Paper ML22

Is the AI Coding Assistant Catching Your Vibe?

A Practical Assessment of Generative AI Tools in Clinical Data Analytics

Chendi Liao, Recursion, Salt Lake City, USA

ABSTRACT

AI-powered coding assistants and vibe coding tools are no longer a novelty nowadays, their potential to revolutionize data analytics is a topic of great interest and intense discussion. But beyond the hype, what is the reality of these GenAI solutions in the setting of clinical data science? This presentation details a hands-on evaluation of several leading commercially available AI code editors and LLMs, such as Claude Code, Cursor AI, applied onto a real-world task of developing a clinical data review dashboard. We'll share our practical assessments of the extent and the boundaries of their capabilities - where they truly accelerate development, and conversely, where they may introduce unforeseen risks or require intensive human-in-the-loop oversight and more rigorous validation. Our experience offers key insights for how to leverage these powerful tools effectively and responsibly, helping clinical data science teams to navigate the promise and pitfalls of AI-assisted coding in a regulated environment.

INTRODUCTION

Generative AI (GenAI) has rapidly moved from novelty to daily companion in many code development workflows. Developers can now utilize AI coding assistants to generate code at unprecedented speed, chat with their code repository or documentations, or even delegate multi-file refactors to agentic AI tools that can edit files and run commands. In parallel, a new culture of "vibe coding" has emerged—an approach that prioritizes momentum of ideas and speed of iteration by describing intent in natural language and letting the AI tool interpret then complete the implementation details.

In clinical data analytics, however, "moving fast" is not the only objective. Clinical data and its analytical results are interpreted and used for high-stakes decision making processes, thus are produced with the non-negotiable expectations of accuracy, traceability, as well as reproducibility. The inherent generative AI behaviors, such as hallucinated syntax, silent logic changes, or overconfident explanation, that may be tolerable for quick software prototyping, can become unacceptable risks when applied to regulated clinical workflows.

The appeal of GenAI is undeniable in clinical development, a field characterized by demanding timelines and high-stake outcomes. Pharmaceutical companies, particularly those sponsoring first-in-human and early-phase trials, require close data monitoring and rapid turnaround of insights for frequent safety reviews and dose-escalation decisions. As clinical data scientists or statistical programmers face pressure to deliver insights faster, can these trending AI tools truly expedite high-quality work without jeopardizing regulatory compliance? This paper introduces several leading commercially available AI coding assistant tools and details a hands-on evaluation of applying these tools on a real-world task of developing a clinical data review dashboard. We aim to separate the hype from the reality, identifying where GenAI and the concept of "vibe coding" may truly accelerate development and where it introduces unforeseen risks requiring intensive human-in-the-loop oversight in the highly regulated setting of clinical data analytics.

A LANDSCAPE OVERVIEW OF AI-POWERED CODING ASSISTANTS

The landscape of AI coding tools has evolved rapidly from simple autocomplete features to fully autonomous agents. To help navigate this crowded space in a way that's most relevant for clinical data analytics, we categorized the tools that are commercially available and enterprise-grade into four major types, each offering different levels of integration and capability (Table 1).

AI CHATBOTS

Conversational chatbots, such as OpenAI's ChatGPT, Google DeepMind's Gemini, and Perplexity AI's Perplexity, are the AI tools most widely recognized and used by the general public. These tools run on general-purpose LLMs are easily interacted with via a web browser or smartphone apps. They are flexible for brainstorming, explaining unfamiliar concepts, drafting documentation, or producing small code snippets, which make them excellent for ideation or architectural advice. However, their major limitation is the lack of contextual awareness - unless the user pastes or upload files, the model cannot see the project structure or existing codebase, leading to higher chance of hallucination or generic suggestions that may not fit the specific implementation details. They also cannot directly execute code in a statistical computing environment (SCE), shifting the burden of validation and traceability to the user, making them less efficient for complex, multi-file code development.

