Paper ML02

Interactive and Instant: GenAI-driven Creation of Clinical Trial Applications

Marcin Dubel, Appsilon, Warsaw, Poland

ABSTRACT

This paper presents TealFlow, a generative AI-driven platform designed to accelerate the creation of clinical trial applications. As the pharmaceutical industry faces growing demands for faster data analysis, reporting, and accelerated speed to market, traditional development approaches struggle to keep pace. TealFlow addresses this challenge by providing a low-code, chat-based interface that enables non-developers to build custom data analysis tools using the {teal} framework in minutes rather than days. The platform leverages a multi-layered technical architecture combining specialized AI agents, the Model Context Protocol (MCP) for tool integration, containerized execution environments, and the open-source {teal} R/Shiny ecosystem. This paper discusses current market trends—including the critical need for faster delivery of medicines to patients—the challenges of applying generic AI solutions to specialized clinical data workflows, provides a deep technical analysis of the TealFlow architecture and its MCP-based implementation, and demonstrates how TealFlow bridges the gap between agentic AI capabilities and accessible user experiences.

INTRODUCTION

The landscape of clinical data application development is undergoing significant transformation. Organizations increasingly demand modular approaches to building internal data analysis tools with reusable components. Predefined solutions and tested building blocks for common problems have proven to accelerate delivery while maintaining quality standards. However, usability remains a persistent challenge—despite UI wizards and visual tools, current solutions still require coding knowledge, limiting accessibility for many stakeholders.

The pharmaceutical industry operates under extraordinary pressure. Bringing a new drug to market takes an average of 10 to 15 years and costs upwards of $2.6 billion. Clinical trial data analysis represents a critical bottleneck in this pipeline: every day of delay in producing trial readouts translates directly into delayed patient access to potentially life-saving therapies. The demand for faster, more accessible analytical tooling is not merely a matter of efficiency—it is fundamentally a question of patient welfare.

Simultaneously, the rise of generative AI has created new possibilities for automating complex software development tasks. Large language models (LLMs) can now generate, debug, and refine code across a wide range of programming languages and frameworks. However, applying these capabilities to the highly specialized domain of clinical trial data analysis requires overcoming significant technical and organizational challenges.

We believe that within one year, the majority of reporting and data analytics will be produced automatically and much faster than current methods allow. This paper explores how generative AI can help realize this vision while addressing the unique challenges of the clinical trial domain. We present TealFlow's technical architecture in detail, describe the iterative design process that led to the current solution, and discuss the open-source TealFlow MCP component that enables broader community adoption.

MARKET TRENDS AND NEEDS

The demand for faster clinical trial data analysis continues to grow across the pharmaceutical industry. Organizations seek to analyze trial data within days of data availability rather than weeks or months. This acceleration serves multiple critical purposes that span the entire drug development lifecycle.

Empowering Users with Rapid Insights

Clinical data scientists and biostatisticians need rapid access to interactive exploratory tools to understand emerging patterns in trial data. Traditional workflows require submitting requests to development teams, waiting for application builds, and then iterating through multiple rounds of revision. This cycle can take weeks, during which critical decisions about trial conduct, safety monitoring, and endpoint analysis are delayed. By enabling domain experts to create their own analytical applications through natural language interfaces, TealFlow eliminates this bottleneck and puts the power of data exploration directly in the hands of those who understand the data best.

Increasing Productivity of Subject Matter Experts

Highly specialized subject matter experts (SMEs) represent a scarce and valuable resource within pharmaceutical organizations. Their deep understanding of therapeutic areas, regulatory requirements, and statistical methodologies makes them essential to the drug development process. However, much of their time is currently spent on activities that do not require their specialized knowledge—configuring software tools, writing boilerplate code, and managing technical infrastructure. AI-assisted application development frees these experts

to focus on what they do best: interpreting data, designing analyses, and making scientifically informed decisions about trial outcomes.

Accelerated Speed to Market and Faster Delivery of Medicines to Patients

Perhaps the most compelling motivation for AI-driven clinical trial application development is the potential to accelerate the overall speed to market for new therapies. The faster that clinical data can be analyzed, visualized, and reported, the sooner regulatory submissions can be prepared and filed. Every week saved in the analytical phase translates directly into earlier patient access to new treatments.

The clinical trial process involves multiple stages where rapid data turnaround is essential: interim analyses that inform go/no-go decisions, safety monitoring that protects patient welfare, and final analyses that support regulatory filings. In each of these contexts, the ability to quickly generate interactive analytical applications can compress timelines significantly. When a biostatistician can create a survival analysis dashboard in minutes rather than requesting one from a development team and waiting days or weeks, the cumulative time savings across an entire clinical program can amount to months of accelerated development.

Furthermore, regulatory agencies worldwide are encouraging the adoption of modern analytical tools and open-source technologies. The FDA has expressed support for R-based submissions, and industry initiatives such as the pharmaverse consortium are building standardized, validated toolkits for clinical data analysis. TealFlow aligns with these trends by generating applications built on the {teal} framework, which is already widely adopted and trusted within the regulatory ecosystem.

Current Constraints in Clinical Data Application Development

Current approaches to clinical data application development face several constraints. While modular app building frameworks exist, they typically require significant technical expertise. Even when predefined solutions accelerate delivery, the complexity of implementation limits who can effectively use these tools. The result is a bottleneck where valuable SME time is spent waiting for developer resources rather than conducting analysis. The industry requires a paradigm shift from developer-centric to user-centric tooling, and generative AI provides the foundation for making this shift possible.

