Journal of proteome research

pubs.acs.org/jpr

Technical Note

Vibe Coding Omics Data Analysis Applications

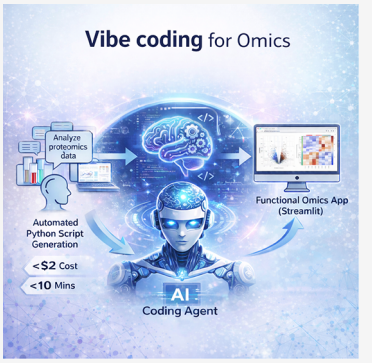
Jesse G. Meyer*

Cite This: J. Proteome Res. 2026, 25, 1191−1197

Read Online

ACCESS | Metrics & More | Article Recommendations | Supporting Information

ABSTRACT: Building custom data analysis platforms has traditionally required extensive software engineering expertise, limiting access for many researchers. Here, I demonstrate that modern large language models (LLMs) and autonomous coding agents can dramatically lower this barrier through a process called "vibe coding", an iterative, conversational style of software creation where users describe goals in natural language and AI agents generate, test, and refine executable code in real time. Importantly, the goal here is not to introduce a new analysis platform. Instead, the example application illustrates that, in minutes, LLMs can now perform work that would normally require at least days of manual programming effort, lowering the cost and time investment by orders of magnitude. As a proof of concept, I used vibe coding to create a fully functional proteomics data analysis platform capable of performing standard tasks, including data normalization, differential expression testing, and volcano plot visualization. The entire application, including user interface, backend logic, and data upload pipeline, was developed in less than 10 min using only four natural language prompts, without writing any additional code by hand, at a model usage cost of under $2, not including hosting or personnel time. Previous works in this area have typically required substantial investment of personnel time from highly trained programmers, often amounting to tens of thousands of dollars in total research effort. I detail the step-by-step generation process and evaluate the resulting code's functionality. This demonstration highlights how vibe coding enables domain experts to rapidly prototype sophisticated analytical tools, transforming the pace and accessibility of computational biology software development.



KEYWORDS: *vibe coding, AI agents, proteomics, bioinformatics, large language models, Streamlit*

INTRODUCTION

Mass spectrometry−based proteomics depends on a multistage computational pipeline, beginning with raw data processing and peptide identification, followed by statistical modeling and interpretive analysis of protein abundance data. Over the past decade, a diverse ecosystem of tools has emerged to support these stages. At the raw data level, MSFragger[1] and DIA-NN[2] have become among the most widely used frameworks for converting instrument files into quantitative protein matrices. These programs produce peptide- and protein-level quantification tables that serve as the input for the next stage of analysis: statistical modeling, visualization, and biological interpretation.

At this interpretive layer, Perseus remains one of the most influential tools.[3] Designed for users without programming experience, Perseus provides a graphical environment for normalization, imputation, clustering, enrichment analysis, and visualization, enabling biologists to perform complex analyses interactively. A new generation of web-based platforms has expanded on this accessibility by moving similar functionality into the browser. ProteoArk is a recent example, offering automated normalization, differential expression, and visualization pipelines that accept standard output formats from MaxQuant or Proteome Discoverer.[4] MSstatsShiny provides a web interface to the established MSstats statistical framework, simplifying quantitative analysis across acquisition types, including label-free, TMT, DIA, and PRM data.[5] More recently, TraianProt introduced an R/Shiny-based application for differential expression and functional enrichment directly from user-uploaded quantification tables.[6] In parallel, emerging collaborative infrastructures such as the Platform for Single-Cell Science (PSCS) extend the concept of browser-native analysis to single-cell and multiomics data, enabling researchers to share data sets, pipelines, and results through no-code interfaces.[7] There are too many such platforms to mention in detail here.[8−11]

Other community tools, including AlphaPept,[12] psm_utils,[13] and ProteoBench,[14] further demonstrate that streamlined web-based analysis environments built with frameworks such as Streamlit are already well established in proteomics. Streamlit provides a general-purpose Python interface that turns ordinary analysis scripts into interactive web applications that can be run locally or hosted on a server, and community platforms have used it effectively for proteomics and other omics data. However, each of these systems still relied on substantial programmer effort to translate biological requirements into

Received: October 10, 2025
Revised: November 21, 2025
Accepted: December 22, 2025
Published: January 6, 2026

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

working code, highlighting both the specialized labor that has driven much of the field's progress and the historical cost of building and maintaining such platforms.