Type	Conversational Chatbot	Coding Partner (IDE extension)	Standalone IDE	CLI
AI Tool Tested	**Enterprise Google Gemini/ChatGPT**	**Github Copilot (via Rstudio)**	**Cursor AI**	**Claude Code**
Embedded IDE	No	Yes	Yes	No
Multi-LLM support	No	No (possible in other IDEs)	Yes	No
Context Scope	None	Single file (can enable project-wide)	Full project / repo	Full local filesystem via Shell access
Multi-file edits	No	No	Yes	Yes
Code execution	No	Yes	Yes (automated with permission)	Yes (automated with permission)
Multi-application execution	No	No	Yes (require system permission)	Yes (require system permission)

Table 1. Feature comparison between major types of AI coding assistants and the commercial products tested in the respective category.

CODING PARTNER INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) PLUGIN OR EXTENSION

AI coding partners for IDEs seamlessly integrate GenAI into popular code editors, such as RStudio or VS Code, often presenting as inline autocompletions. By analyzing the code or comments in the active window, these assistants proactively predict syntax, suggest the next few lines of code, or in some cases, even provide a complete function with arguments, as the user types. Their key advantage is proximity to active development, as the micro-acceleration of code completion can accumulate significant time savings on tasks like localized refactors, unit test skeletons, or repetitive text modification. On the other hand, they are less effective for tasks that require multi-file editing, as their context window is often limited to the active file only. For example, they may struggle to understand how a change in an ADaM data processing script may affect a visualization script in a separate folder, even if they are under the same project. Currently, the most widely adopted AI coding partner for IDEs is Github Copilot, other noteworthy competitors include Gemini Code Assist and Tabnine.

STANDALONE AI-NATIVE IDE

AI-native IDEs are full code editors purposely built around an "AI agent" that is capable of reasoning across the entire project codebase, concurrently editing multiple files, and iteratively correcting errors. Examples of most well-known AI-native IDEs include Cursor AI, Windsurf and Aide. These IDEs often feature different task modes, such as read-only exploration, plan-first workflows, or autonomous coding. They are designed to maximize context scope by indexing the full repository to maintain a persistent, project-level understanding without losing implementation details, making them the preferred tool for “vibe coding”. These AI-native IDEs can outperform traditional IDE plugins for complex refactors, by coordinating changes across multiple files—for instance, suggesting updates to a downstream visualization script when a variable name changes in a data processing script.

Nevertheless, switching to a new IDE can present barriers regarding user adoption or IT approvals. Furthermore, there are also risky tradeoffs, particularly in regulated environments. Automated multi-file edits produce wide-reaching codebase changes that are more difficult to audit, review and validate. Debugging large volumes of AI-generated code will also be challenging if silent logic errors are present.

COMMAND LINE INTERFACE (CLI) CODING AGENTS

CLI-based coding agents, such as Claude Code and Gemini CLI, offer powerful automation capabilities that are similar to the AI-native IDEs, but operate directly within the terminal (or Windows Command Prompt). They are designed for typical software developer routines, allowing GenAI to access and manage local file systems, execute shell commands, and interact with version controls. One big drawback for seasoned clinical data scientists is the lack of visual interface, making them less intuitive for step-by-step data processing, or visual outputs like plots and dashboards. Furthermore, they pose the highest operational risk due to their system-level command execution capability. In regulated settings, this functionality would require stringent constraints, exhaustive assessment, and rigorous monitoring to effectively manage compliance risk.

CLINICAL DATA DASHBOARD USE CASE

As a small clinical-stage techbio with clinical pipeline consists of mostly early phase and first-in-human trials, we need to perform frequent safety reviews and dose-escalation decision meetings requiring close data monitoring and rapid turnaround of insights. To support these routine needs, we have built open-source-based analytics solutions that included a static HTML patient profile report using `RMarkdown` framework (Liao, 2025).

These consolidated reports are vital for clinical scientists and medical monitors to review subject disease characteristics, safety and efficacy signals across listing, figures and summary tables. However, over the course of

the trial, as data volume and analysis contents increased, this type of reporting tool hit performance bottlenecks resulting in excessive slow loading time and significant delayed responses. Stakeholders struggled to navigate the report and find the desired data efficiently, leading to a poor user experience. Therefore, we aimed to transform this legacy report into a more modernized clinical data review dashboard, that is easy to maintain and uses more contemporary languages.