CHALLENGES WITH GENERIC AI SOLUTIONS

Applying generative AI to clinical data application development presents unique challenges that generic AI solutions struggle to address. While general-purpose LLMs have demonstrated impressive code generation capabilities, the clinical trial domain imposes requirements that expose the limitations of off-the-shelf AI approaches.

Lack of Specialized Framework Knowledge

AI systems cannot work out-of-the-box with specialized frameworks and tools like {teal}, which is widely used in pharmaceutical research. The {teal} framework, maintained by the insightsengineering community (led by Roche/Genentech), provides a comprehensive ecosystem for building interactive clinical data exploration applications in R/Shiny. It includes domain-specific module structures (teal.modules.clinical, teal.modules.general), a sophisticated data filtering facility, and specific conventions for module registration, server function signatures, and data connector patterns.

General-purpose AI models have limited exposure to {teal} in their training data compared to mainstream web frameworks. They frequently generate syntactically valid but functionally incorrect code that fails to follow {teal}'s specific patterns for module initialization, data handling, and UI composition. For example, {teal} requires modules to accept specific data arguments through a filtered_data object, uses a particular pattern for registering modules with the main application, and expects specific function signatures that differ from standard Shiny module conventions.

Internal Frameworks and Organizational Policies

Generic AI solutions encounter significant obstacles when dealing with domain-specific implementations. Each pharmaceutical organization may have custom {teal} modules tailored to their specific therapeutic areas, coding standards that reflect regulatory requirements (such as 21 CFR Part 11 compliance), and internal validation protocols that must be followed for any code deployed in a GxP context. These organizational constraints fall outside the scope of general AI training data and cannot be addressed by fine-tuning alone.

Additionally, organizations often maintain proprietary data connectors, custom visualization libraries, and internal package repositories that must be integrated into any generated application. A practical AI solution must be configurable to understand and respect these organizational boundaries while still providing useful code generation capabilities.

Usability Barriers for Non-Technical Users

Agentic AI approaches, while powerful, typically require familiarity with development tools and pre-configured working environments. Tools like GitHub Copilot, Cursor, and Claude Code operate within integrated development environments (IDEs) that assume a level of technical proficiency. This creates a barrier for the very users who would benefit most from AI-assisted application development—the domain experts who understand clinical trial requirements but may not have programming backgrounds.

The challenge is not simply one of interface design; it requires rethinking the entire interaction model. Non-technical users need to express their analytical requirements in domain-specific language ("I need a Kaplan-Meier plot for overall survival stratified by treatment arm") rather than technical specifications. The AI system must translate these high-level requirements into specific {teal} module configurations, appropriate statistical parameters, and valid R/Shiny application code.

TEALFLOW: SYSTEM ARCHITECTURE AND TECHNICAL IMPLEMENTATION

TealFlow's architecture is designed to address each of the challenges described above through a layered system that combines specialized AI capabilities with domain-specific tooling and an accessible user interface. This section provides a detailed technical analysis of the platform's key architectural components, drawing on the concrete implementation of the system.

High-Level Architecture Overview

The TealFlow platform consists of four primary layers that work together to transform natural language requests into production-ready clinical trial applications:

The first layer is the **User Interface Layer**, which provides a browser-based chat interface accessible to non-technical users. This layer handles conversation management, session persistence, and application preview rendering. Users interact exclusively through natural language, with the system providing visual feedback as the application takes shape.

The second layer is the **AI Agent Layer**, which processes user intent, analyzes uploaded data schemas, and orchestrates the application generation workflow. This layer employs specialized prompting strategies and maintains context about the {teal} framework, available modules, and clinical data conventions.

The third layer is the **Tool Integration Layer**, built on the Model Context Protocol (MCP). This layer provides the AI agent with structured access to {teal} documentation, module catalogs, code validation utilities, and deployment services. The MCP integration enables the agent to perform actions beyond simple text generation—it can query live documentation, run code tests, and validate generated applications.

The fourth layer is the **Execution Environment**, a containerized runtime that provides isolated, secure sandboxes for code generation, testing, and application preview. Each user session operates within its own container, ensuring data isolation and preventing cross-session interference.

The {teal} Framework Foundation

At the core of TealFlow's domain specificity is the {teal} framework. Developed and maintained by the insightsengineering community, {teal} is an open-source R/Shiny framework purpose-built for exploratory analysis of clinical trial data. It provides a modular architecture where individual analysis components (modules) can be composed into complete applications through a declarative configuration.

Key {teal} capabilities that TealFlow leverages include: a dynamic filtering facility that allows users to subset data interactively across all modules simultaneously; a reproducible code generation system that records every analytical step for audit and validation purposes; a module ecosystem spanning common clinical analyses including demographics tables, adverse event summaries, Kaplan-Meier survival plots, forest plots, response analysis, and patient profiles; and integration with the broader pharmaverse ecosystem of validated R packages for clinical data analysis.

TealFlow's AI agent is specifically trained to generate code that adheres to {teal}'s conventions. This includes proper module registration using the `modules()` function, correct data argument passing through the `teal_data()` interface, appropriate use of the `tm_*()` family of module constructors, and valid integration with {teal}'s reporter module for generating exportable reports.

In production, the platform ships with pre-loaded CDISC ADaM datasets (ADSL, ADTTE, ADRS, ADQS, ADAE) as RDS files in the workspace directory. The generated application code references these datasets using `teal_data()` with standard CDISC join keys, ensuring that the resulting app properly relates subject-level data (ADSL) to domain-specific analysis datasets through `default_cdisc_join_keys`. This convention-over-configuration approach dramatically reduces the surface area for code generation errors.