Despite this mature ecosystem, the starting point for a new application remains the same: A domain expert must either become a competent software developer or collaborate closely with one, investing weeks to years of human time in design, implementation, and debugging. Here, I explore an alternative route, which I refer to as vibe coding, in which large language models act as autonomous coding agents that iteratively generate, test, and refine working applications from natural language prompts. Rather than introducing yet another analysis platform, I use a simple proof of concept to show what changes when an LLM handles all the programming; I build an omics data analysis application capable of normalization, differential testing, and visualization, built entirely by vibe coding with only four prompts and less than 10 min. By documenting the prompting process, evaluating code functionality, and comparing development efforts to conventional approaches, this work illustrates how LLM-based coding can sharply reduce the technical barrier to building domain-specific analysis tools and accelerate the prototyping of scientific software in computational biology.

This study does not propose another platform to compete with these existing tools. Instead, it shows that the engineering effort they traditionally require can now be front-loaded and amplified by short natural language instructions that cause an autonomous coding agent to generate an entire multifile application on demand, allowing software engineers to focus their time on design, validation, and more complex functionality rather than boilerplate implementation.

The intended audience for this work is experimental and computational proteomics researchers who are already familiar with standard analysis workflows, such as normalization, missing-value handling, and differential expression testing but who may not be professional software engineers. It may also be of interest to experienced software engineers in bioinformatics who want to learn how they may supercharge their productivity with vibe coding. Vibe coding is not a substitute for understanding statistical analysis or computational logic. Instead, it is meant to lower the barrier for such researchers to prototype interfaces and pipelines that can then be reviewed, validated, and hardened, potentially in collaboration with colleagues who have a deeper software engineering expertise.

METHODS

Replit Prompting

In this work, code generation was performed using the Replit online development environment, which exposes an LLM-based coding agent. The quoted dollar amounts refer only to Replit's model usage charges for the prompts listed above. They do not include my own time, any cloud or local computer used to run the app, or any separate hosting costs if the application is deployed as a public website. The following prompt was used with Replit.com to produce the initial prototype:

"I have proteomics data where the first column gives the protein and the next columns give the condition name followed by underscore and the replicate number. Help me make an app that can do standard data processing and statistics to find the significant proteins under the two conditions. It should optionally normalize and scale the data, optionally impute the missing values using k-nearest neighbor methods from scikit learn, perform statistical comparisons using optionally wilcoxon or t-tests (both with BH p-value correction), and then add as many visualizations as you can think: heatmap for all proteins, or filtered for only the statistically significant proteins, show 2d and 3d PCA or UMAP colored by any protein of interest of the number of proteins detected in each sample (before imputation), and volcano plots of statistically significant protein changes that allow the user to change it to any cutoff. Use plotly for all the visualizations so that we can interact with the data."

That prompt produced a prototype, but there was an error that required the following prompt:

"The data transformation seems to be working but I'm seeing this error at the bottom before any visualizations are coming up. Can you check if the statistics are working or what is the error. Also check the console because there are some errors printed there that may help answer what is going on."

To add the feature, the filter samples by the number of protein IDs in that sample:

"add an option to drop samples that have proteins less than some % of the average number of proteins"

Finally, to improve the quality of the plots for direct usage in this manuscript:

"the plot downloads are not quite publication ready - can you make the download appear as a smaller plot with larger legend and tick labels that are larger and darker, with larger points"

In a conventional development workflow, the features implemented here would require significant time for design, implementation, and debugging. A realistic estimate for an experienced software engineer to build a comparable application is several days to several weeks, depending on the experience level and testing and documentation requirements.

Synthetic Proteomics Data Set with Ground-Truth Differential Expression

To evaluate whether the Vibe-coded application reproduced the intended analysis pipeline, I generated a synthetic proteomics data set with known ground-truth differential expression. I created a matrix of $n = 1000$ proteins measured across two experimental conditions (A and B), with $r = 5$ biological replicates per condition. Baseline log2 protein intensities were sampled from a normal distribution (mean = 25, SD = 1). A defined fraction of proteins (1%) was randomly selected as differentially expressed (DE). For each DE protein, the true log2 fold change was drawn from a discrete set $\{-2, -1, +1, +2\}$ with user-specified probabilities.