The requirements for the newly revamped dashboard are straightforward but substantial:

- **Improve performance**: Fast loading time regardless of trial size
- **Code reuse**: Reuse of the existing `RMarkdown` codebase as much as possible, preserving legacy knowledge and reducing re-validation scope
- **Add flexibility**: Enable multiple output formats rendering, for example HTML for interactivity during review and PDF for archival needs
- **Code maintenance**: Avoid the use of R-shiny to reduce code complexity and maintain code readability

This dashboard modernization project provides an excellent, realistic opportunity to apply and assess AI coding assistants within clinical analytics. The evaluation would encompass various task types, ranging from high-level conceptualization to specific code implementation, and includes aspects of user interface, formatting, as well as core analytical content. Furthermore, as we already have a clear strategy for the migration, the associated risks of using GenAI are minimal and easily managed.

Due to the diverse range of tasks involved, we selected different categories of GenAI tools based on their strengths. For isolated non-coding tasks requiring a user interface, such as ideation and image processing, we employed conversational chatbots Google Gemini using Gemini 2.5 Flash/Pro (Google DeepMind, 2025) and ChatGPT with GPT-5.2 (OpenAI, 2025). The core coding and implementation demonstrated in this paper were primarily conducted using the AI-native IDE Cursor (Anysphere, 2026), specifically utilizing its Auto Mode, which consisted of a blend of LLMs including Composer 1 (Anysphere, 2025), Claude 4.5 Opus or 4 Sonnet (Anthropic, 2025), and GPT-5.2 (OpenAI, 2025). This mode features an AI agent that dynamically selects the most appropriate and reliable LLM for each immediate task. A key capability of Auto Mode is its ability to detect when output performance degrades and automatically switch models to maintain high reliability.

TASK 1: IDEATION AND ARCHITECTURE APPROACH

We prompted AI chatbots with the problem and the set of requirements, and asked it to propose some potential solutions and enhancements for transforming the legacy `Rmarkdown` patient profile reporting pipeline into a modernized data review dashboard.

AI tools correctly pinpointed the bottleneck of our legacy pipeline: the embedding of large data within HTML. Specifically, Gemini (Google DeepMind, 2025) suggested moving to a “Hybrid Quarto” approach. This solution, which perfectly matched our independently human-brainstormed strategy, involved using `Quarto` (Allaire, 2025) software to create a website by pre-rendering individual patient profile pages through a parameterized loop and implementing a lazy-loading technique to optimize the loading speed of each webpage.

In contrast, while ChatGPT (OpenAI, 2025) also identified the underlying cause, it suggested alternative programming languages (e.g. JavaScript) and more advanced HTML formatting (e.g. JSON), which does not align with our own strategy or institutional knowledge. Nonetheless, both chatbots were eager to offer code snippets and start generating code for the new workflow, which is beyond the scope of the original prompt.

TASK 2: VIBE CODING TO IMPLEMENT THE MIGRATION TO QUARTO WEBSITE

Once we determined the approach for the new dashboard - creating a Quarto website - we utilized Cursor IDE (Anysphere, 2026) to start implementing the code migration accordingly. Within merely 10 minutes, after an internal “analyzing” and “thinking” process, the AI IDE generated a complete suite of files: several Quarto scripts, a custom CSS styling script, an R script for creating individual patient webpages, as well as a README file detailing the execution pipeline and dashboard rendering instructions.

After manually correcting a minor bug involving a missing library call, the Quarto website rendered successfully, the resulting dashboard’s core functionalities and the smooth loading speed met the original requirements. Nevertheless, the result was not flawless. The website’s layout and formatting presented several suboptimal elements, such as a truncated table of contents, overlapping content, and awkward placement of the patient list, etc. Furthermore, despite the prompt’s instruction to convert the code to Quarto, the code largely retained legacy `Rmarkdown` syntax. This, however, did not cause a failure because of Quarto’s extensive syntax and language compatibility.

Overall, the AI coding assistant created a solid foundational structure for the new Quarto dashboard, although further refinement is warranted, it is able to serve as a good starting point while significantly reducing development time from

hours to mere minutes. This strong performance is partly due to the fact that the task focused solely on structural coding and syntax shift, and did not require the creation or modification of any analytical content.