Multi-Agent Architecture

Rather than relying on a single monolithic AI model, TealFlow implements a **polymorphic agent system** where each supported framework is handled by a specialized agent class. All agents conform to a common abstract interface defined by the `Agent` base class, which enforces two properties: a `system_prompt` and a `type` identifier.

The platform currently supports the following agents:

- **TealAgent** — the primary agent for clinical trial applications, configured with {teal}-specific prompting
- **RShinyAgent** — for general-purpose R/Shiny applications (e.g., using bslib for layout)
- **PyShinyAgent** — for Python-based Shiny applications, with extensive guardrails against common Shiny-for-Python anti-patterns
- **QuartoAgent** — for generating Quarto reports combining narrative and executable Python code
- **RShinyModuleAgent** — for generating reusable R/Shiny module components with the `_flow` naming convention
- **MarkdownAgent** — for structured documents and non-interactive content

Each agent carries its own system prompt assembled from a **template-plus-specialization** pattern. A shared `app_prompt.md` template defines the general code generation structure and response format, while language-specific instruction files (e.g., `app_prompt_r.md`, `app_prompt_py.md`, `app_prompt_teal.md`) inject framework-specific rules. For the Python agent, these rules are particularly detailed—covering deprecated API patterns (`@output` decorator removal), correct import paths for `reactive` (never from `shiny.express`), proper sidebar layout functions, and naming collision avoidance between reactive values and render functions. This layered prompt architecture ensures that each agent generates code following the precise conventions of its target framework.
The agent is selected at session start through a frontend `AgentPicker` component. Users choose from a dropdown (Teal, Quarto, Shiny for R, Shiny for Python, Shiny for R Module), and the selection propagates to the backend via Shiny input binding (`agent_type`). The server resolves this to the corresponding agent instance from a pre-initialized dictionary, ensuring zero cold-start latency.

Claude Code SDK Integration via Subprocess Isolation

The AI backbone of TealFlow is the **Claude Code SDK**, integrated through a custom subprocess isolation architecture. This design decision addresses a fundamental technical incompatibility: the Claude Code SDK uses the AnyIO async runtime, which conflicts with Shiny's asyncio event loop. Running both in the same process causes deadlocks and race conditions.
The solution is a two-process architecture implemented in `ClaudeCodeManager`:

Main Process (Shiny application):

The `ClaudeCodeManager` class extends `LLMManager` and manages session state, prompt preparation, and response processing. When a user sends a message, the manager:
1. Prepares the prompt by extracting the latest message from the conversation history
2. Resolves the working directory to a per-chat workspace (`working_workspace/<chat_id>`)
3. Serializes the request (prompt, session ID, workspace path) as JSON
4. Spawns a Python subprocess running `claude_subprocess_runner.py`
5. Streams subprocess stdout line-by-line, deserializing each response

Subprocess (Claude Code SDK):

The `claude_subprocess_runner.py` script runs in complete isolation with its own AnyIO event loop. It:
1. Reads the JSON request from stdin
2. Initializes `ClaudeCodeOptions` with the workspace path and allowed tools (`Read`, `Write`, `Bash`, `Edit`, `MultiEdit`)
3. Calls the Claude Code SDK's `query()` function with streaming
4. For each response message, serializes the Claude SDK object using Python's `pickle` module, encodes it as base64, and writes it as a JSON line to stdout
5. Captures session IDs from `init` subtype messages for conversation continuity

This architecture enables **conversation continuity** across multiple turns. The `ClaudeCodeManager` maintains a `_chat_sessions` dictionary mapping chat IDs to Claude session IDs. On the first message of a conversation, the subprocess creates a new session; on subsequent messages, it passes `resume=session_id` and `continue_conversation=True` to the SDK, allowing Claude to maintain full context of the prior interaction.

Processing State Machine

Response processing follows a state machine pattern encapsulated in the `ProcessingState` class. As Claude SDK objects stream in from the subprocess, the state machine tracks:

- **Accumulated output** — the running text of Claude's conversational response
- **Latest artifact** — the most recent generated code file, with automatic type detection based on file extension (`.R` → R/Shiny, `.py` → Python/Shiny, `.qmd` → Quarto, `.md` → Markdown)
- **Should refresh artifact** — a flag set when Claude invokes `Write`, `Edit`, or `MultiEdit` tools, triggering a filesystem read on the next processing cycle to capture the updated code
- **Current TODOs** — a task list extracted from Claude's `TodoWrite` tool invocations, providing users with real-time visibility into the agent's plan

The state machine processes Claude SDK message blocks polymorphically:

- `TextBlock` — appended to accumulated output
- `ToolUseBlock` — formatted for display (e.g., "📝 Creating `app.R`", "🔧 Running: `Rscript ...`") and triggers artifact refresh for file-mutating tools
- `ToolResultBlock` — formatted as code block output

Each processing step yields a `PartialResponse` to the frontend, enabling **real-time streaming** of both the conversation and code artifacts. The final `ResultMessage` from the SDK produces a `Response` object that is persisted to the database.

Response Formatting Layer

A dedicated `ClaudeResponseFormatter` class translates raw tool invocations into user-friendly markdown. Each tool type receives a distinct emoji and description:
This layer ensures that non-technical users see meaningful progress updates rather than raw tool names and JSON payloads.