Replicate intensities were simulated by adding Gaussian noise (SD = 0.3) in log2 space and converting the values to linear intensities. Missing values were introduced by using a mechanism that increased the probability of missingness for lower-intensity proteins, mimicking real mass spectrometry behavior. The synthetic generator returned the full wide matrix, a tidy long table, and a reference differential expression table containing each protein's known DE status and true log2 fold change.

Local Processing Pipeline

To establish a trusted reference implementation, the synthetic data set was processed offline using a fixed Python workflow:

1. Log2 transformation: All positive intensity values were transformed as log2(intensity).
2. Missing-value imputation: Missing values were imputed using *k*-nearest neighbors (KNN) applied in sample space, implemented using "sklearn.impute.KNNImputer" with *k* = 5.

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

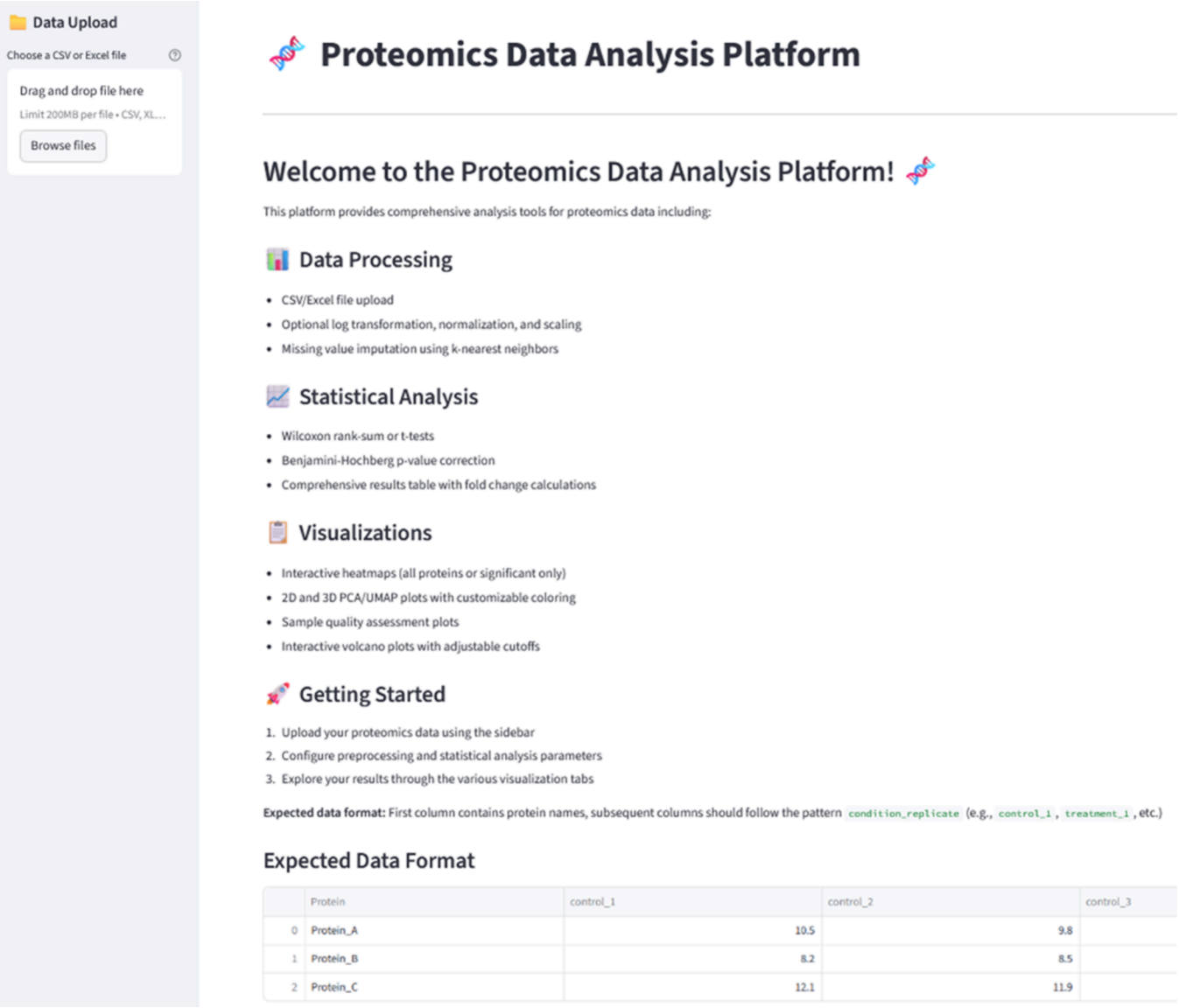


Figure 1. Screenshot of the Vibe-coded Streamlit application.