TASK 3: ADDING AND MODIFYING ANALYTICAL CONTENTS

Following the constructive migration to Quarto, which resolved loading performance issues, we shifted our evaluation to the AI coding capabilities in clinical data analytics programming.

Our first test involved asking Cursor to program for a standard summary table for treatment-emergent adverse events (TEAE) by system organ class (SOC) and preferred term (PT), utilizing the widely recognized R package for clinical reporting, `tern` (Zhu, 2025). The AI promptly generated a substantial block of R code after a brief reasoning phase. This code included the proper table header and employed appropriate `tern` functions and structure, appearing correct on syntax from a quick glimpse. However, the code execution resulted in an error. A more thorough review revealed that the AI code used incorrect options for several function arguments, leading to logical errors during computation that resulted in failure. Additionally, it omitted a necessary formatting argument leading to an undesirably formatted table that might lead to interpretation errors. These types of errors may easily slip through, without a rigorous code review or reviewer without deep domain knowledge about the correct algorithm or the R package itself.

Next, we increased the task complexity by prompting the AI assistant to refactor sections of the analysis code into robust reusable functions to enhance maintainability, and then apply the function to generate a similar TEAE summary table but subsetted to events related to the study drug (Figure 1A). Cursor handled the initial code refactoring task effectively, encapsulating the lengthy analysis code pipeline into well-structured, modular functions. However, for the subsequent application to the new analysis, it failed to reuse the existing function. Instead, it created a largely duplicated version (with minor data processing differences) under a similar name. This approach contravenes the principles of good functional programming and defeats the objective of maintaining clean, modular code.

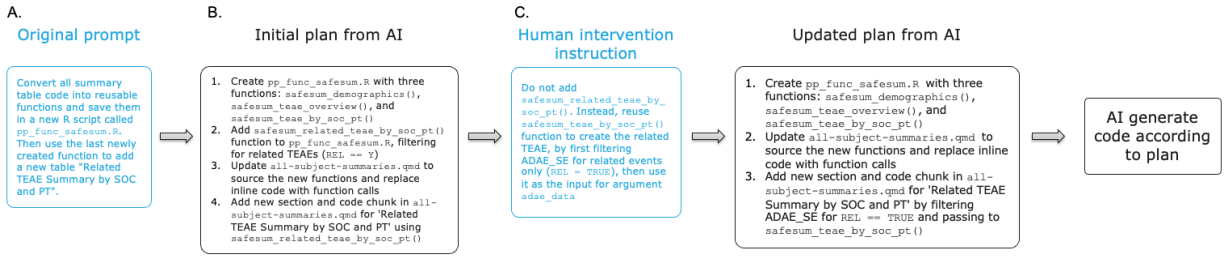

Figure 1. An example of specification-driven development with an AI coding assistant.

To improve AI performance, we reused the same prompt but activated the “Plan Mode” setting on the Cursor IDE, instructing it to first generate an implementation plan for review and approval instead of writing code immediately. This resulted in the AI producing a very thorough, step-by-step execution plan (Figure 1B). The plan detailed the design and parameterization of each function, listed the sequence of actions, and concluded with a task summary.

Notably, this to-do list accurately reflected the inherent limitations and suboptimal "train-of-thought" observed in the AI's prior coding attempt. However, the critical benefit of this “spec-first” approach is that it provides human users with the opportunity to intervene early. As shown in Figure 1C, users can course-correct the AI’s strategy before any code is written. The subsequent, updated action plan, and the AI-generated code that followed, perfectly aligned with the desired refactor strategy, resulting in a substantial reduction of redundant code. This specification-driven code development approach is effective because it enforces human oversight, preserves context, significantly improves code quality and reliability, and mitigates issues related to AI hallucination and misalignment.