Task Tracking with TodoWrite Integration

TealFlow integrates Claude's native `TodoWrite` tool to provide **real-time task visibility**. When the AI agent plans its approach—for example, deciding to first read the dataset schema, then select appropriate {teal} modules, then generate the application code—it writes these steps as TODO items with statuses (`pending`, `in_progress`, `completed`, `cancelled`).
The frontend renders these tasks using animated `TodoItem` components with distinct visual states: emerald backgrounds for completed tasks, blue for in-progress, red for cancelled, and gray for pending. The component uses framer-motion for smooth entry animations, creating a polished experience where users can watch the AI's plan materialize and progress in real time.
This feature directly addresses a common concern with AI-generated code: the "black box" problem. By exposing the agent's reasoning process as a structured task list, users can verify that the AI understood their requirements before the code is even generated.

AI Agent Pipeline

The AI agent at the heart of TealFlow operates through a structured pipeline that transforms user requests into validated applications. The pipeline begins with intent analysis, where the agent parses the user's natural language request to identify the type of analysis requested, the relevant data domains, and any specific parameters or configurations mentioned.
Next, the agent performs schema analysis on the uploaded dataset. Critically, TealFlow's design ensures that the AI agent inspects only the dataset's schema (column names, data types, and metadata) rather than the actual patient-level data. This approach preserves data privacy while providing the agent with sufficient information to select appropriate modules and configure data mappings.
The agent then enters a module selection phase, where it matches the user's analytical requirements against the catalog of available {teal} modules. For example, a request for "survival analysis" would trigger selection of the `tm_g_km()` module for Kaplan-Meier plots and potentially `tm_t_tte()` for time-to-event summary tables. The agent considers not only the primary analysis but also supporting modules such as data summary tables and patient-level listings that provide context for the main analysis.
Following module selection, the agent generates the complete application code. This includes the `app.R` file with proper {teal} initialization, module configuration with appropriate default parameters mapped to the specific dataset columns, and any custom styling or layout adjustments. The generated code is then validated through automated testing within the container environment before being presented to the user.

Model Context Protocol (MCP) Integration

The Model Context Protocol (MCP) is an open standard developed by Anthropic that enables AI models to interact with external tools and data sources through a structured interface. TealFlow leverages MCP as the bridge between its AI agent and the specialized tools required for clinical application development.

MCP Server Architecture

The TealFlow MCP server is implemented as a modular Python package designed for maintainability and extensibility. The server follows a clean separation of concerns:

A key architectural property of the MCP server is that it operates as a **local data provider**. It reads static JSON metadata files from the `knowledge_base/` directory and does not require any API keys, internet connectivity, or external service dependencies. All {teal} module definitions, parameter schemas, dataset specifications, and code templates are bundled as structured JSON files within the server package. This design ensures that the MCP server can operate in air-gapped environments—a common requirement in pharmaceutical organizations with strict network security policies.
The server implements the MCP stdio transport protocol, making it compatible with any MCP-compliant client. It is LLM-agnostic by design: while TealFlow's hosted platform uses Anthropic's Claude as its AI backbone, the MCP server itself provides tools that any LLM can consume. Confirmed compatible clients include Claude Code, GitHub Copilot, Cursor, and any other tool supporting the MCP stdio protocol.

Available MCP Tools

The TealFlow MCP server exposes eight specialized tools that collectively provide comprehensive support for the {teal} application development lifecycle. Each tool accepts structured parameters and returns results in either Markdown (for human-readable display) or JSON (for programmatic consumption), controlled by an optional `response_format` parameter.

MCP Tool Orchestration Patterns

The tools are designed to be composed in sequences that mirror natural development workflows. A typical application generation session involves the following orchestrated tool chain:

1. **`tealflow_agent_guidance`** — The agent retrieves its operational playbook, establishing the recommended workflow for the current interaction.
2. **`tealflow_get_app_template`** — The agent obtains the application scaffold when starting a new application from scratch.
3. **`tealflow_search_modules_by_analysis`** — Based on the user's natural language request (e.g., "I need survival curves and a demographics table"), the agent identifies candidate modules.
4. **`tealflow_check_dataset_requirements`** — For each candidate module, the agent verifies that the user's available datasets satisfy the module's data dependencies.
5. **`tealflow_get_module_details`** — For confirmed modules, the agent retrieves full parameter specifications to generate correctly configured code.
6. **`tealflow_generate_module_code`** — The agent generates insertion-ready code for each selected module.
7. **`tealflow_check_shiny_startup`** — After assembling the complete application, the agent validates that it starts without errors, iterating on fixes if startup validation fails.

This orchestration pattern ensures that the agent follows a disciplined development process rather than attempting to generate an entire application in a single step. The intermediate validation steps (dataset requirement checks, startup validation) catch errors early and reduce the number of iteration cycles needed to produce a working application.

MCP Client Configuration

The TealFlow MCP server integrates with any MCP-compatible client through a standard JSON configuration block. The server is launched via `uv` (a Python project manager) and communicates over the stdio transport:

This configuration can be added to Claude Code's MCP settings, Cursor's MCP configuration, or any other client that supports the MCP stdio protocol. Each team member maintains their own local instance of the MCP server—no shared server infrastructure is required. The server's dependencies are managed via `uv sync` with a pinned lockfile (`uv.lock`), ensuring reproducible installations across environments.

For development and debugging, the MCP server can be tested interactively using the MCP Inspector:

```
npx @modelcontextprotocol/inspector uv --directory
/absolute/path/to/TealFlowMCP/ run tealflow_mcp.py
```

This launches a browser-based interface where developers can invoke individual tools, inspect their parameters and return values, and verify correct behavior before integrating the server into their LLM workflow.

MCP Runtime Characteristics

Because the MCP server reads static JSON files and performs local computation only, it imposes no API rate limits, usage quotas, or per-invocation costs. The LLM client's own API usage (e.g., Anthropic API tokens consumed by Claude) is the only cost associated with using TealFlow MCP tools. This makes the MCP server suitable for high-frequency iterative development workflows where the agent may invoke dozens of tool calls per session without cost concerns from the tool layer.