3. Protein-wise standardization: Each protein (row) was z-scored across samples using "sklearn.preprocessing.StandardScaler" (mean = 0, SD = 1).
4. Differential expression testing: For each protein, Student's t test ("scipy.stats.ttest_ind") was performed comparing condition A versus condition B. p-values were corrected for multiple testing using the Benjamini–Hochberg false discovery rate (FDR) procedure. The resulting table contained mean standardized expression per group, log2 fold change, t statistic, p-value, and FDR q-value.

This offline pipeline serves as the "ground truth" reference for evaluating the correctness of the Vibe-coded implementation.

Comparison with the Vibe-Coded Application

I selected the same processing steps in the app as described for the local pipeline above. To verify correctness, I compared the app's outputs to those from the local reference pipeline. The processed matrices (log2 → KNN → z-scaling) from the app and the local code were aligned by sample and protein, and scatter plots were generated to test numerical equivalence. Estimated log2 fold changes, p-values, and BH-corrected q-values were compared between the local pipeline and the app. All metrics matched to numerical precision (identity diagonals for each comparison). Significant proteins ($q < 0.05$) were extracted independently from both pipelines and visualized. Heatmaps generated by the app and by the reference implementation showed identical expression patterns.

RESULTS

Using a generative, prompt-driven workflow, I built a functional omics data analysis web application in just a few prompts with vibe coding. The purpose of this example is not to introduce a new analysis interface but to document how a complete omics analysis environment can now be obtained directly from an LLM with only a few prompts. The base version of the app was generated on Replit using two prompts at a total cost of $1.09. The resulting Streamlit-based site provided a complete front-end interface for file uploading, preprocessing, statistical testing, and visualization. To extend functionality, a third prompt added the ability to filter samples by a minimum detection percentage for an additional $0.38, and a final refinement prompt standardized the appearance and downloadability of plots for