TASK 4: UI CUSTOMIZATION

To enhance the dashboard’s visual appeal and offer more information content about the presented data, we tasked Cursor with creating a floating sidebar to the landing webpage that can dynamically match any developer-defined bootstrap or branding theme. The AI tool performed exceptionally well, quickly generating clean, fully functional CSS and automatically applied it to the Quarto document within minutes. This type of low-risk aesthetic work, such as adjusting spacing and padding, creating matching color palettes, and customizing layout, though often tedious for data scientists, represent some of the highest-return uses of AI.

TASK 5: ADDING CONDITIONAL QUARTO BLOCKS FOR PDF RENDERING

A key requirement for the new dashboard was to support multiple output formats, leveraging Quarto’s native multi-format capability. We asked Cursor to follow an example code snippet and update the Quarto script to systematically

add a conditional PDF rendering code chunk for all summary tables. Crucially, the underlying statistical computation was to remain untouched. Since this was a rule-based pattern implementation, the AI coding assistant excelled at this repetitive refactoring task. Once the change pattern was successfully identified, it was applied consistently across the entire codebase, highlighting the value of AI in pattern recognition and automating “grunt work”.

GUIDELINES FOR OPTIMAL HUMAN-AI COLLABORATION

Our above experience has demonstrated that AI coding assistants can meaningfully boost productivity in certain scenarios, but they require careful human oversights in others. Its performance degraded significantly in terms of accuracy and precision when we moved from coding structure to analytical substance. We have synthesized these practical lessons into the following general guidance for integrating AI coding assistants into clinical data science workflows. The aim is to harness the acceleration power of AI while strictly maintaining validation discipline.

BETTER USE CASES

- **Ideation**: Rather than just asking it to write code, leverage AI coding assistants as a senior architect for brainstorming. Asking for options on how to solve a problem can often yield educational alternatives that hadn’t been considered before.
- **Automation of "grunt work"**: Lean on AI for tedious but non-analytical tasks such as writing boilerplate code, setting up unit tests, generating documentation, automating repetitive code modifications. Treat AI like junior coding assistants who can follow instructions to work independently on small low-risk tasks, but assume they will be fallible and make mistakes that will require rigorous quality control.
- **Refine or customizing UI**: AI excels at web-interface technologies (e.g. CSS, HTML, JavaScript). As long as the desired visual outcome can be described, AI can usually generate the code to achieve it. These are typically high-effort, low-risk areas where productivity gains are significant and the validation is easy.
- **Leverage for Regex**: AI is surprisingly good at parsing string patterns and generating straightforward regular expressions, a task that is notoriously confusing and error-prone for humans.

AREAS FOR CAUTION AND RIGOROUS OVERSIGHT

- **Domain-heavy logic**: Clinical data analytics require knowledge of complex industry standards like CDISC, medical or disease context, as well as study-specific knowledge like study protocol or statistical analysis plan. But currently available commercial AI coding assistants are primarily designed for mainstream software development workflows. Without the defined and correct clinical context, AI has a high risk of hallucinating, misunderstanding scope or making silent logic mistakes.
- **Clinical data derivations or analysis**: In the high-stake, regulated environment of clinical analytics, the absence of errors does not imply correctness; in fact, code that is “almost correct” can pose a greater risk than “not done”. Therefore, mistakes or logical errors from AI-generated code must be considered costly, necessitating exhaustive validation and expert review for all tasks involving clinical data analysis.
- **Avoid "generate insights" directly**: AI coding tools are not clinical data analysts or biostatisticians, thus should not be used to interpret clinical data results. It may create flawed explanations and hallucinations that look plausible but are statistically and clinically unfounded.