The server requires Python 3.10+ and a local R installation with the `shiny`, `teal`, `teal.modules.general`, and `teal.modules.clinical` packages for the `tealflow_check_shiny_startup` tool to function. Dataset files (RDS format) should be placed in a data directory accessible to the workspace; sample CDISC ADaM datasets are provided in the `sample_data/` directory of the repository for immediate use.

Per-Session Workspace Isolation

TealFlow creates a fully **isolated workspace for each chat session**. When a new chat is initiated, the system copies a template workspace directory—containing pre-configured R package environments, sample CDISC ADaM datasets (ADSL, ADTTE, ADRS, ADQS, ADAE as RDS files), an `app.template.R` scaffold, and shell scripts—into a per-chat directory at `working_workspace/<chat_id>`.

This copy-on-create approach, implemented via Python's `shutil.copytree`, provides several guarantees:

1. **Isolation**: Each session operates on its own copy of the workspace. Claude Code SDK's file operations (`Write`, `Edit`, `Bash`) modify only the session-specific directory, preventing cross-session interference.
2. **Reproducibility**: The template workspace contains pinned R package versions managed by `renv`, ensuring that the same code produces the same results regardless of when it is generated.
3. **Recoverability**: If the AI agent produces invalid code, the original template remains untouched. Users can start a new session without side effects.

After workspace creation, path references in shell scripts (e.g., `run_app.sh`) are dynamically rewritten to point to the new session directory, ensuring that application launch commands resolve correctly.

Database-Backed Persistence

All conversations, artifacts, and user data are persisted in a **SQLModel-backed relational database** (SQLite by default, configurable via environment variable). The data model consists of five primary entities:

- **Chat** — stores the agent type, user email, title (AI-guessed from the initial query), creation timestamp, and archive status
- **Message** — linked to a Chat, stores sender role (`user`, `assistant`, `system`), content text, and timestamp
- **Artifact** — linked to a Message, stores the generated code content, artifact type (e.g., `shiny_for_r`, `quarto_report`), and creation timestamp
- **Attachment** — linked to a Chat, stores uploaded files as binary blobs with type detection based on file extension (image, CSV, JSON, PDF)
- **Secret** — stores per-user API keys (GitHub, Posit Connect) for deployment operations

The `DBManager` class provides a clean query interface with methods for retrieving the last correct artifact (filtering out error placeholders), managing chat archives, and ensuring default secrets exist for each user. This persistence layer enables session resumption: users can close the browser, return later, and continue interacting with a previously generated application.

Frontend Architecture: React-in-Shiny

TealFlow's frontend is a **hybrid architecture** combining Python Shiny's server-driven reactivity with a rich React component layer. The UI is built with TypeScript/React components using the shadcn/ui design system, bundled via Bun, and injected into Shiny's HTML output through a custom binding mechanism.

The binding layer works as follows:

1. Python renders a placeholder HTML tag (e.g., `<split-view>`) with Shiny HTML dependencies
2. A JavaScript module import (`index.tsx`) calls `render_id_to_elems()` or `bind_id_to_elem()`, which locates the placeholder element and mounts a React component tree into it using `createRoot`
3. For input components (like the main chat view), `makeReactInput` from `@posit-dev/shiny-bindings-react` creates a bidirectional binding: React state changes propagate to Shiny inputs via `setShinyInputValue()`, and Shiny server messages update React state via `window.Shiny.addCustomMessageHandler()`

The **SplitView** component is the primary workspace interface, implemented as a two-column layout:

Left Column (Chat):

- Renders the conversation as a scrollable message list with distinct avatars for user and AI messages
- Supports markdown rendering via `react-markdown` with syntax-highlighted code blocks (using `react-highlight`)
- Displays a real-time typing indicator (animated bouncing dots with "TealFlow AI is thinking..." text) controlled by `ai_typing_status` custom messages from the server
- Disables the input textarea while the AI is responding, preventing race conditions
- Supports `Ctrl/Cmd+Enter` keyboard shortcut for message submission

Right Column (Code & Preview):

- **Tasks panel** (top 40%) — renders the AI's TODO list using animated `TodoItem` components
- **Files panel** (bottom 60%) — tabbed interface showing `app.R` (editable with `react-simple-code-editor` and Prism.js syntax highlighting) and `data.R` (read-only, showing dataset loading code)
- **Preview tab** — renders the running application via iframe, with support for multiple preview types (raw HTML, Shinylive R/Python URLs, Quarto rendered output, or direct URL)
- **Toolbar** — includes clipboard copy, Deploy to Connect, Deploy to GitHub, fullscreen preview, and a Local/Shinylive run mode toggle

Communication between the React frontend and Python backend uses a well-defined set of custom message channels:

Application Rendering and Preview

Once the AI generates an application, TealFlow must render a live preview for the user. The platform implements a **polymorphic renderer system** using a factory pattern. Each artifact type maps to a dedicated renderer class:

For **local execution**, the `ApplicationsManager` class manages a pool of dynamically allocated ports (range 8501–8600). When a preview is requested:

1. Any existing process for the user session is terminated
2. A free port is selected from the pool
3. The application is launched as a subprocess (e.g., `Rscript -e "shiny::runApp('app.R', port=8523, host='0.0.0.0')"`)
4. Background threads monitor stdout and stderr for the "Listening on" message, confirming successful startup
5. The preview URL is sent to the frontend, which renders it in a sandboxed iframe

For **Shinylive execution**, the renderer uses the `shinylive` Python package to encode the entire application into a self-contained URL. This approach requires no server-side process management but is limited to applications that can run entirely in the browser via WebAssembly.
The run mode is user-selectable via a `RunModeSwitch` toggle in the toolbar, allowing users to switch between local execution (faster, supports all R packages) and Shinylive (portable, shareable, no server dependency).