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

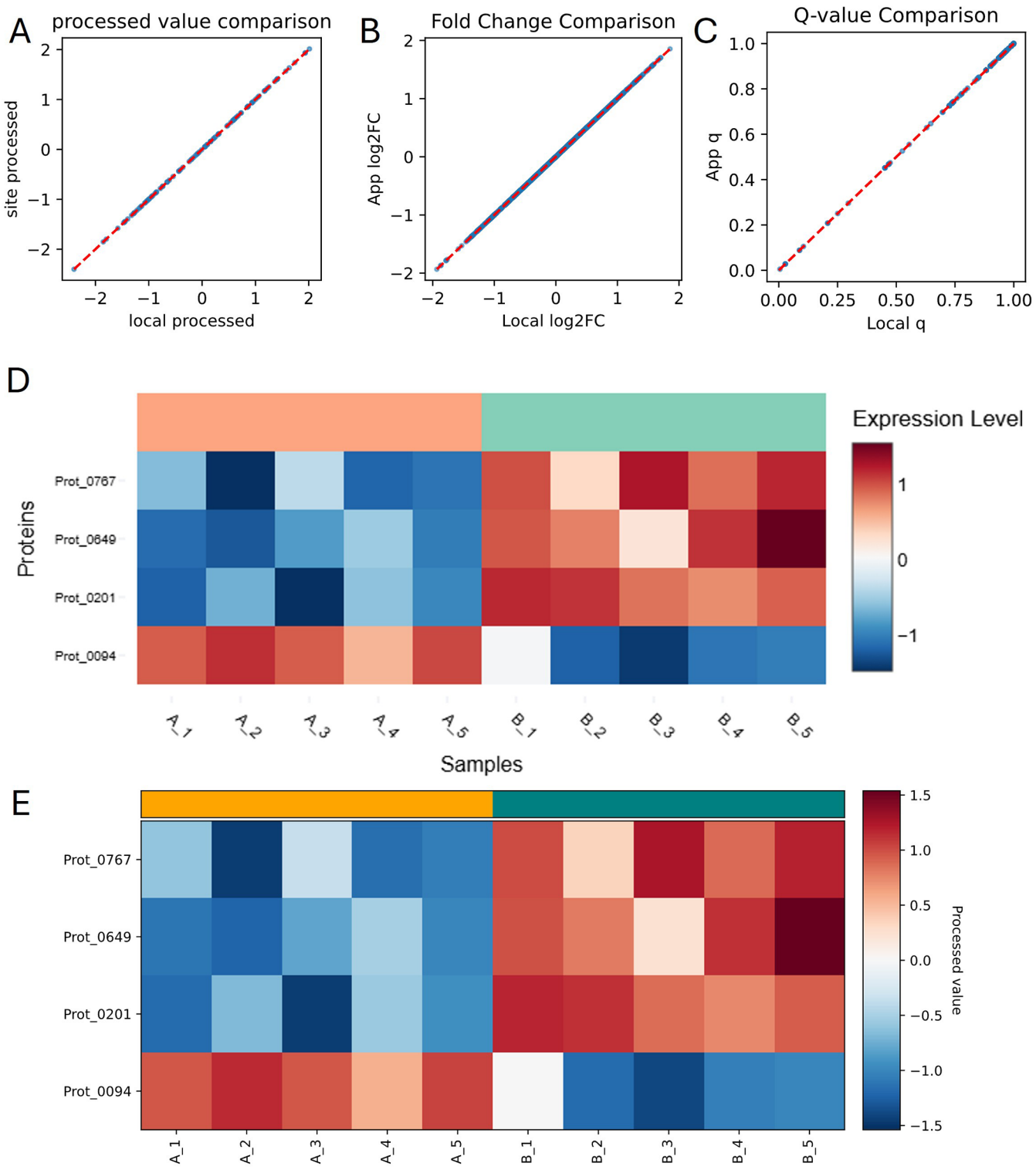


Figure 2. Verification of app functionality by comparing manual data analysis with synthetic data. Synthetic proteomics data set with known differentially expressed proteins was generated with specified log2 fold changes and missingness. (A) Comparison of the processed values from the local pipeline and the app using only the first 10 rows of either data set (the app by default only allowed download of the first 10 rows). (B) Comparison of the fold changes computed locally and via the app. (C) Comparison of Q-values computed with the local pipeline and the app. (D) Heatmap generated by the app for the four statistically significant protein changes (q-value <0.05) using the plotly library. (E) Heatmap generated by the local processing pipeline with matplotlib of the statistically significant proteins (q-value <0.05).

$0.49. Thus, the total cost to generate a fully functional proteomics analysis application was $1.96.

The application is organized into four Python modules: data_processing.py (125 lines), statistics.py (267 lines), visualizations.py (496 lines), and the main app.py file (524 lines). The total codebase comprises approximately **1400 lines of automatically generated code**, all written autonomously by the model without manual debugging or restructuring. The code

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

is compatible with local execution using the Streamlit framework (see the Supporting Information) and is openly available from github (see methods). Figure 1 shows the welcome page displayed on application launch, while Figures S1−S5 present screenshots of each analysis module—data overview, statistical analysis, heatmap visualization, principal component analysis (PCA), and volcano plots.

To determine whether the vibe-coded application faithfully implemented the intended proteomics analysis workflow, I generated a synthetic data set with known ground-truth differential expression and processed it both locally and within the app. The synthetic data set consisted of 1000 proteins measured across two conditions with realistic noise and missing-value structure, enabling direct comparison between the app outputs and a trusted reference pipeline.

I first compared the processed expression values after log2 transformation, KNN imputation, and z-score scaling. Note that the vibe coded app allowed the downloading of only the first 10 rows by default. The processed intensities produced by the app for those first 10 rows were numerically indistinguishable from those computed offline, with all points falling on the identity line (Figure 2A). Estimated log2 fold changes were identical across the two pipelines (Figure 2B), and Benjamini−Hochberg-adjusted q-values overlapped perfectly (Figure 2C), demonstrating that the app's statistical testing and multiple-hypothesis correction faithfully replicated the intended implementation.

Next, I extracted proteins with significant differential expression ($q < 0.05$) and examined whether the app reproduced the expected expression patterns. Heatmaps generated by the app (Figure 2D) and using the local implementation (Figure 2E) displayed the same effect patterns, including the separation of conditions A and B and recovery of the synthetic ground truth. This confirms that the visualization module, statistical results, and preprocessing pipeline are internally consistent and that the vibe-coded application performs end-to-end proteomics analysis correctly.