TIPS FOR FUTURE SUCCESS

- **Use Spec-Driven Development**: The quality of the output is directly proportional to the quality of the instructions and specification. Move away from vague prompts or “vibe”; provide clear objectives, non-negotiable constraints, acceptance criteria, scope boundaries, and desired logic if any. Instead of immediate code generation, leverage AI to first create an implementation or execution plan that is easy to understand and review.
- **Domain-adapted approaches**: For a specialized field of clinical analytics, to optimally harness deeper AI capabilities, domain-tuned systems are a more promising direction
    - **RAG (Retrieval-Augmented Generation)** enhances LLM responses by retrieving and indexing information from specific sources, such as CDISC standards, SOPs, or internal codebases. This additional relevant data is then injected into the LLM prompt, which helps to ground the responses within a domain-specific context, further reducing the likelihood of hallucination.
    - **MCP (Model Context Protocol)** provides more structured integration layers, allowing AI tools to connect securely via APIs to external databases, tools or data in real-time. This process ensures the LLM utilizes standardized and auditable tools to interact with live systems or latest information.
    - **PEFT (Parameter-Efficient Fine-Tuning)** adjusts and customizes the model weights or parameters via model training to handle specialized tasks. This technique can improve AI model performance for domain-heavy fields such as clinical data analytics.
- **Reserve AI-assisted coding for experienced analysts** who possess the domain expertise to distinguish helpful suggestions from flawed hallucination. They can leverage AI tools as a high-speed typist and productivity accelerator, while maintaining control over the scope and quality of the code and results.

CONCLUSION

While commercial AI coding assistants can accelerate clinical analytics, their value is realized only when applied responsibly and within appropriate scopes, as demonstrated in our clinical data review dashboard modernization project. The most reliable benefits came from low-risk tasks such as UI customization (e.g., CSS adjustments) and repetitive non-analytical tasks (e.g., modifying Quarto block syntax). GenAI also performed well for isolated small tasks like ideation or image processing, provided the correct agent was used. Conversely, applying AI coding assistants to clinical data derivation or analysis that required specialized context and heavy domain knowledge introduced unacceptable risks without meaningful efficiency gains. In these high-risk scenarios, the time saved in writing code was merely shifted to the necessary process of reviewing and validating the AI-generated code.

It is important to recognize that the current commercial AI coding tools are optimized for general software engineering, thus may not consistently align with the working principles and risk profile of clinical data science and statistical programming. However, even for a highly regulated field where accuracy, reproducibility and traceability are non-negotiable constraints, the overarching goal should not be to discourage the adoption of AI coding assistants, but to foster safe productivity and responsible-use. As GenAI tools continue to evolve rapidly, meaningful improvements will come from models and systems that are fine-tuned to clinical analytics languages, integrated with enterprise governance and grounded in validated standards and knowledge. Until such systems are realized, the most effective resource for clinical data scientists remains their own domain expertise, amplified but not replaced by AI. Ultimately, human-in-the-loop is the best validation tool, ensuring that AI-assisted coding is not merely a vibe-driven generation but a reliable and auditable acceleration.

REFERENCES

Allaire, J., Teague, C., Scheidegger, C., Xie, Y., Dervieux, C., & Woodhull, G. (2025). Quarto (Version 1.8) [Computer software]. https://doi.org/10.5281/zenodo.5960048

Anthropic. (2025). Claude 4 Sonnet (22 May 2025 version) & Claude 4.5 Opus (24 November 2025) [Large language model]. https://claude.ai

Anysphere. (2025). Composer 1 (29 October 2025 version) [Large language model]. https://cursor.com/blog/composer

Anysphere. (2026). Cursor (10 January 2026 version 2.3.34) [Integrated development environment]. https://cursor.com/home

Google DeepMind. (2025). Gemini 2.5 Pro (25 March 2025 version) [Large language model]. https://gemini.google.com

Liao, C., & Yousefi, K. (2025). OS07: Building Rome in a Few Months: How Open-Source Technologies Empower Small Pharma. Proceedings of PHUSE US Connect 2025, Orlando, FL. https://www.lexjansen.com/phuse-us/2025/os/PAP_OS07.pdf

OpenAI. (2025). GPT-4.1 (14 April 2025 version) & GPT-5.2 (11 December 2025 version) [Large language model]. https://platform.openai.com

Zhu J, Sabanés Bové D, Stoilova J, Garolini D, de la Rua E, Yogasekaram A, Wang H, Collin F, Waddell A, Rucki P, Liao C, Li J (2025). tern: Create Common TLGs Used in Clinical Trials (Version 0.9.10) [R package]. https://insightsengineering.github.io/tern/

CONTACT INFORMATION

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

Author Name: Chendi Liao
Company: Recursion
Email: chendi.liao@recursion.com

Brand and product names are trademarks of their respective companies.