In-Browser Code Editing

A distinguishing feature of TealFlow's interface is the **editable code panel**. Rather than presenting the generated code as read-only output, the platform uses `react-simple-code-editor` with Prism.js R language grammar to provide a lightweight but functional code editor directly in the browser.
Users can modify the AI-generated `app.R` code and click "Reload Code" to:
1. Write the modified code to the session workspace filesystem
2. Trigger a `reread` operation in the `ChatsManager`, which reads the updated file, creates a new artifact in the database, and re-renders the preview
This workflow supports an iterative development pattern: the AI generates the initial application, the user makes minor adjustments (changing labels, adding parameters, reordering modules), and the system re-renders without requiring a new AI interaction. The modified code is also stored as a new artifact, maintaining a complete audit trail of changes.

Deployment Pipeline

TealFlow provides two production deployment pathways directly from the chat interface:

Posit Connect Deployment:

The platform uses the `rsconnect-python` library to deploy generated applications to Posit Connect instances. For R/Shiny applications, the deployment process:
1. Creates a temporary directory with the `app.R` file
2. Generates an MD5 checksum of the application code
3. Applies a manifest template (`r_manifest.json`) with the checksum
4. Calls `deploy_and_verify_manifest()` to upload the bundle, deploy, and verify

For Quarto reports, the process additionally handles attachments (uploaded CSVs, images), embeds them in the deployment bundle, and generates appropriate YAML headers with `embed-resources: true` for self-contained output.

GitHub Gist Deployment:

For R/Shiny modules, TealFlow generates a specialized gist script using the `generate_gist_script_from_content()` utility. This function parses the generated code to extract all functions ending in `_flow` (the TealFlow module naming convention), wraps them in a `local({...})` block with required library imports, and produces a self-contained R script that can be sourced via `devtools::source_gist()`. This enables a clean integration path: users generate a module in TealFlow, deploy it to GitHub, and import it into their existing R projects with a single function call.

Attachment Processing Pipeline

TealFlow supports file uploads (images, CSVs, PDFs, JSON) that are incorporated into the AI conversation context. The attachment processing pipeline handles each file type distinctly:

- **Images** — base64-encoded and embedded as `image_url` content blocks in the OpenAI-compatible message format, enabling the AI to analyze screenshots or mockups
- **PDFs** — converted to PNG images page-by-page using `pdf2image`, then base64-encoded, allowing the AI to "see" the content of statistical analysis plans or regulatory documents
- **CSVs** — decoded to text and truncated to a configurable threshold (default: 50 lines) to avoid exceeding context windows, with a note indicating how many additional lines exist
- **JSON** — parsed and pretty-printed with `json.dumps(indent=2)` for readability

All attachments are stored as binary blobs in the database and associated with the chat, ensuring they persist across session restarts and are available for deployment bundles.

User Authentication and Secret Management

TealFlow integrates with Posit Connect for user authentication. On session start, the server queries the Connect API to resolve the current user's identity (email, first name, last name), which is used for chat ownership, secret scoping, and display.
Users manage their API keys through a dedicated **Secrets Editor** interface. The editor provides a table of key-value pairs with masked token display (showing only the first and last four characters). Two secrets are

treated as "protected" defaults—`POSIT_CONNECT_API_KEY` and `GITHUB_ACCESS_TOKEN`—and cannot be fully deleted, only cleared. Custom secrets can be added for organization-specific integrations.

Secrets are injected into the AI agent's system prompt as available environment variable names (without values), enabling the agent to generate code that references them. During application rendering, secrets are converted to environment variables and passed to the execution environment, ensuring that deployed applications can access external APIs without hardcoding credentials.

Feedback and Analytics

TealFlow includes a built-in feedback mechanism. Users can submit feedback through a modal dialog with free-text input and sentiment selection (sad, neutral, happy). Submissions are forwarded to a Slack webhook for team review.

The platform also integrates Amplitude analytics with session replay capabilities, providing product telemetry on feature usage patterns. A `request_feature` function tracks user interactions with unimplemented features (such as "Rename" and "Permanent Delete" for chats), providing data-driven input for the product roadmap.

TEALFLOW VISION AND CAPABILITIES

TealFlow embodies a vision of democratized clinical data application development. The platform provides low-code access for non-developers, enabling them to bypass development teams for early validation of analysis concepts. The system outputs reviewable, shareable, production-ready code that can be handed off to development teams for refinement if needed.

The core interaction model uses a chat-based interface that guides users through application creation. Users provide their data and Statistical Analysis Plan (SAP), and the AI suggests appropriate analyses to run based on the input. Everything is conducted via natural language conversation within an AI feedback loop that refines the output iteratively. The platform produces both a working application and the underlying code, which can be validated by experts before deployment. A simple button allows users to publish results when ready.

This approach enables custom tools to be built in minutes rather than the days or weeks traditionally required. The implications for clinical trial workflows are significant: interim analyses that previously required weeks of developer time can now be initiated by the biostatistician directly, safety review dashboards can be generated on demand as new data becomes available, and exploratory analyses that might never have been prioritized in a development backlog can be created by the domain experts who need them.

THE PATH TO OPTIMAL USER EXPERIENCE

Developing an effective solution required iterating through multiple approaches, each building on lessons learned from the previous stage. This evolutionary process reflects the inherent tension between AI capability and user accessibility that characterizes the current state of agentic AI tools.