Together, these comparisons demonstrate that the autonomous code generated via vibe coding can reproduce a complete proteomics workflow with high fidelity. By validating against synthetic data with known ground truth, I show that each module of the app—imputation, scaling, statistical testing, and heatmap generation—behaves as expected and yields results indistinguishable from those of a local reference workflow.

DISCUSSION

This work demonstrates that functional, domain-specific web applications for omics data analysis can now be prototyped in minutes through natural language interaction with large language models at a marginal cost that is much lower than writing an equivalent prototype from scratch, while still relying on standard software engineering practices to turn such prototypes into robust tools. The example application itself is not intended as a novel analysis platform; it serves as a concrete test case to illustrate what changes when the development process is handled by an LLM rather than a human programmer. The example described here shows that a complete Streamlit-based analysis environment, including data processing, statistical testing, and interactive visualization, can be generated and refined through fewer than five prompts for under two dollars. In contrast to prior workflows that presuppose substantial programming expertise, comparable applications can now be produced directly from natural language prompts by users without formal software engineering training. At the same time, effective use of this approach still assumes that the user understands the structure of the analysis they want to perform, including appropriate normalization strategies, statistical tests, and diagnostics, and can critically evaluate whether the generated code behaves as intended.

For professional developers, the same approach acts as a force multiplier, offloading boilerplate implementation to an LLM and freeing time for design, validation, and more sophisticated functionality. This accessibility is meant to complement, not supplant, professional software engineering by allowing domain experts to reach a working prototype that engineers can then harden, extend, and validate. In this framing, professional software engineers remain central to building trustworthy tools, but their effort can be concentrated on designing architectures, tests, and benchmarks that can be reused across many vibe-coded prototypes, reducing the amount of time spent repeatedly building similar scaffolding from scratch. In the context of proteomics, where analytical reproducibility and interpretability are essential, such technology could allow any investigator to rapidly construct custom interfaces for specific data sets, experiments, or collaborators while also enabling experts to build and iterate on more ambitious tools than would otherwise be feasible.

However, the power of autonomous code generation also introduces new responsibilities. While the resulting application functioned as intended on a synthetic benchmark, with quantitative agreement to expected patterns, the process of vibe coding currently lacks the formal safeguards that distinguish professionally engineered scientific software, such as unit testing, systematic benchmarking, and peer review of source code. Vibe-coded applications should not be trusted without formal verification of expected behavior. Unit tests can be requested as part of the vibe coding process, but care must be taken to review those tests to ensure they are not hard-coding the desired behavior. Each model-generated implementation should therefore be treated as untrusted until it has been subjected to the same kinds of verifiable tests that software engineers already apply to scientific tools in order to ensure both computational accuracy and statistical validity. In practice, this means that a relatively small number of experts can develop shared automated test suites and benchmark workflows that can be applied repeatedly to many vibe-coded prototypes rather than leaving every individual user to validate untested code on their own. For scientific use, automated test suites, reproducibility checks, and comparison against benchmark data sets should become standard components of any vibe-coded platform. Ideally, each generated function—normalization, imputation, or statistical test—should include automated validation routines that confirm that its outputs match known results or reference libraries. Without such verification, there is a risk of introducing undetected errors or inconsistencies that could propagate through the analyses.

Another consideration is transparency. Even though large language models can now synthesize complex software architectures, users must still understand the underlying computational logic to interpret their data responsibly. In this demonstration, the generated code was human-readable and organized into modular files corresponding to the standard proteomics analysis stages. This suggests that AI-assisted development can produce not only functional but also interpretable software—an encouraging sign for educational and collaborative use. Nonetheless, code provenance and traceability remain critical. For vibe coding to gain acceptance

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

in scientific contexts, outputs should be accompanied by automatically generated documentation summarizing dependencies, algorithmic decisions, and parameter defaults, ideally in a machine-readable format to facilitate auditing and reproducibility.

