients in need, with speed and quality. Our experience is not a constraint; it is the anchor of excellence in an increasingly rapid digital landscape.

CONTACT INFORMATION

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

Steven Nicholas
Atorus Research
Steve.nicholas@atorusresearch.com
www.Linkedin.com/in/steventnicholas