Step 1: Agent with Prepared Context

The first step involved creating an agent with prepared context, incorporating deep search capabilities on {teal} documentation combined with system prompts optimized for the task. This approach used retrieval-augmented generation (RAG) to ground the AI's responses in authoritative {teal} documentation, ensuring that generated code followed framework conventions. While this established a solid foundation of domain knowledge, it lacked an accessible interface—users needed to interact through API calls or command-line tools, limiting adoption to technically proficient users.

Step 2: Frontend Application

The second step introduced a dedicated frontend application. This approach offered ease of use through browser access and standard features like chat history, with additional capabilities such as sharing generated applications to GitHub or Posit Connect. The web interface significantly lowered the barrier to entry, allowing non-developers to engage with the system through a familiar chat paradigm. However, this approach was limited to simple conversational interactions without true agentic capabilities—the system could generate code based on a single prompt but could not iteratively refine, test, or deploy applications.

Step 3: Developer Tools Integration

The third step explored integrating directly with developer tools. By leveraging the Model Context Protocol (MCP), this approach provided the full capability of state-of-the-art AI agents with tools for auto-testing generated code, accessing live {teal} documentation, and managing the development workflow. Developers could use their preferred IDE (such as VS Code or RStudio) with the TealFlow MCP server providing specialized {teal} capabilities alongside general AI coding assistance. While technically powerful, this approach presented a significant barrier for non-coders who found the development environment intimidating and required considerable setup effort.

Step 4: TealFlow — Unified Platform

The fourth step combined the best elements of all previous approaches into the TealFlow platform. This solution merges the agentic approach with a safe container environment, providing a single platform for multiple agents with easy browser access, user history, and collaborative features. The platform integrates Claude Code as its AI

backbone, providing sophisticated code generation and refinement capabilities within a user-friendly web interface. While it offers a somewhat limited development experience compared to full IDEs, this trade-off enables accessibility for a much broader user base—which is precisely the target audience for democratized clinical data application development.

The technical realization of Step 4 is the system described in the preceding architecture sections: a Python Shiny backend orchestrating Claude Code SDK via subprocess isolation, with a React-based frontend providing real-time streaming of conversations, code artifacts, task lists, and live previews—all accessible through a standard web browser with no development environment setup required.

PRACTICAL WORKFLOW: FROM SAP TO APPLICATION

To illustrate TealFlow's capabilities in a realistic scenario, consider the following workflow for generating an efficacy analysis application from a Statistical Analysis Plan (SAP).

A biostatistician working on a Phase III oncology trial needs to create an interactive application for exploring overall survival and progression-free survival endpoints. Traditionally, this would require submitting a request to the development team, providing detailed specifications, and waiting for the application to be built and tested—a process that typically takes one to three weeks.

With TealFlow, the biostatistician opens the platform in their browser, uploads the ADaM-format analysis datasets (ADSL and ADTTE), and describes their requirements: "I need a survival analysis application with Kaplan-Meier plots for overall survival and progression-free survival, stratified by treatment arm, with hazard ratio estimates and a demographics summary table."

The TealFlow agent analyzes the uploaded datasets' schemas, identifies the relevant variables (PARAMCD for endpoint identification, AVAL and CNSR for time-to-event data, ARM or ACTARM for treatment stratification), and proposes a module configuration. The agent selects `tm_g_km()` for Kaplan-Meier visualization, `tm_t_tte()` for time-to-event summary statistics including hazard ratios, and `tm_t_summary()` for the demographics table. The user reviews the proposed configuration, requests a minor adjustment (adding a subgroup analysis by region), and the agent regenerates the application with the additional module.

Behind the scenes, TealFlow's task tracking panel shows the agent's plan progressing in real time: "Read dataset schema" → completed, "Select teal modules" → completed, "Generate app.R" → in progress, "Validate application" → pending. The code editor updates live as the agent writes the `app.R` file, and once complete, the preview panel automatically launches the application for interactive exploration.

Within minutes, the biostatistician has a fully functional interactive application that they can use for data exploration, share with colleagues for review, or publish to the organization's Posit Connect server for broader access. The generated R code is available for download and can be handed to the development team for further refinement, validation, and eventual inclusion in the regulatory submission package.

TEALFLOW MCP: OPEN SOURCE CONTRIBUTION

To support the broader community and address the trust gap that non-coders often experience with AI-generated code, we have open-sourced the TealFlow MCP component. This tool integrates with developers' environments and is publicly available for anyone to use in {teal} projects.

The TealFlow MCP server implements the Model Context Protocol specification, providing a standardized interface through which any MCP-compatible AI client can access {teal}-specific tools. The server exposes the full suite of nine tools described in the MCP Integration section—from agent guidance and module discovery through code generation and runtime validation—enabling any MCP-compatible LLM client to build {teal} applications with domain-specific knowledge that would otherwise be unavailable to general-purpose AI models.

Adoption Without Platform Lock-In

A critical design goal of the open-source MCP component is to provide value independently of the full TealFlow platform. Developers who prefer working in their existing IDEs can configure the TealFlow MCP server alongside their preferred AI coding assistant—whether Claude Code, GitHub Copilot, Cursor, or another MCP-compatible tool—and immediately gain access to {teal}-specific module catalogs, code generation templates, and startup validation. This provides an incremental adoption path: teams can evaluate the MCP tools in their familiar development environment before deciding whether to adopt the full TealFlow web platform.

The server's local-only architecture (no API keys, no internet dependency, no usage quotas) makes evaluation frictionless. A developer can install the MCP server, configure their client with a single JSON block, and begin generating {teal} applications with enhanced AI assistance within minutes. The complete test suite (`pytest`-based with coverage reporting) provides confidence in the server's correctness, and the MCP Inspector integration enables interactive exploration of each tool's behavior.

Community Goals

By publishing this component as open source, we aim to achieve several goals. First, we seek to build confidence in AI-assisted development within the clinical data community. By making the tool integration layer transparent and auditable, organizations can evaluate exactly how the AI agent interacts with {teal} and what capabilities it has access to. This transparency is essential in a regulated environment where trust and verifiability are paramount.

Second, we aim to enable organizations to evaluate and adopt the technology at their own pace. The MCP server can be integrated into existing development workflows without requiring adoption of the full TealFlow platform, providing the incremental adoption path described above.

Third, we contribute to the growing ecosystem of MCP-compatible tools that are making AI development assistants more capable and more specialized. As the Model Context Protocol gains broader adoption across the industry, having domain-specific MCP servers for clinical data workflows will become increasingly valuable.

The TealFlow MCP is available at https://github.com/Appsilon/TealFlowMCP.

SECURITY AND COMPLIANCE CONSIDERATIONS

Deploying AI-driven tools in the pharmaceutical industry requires careful attention to security and regulatory compliance. TealFlow's architecture addresses these concerns through several design decisions.

Data privacy is preserved through TealFlow's schema-only analysis approach. The AI agent never inspects patient-level data; it works exclusively with dataset metadata (column names, data types, and value labels) to select and configure modules. Actual patient data remains within the containerized execution environment and is processed only by the validated {teal} framework code, not by the AI model itself. This architectural decision ensures that sensitive clinical data is never transmitted to external AI services.

Reproducibility and auditability are supported through complete code generation. Every application produced by TealFlow comes with the full source code, enabling organizations to review, validate, and version-control the generated output. The code serves as a complete audit trail of the analytical configuration, meeting the traceability requirements of GxP environments. The database-backed persistence of all messages, artifacts, and their timestamps provides an additional layer of auditability.

Access control and session isolation are enforced through the workspace architecture. Each user session operates within an isolated directory with its own file system. Workspace lifecycle management ensures that temporary data is contained within session boundaries, preventing data leakage between users or across sessions. User authentication is delegated to Posit Connect, and per-user secret management ensures that deployment credentials are scoped appropriately.

FUTURE DIRECTIONS

The TealFlow platform continues to evolve along several development axes. Near-term enhancements include expanding the catalog of supported {teal} modules to cover additional therapeutic areas and analysis types, improving the agent's ability to handle complex multi-module applications with inter-module data dependencies, and adding support for custom organization-specific modules through a plugin architecture.

Longer-term research directions include exploring multimodal interactions where users can provide visual references (mockups, screenshots of existing reports) as input to the application generation process—a capability already partially supported through the PDF and image attachment pipeline. We are also investigating collaborative workflows where multiple users can co-create applications through shared chat sessions, and developing automated validation pipelines that can generate the documentation required for regulatory submissions directly from the generated application code.

We are also investigating the integration of TealFlow with emerging standards and platforms in the clinical data ecosystem, including the CDISC Analysis Results Standard and the pharmaverse suite of validated R packages. These integrations would further strengthen TealFlow's position as a bridge between AI-driven productivity and regulatory-grade analytical tooling.

CONCLUSION

TealFlow represents a significant step forward in making clinical trial application development accessible to a broader audience while maintaining the technical rigor required in regulated environments. By combining specialized AI capabilities with an intuitive chat-based interface, the platform addresses the key challenges that have limited the adoption of AI tools in the clinical data domain: lack of framework-specific knowledge, inability to accommodate organizational constraints, and usability barriers for non-technical users.

The platform's technical architecture—built on a multi-agent system with Claude Code SDK subprocess isolation, the {teal} framework, the Model Context Protocol, per-session workspace isolation, a React-in-Shiny hybrid frontend, and a polymorphic rendering pipeline—provides a robust foundation for AI-assisted clinical application development. The multi-layered validation pipeline ensures that generated applications meet the quality standards expected in the pharmaceutical industry, while the schema-only data analysis approach preserves patient privacy.

The approach demonstrates that with careful attention to user experience, domain-specific tooling, and security architecture, generative AI can effectively support specialized workflows while remaining accessible to non-technical users. As the technology continues to evolve, we anticipate further opportunities to accelerate clinical data analysis and empower subject matter experts throughout the pharmaceutical industry, ultimately contributing to faster delivery of life-saving medicines to patients.

We invite feedback from the community on how this technology might integrate into existing workflows, what challenges it could address for development teams, and which features would create the most value for organizations working with clinical trial data.

ACKNOWLEDGMENTS

The author thanks the Appsilon team for their contributions to the development of TealFlow and the broader {teal} community for their continued advancement of open-source tools for clinical data analysis. Special recognition goes to the insightsengineering community and Roche/Genentech for creating and maintaining the {teal} framework that forms the foundation of this work. The author also acknowledges the Anthropic team for the development of the Model Context Protocol standard and the Claude Code SDK that enables TealFlow's agentic capabilities.

RECOMMENDED READING

- https://insightsengineering.github.io/teal/latest-tag/
- https://modelcontextprotocol.io/
- https://github.com/Appsilon/TealFlowMCP
- https://pharmaverse.org/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Marcin DubelCompany: AppsilonAddress: Chmielna 2/31, 00-020 Warszawa, PolandWork Phone: +48 694 293 142Email: marcin@appsilon.comWebsite: appsilon.com/tealflow