A practical lesson that emerged from this demonstration is the importance of underspecification in conversational software creation. For example, in this case, the code enforced missing-value replacement with zero before generating a heatmap. When users describe their goals in natural language, they often omit operational details that a conventional software engineer would need before writing code. In a vibe coding workflow, the model must still produce a complete, executable pipeline, which forces it to choose defaults for normalization, imputation, scaling, and error handling. These defaults are usually defensible but may not match what the user intends. This phenomenon is not a flaw in the approach but an unavoidable consequence of rapid specification through a natural language interaction. Vibe coders should therefore treat LLM-generated code as a draft that reflects the assumptions that the model needs to make in order to keep the workflow functional. Inspecting these assumptions is therefore a core part of using this paradigm responsibly. In practice, this means both reading the generated code or summaries of it and, where helpful, asking the model to explain what specific functions do and why particular defaults were chosen. Such explanations can be useful pedagogically, but they should not be treated as guarantees of correctness. Users still need enough familiarity with analysis workflows to decide whether the chosen normalization, imputation, or testing strategy is appropriate and to revise the code or prompts when it is not. As vibe coding becomes more common, developing norms for prompting, documenting defaults, and validating model-selected choices will be crucial for ensuring scientific reliability.

Beyond individual use cases, the broader implication of this demonstration is that AI-assisted software creation may change the way computational tools are described and shared. In addition to standard version-controlled code repositories, developers could also distribute compact model prompts or "vibe blueprints" that record the natural language specification, model name, and inference settings used to generate a given application. Under a fixed model version and deterministic settings such as temperature set to zero and fixing a random seed, the same blueprint will in practice regenerate the same code, providing a second, higher-level description of the tool alongside the canonical source. In this view, scientific software remains a versioned, deterministic product, while the blueprint serves as a reproducible process description that can be cited, audited, and extended at the level of natural language intent.

In practice, the specific LLM or development environment used for vibe coding is likely to change over time. This demonstration used the Replit coding agent for concreteness, but the same interaction pattern can be implemented with other model providers and IDE integrations, including notebook-based environments, editor plugins, or hosted chat interfaces. For prospective users, the choice among these options should be guided less by brand and more by capabilities that matter for scientific software: support for long context windows and multi-file code bases, high-quality code generation and refactoring, deterministic execution options for reproducibility, tight integration with local tooling and version control, and transparent pricing that permits iterative development. Systematic benchmarking of different LLMs for scientific code generation is an important direction for future work, but the core ideas of vibe coding are model-agnostic.

While this study focused on proteomics data, the same approach can be applied across omics and biomedical research more broadly, including transcriptomics, metabolomics, and clinical data integration. The present example does not aim to replace established, validated platforms such as Perseus, MSstatsShiny, or ProteoArk, but rather to demonstrate that a comparable interactive interface can be built autonomously and transparently by an AI assistant in a fraction of the time. Future work should explore standardized frameworks for vibe-coded validation, integration with continuous testing pipelines, and the establishment of open benchmarks for LLM-generated scientific software.

In summary, vibe coding enables researchers to move from an idea to a functional prototype at unprecedented speed. Its success depends not only on the intelligence of language models but also on our ability as a community to apply rigorous software engineering practices and to develop robust systems for verification, reproducibility, and transparency, ideally designed and maintained by software engineers who can reuse them across many LLM-generated prototypes. As these practices mature, AI-assisted code generation may become a routine part of computational research, allowing scientists to focus less on syntax and more on discovery.

■ ASSOCIATED CONTENT

Data Availability Statement

The example vibe-coded Streamlit app is available from github: https://github.com/xomicsdatascience/ProteomicsAnalyzer. If users want to use this tool with their own data, they simply need a table where proteins are in the first column and each subsequent column gives the name of a replicate with the quantities of proteins measured in that replicate. For example with DIA-NN's pg matrix, this can be achieved by deleting most of the nonquantity columns and retaining the one protein identifier of interest. This can be run locally according to Streamlit usage instructions, which can be found on the Streamlit website; no web development experience is required beyond the ability to install Python and run a command line script: https://docs.streamlit.io/develop/concepts/architecture/run-your-app The synthetic data generator, local analysis, and comparison with the app is available from github: https://github.com/xomicsdatascience/ProteomicsAnalyzer-Data

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acs.jproteome.5c00984.

Screenshots of the example Vibe-coded platform; (Figure S1) screenshot of the data overview page; (Figure S2) screenshot of the statistics page; (Figure S3) screenshot of the heatmap page; (Figure S4) screenshot of the PCA page; and (Figure S5) screenshot of the volcano plot page (PDF)

■ AUTHOR INFORMATION

Corresponding Author

Jesse G. Meyer − *Department of Computational Biomedicine, Cedars Sinai Medical Center, Los Angeles, California 90048, United States;* orcid.org/0000-0003-2753-3926; Email: jesse.meyer@cshs.org

https://doi.org/10.1021/acs.jproteome.5c00984
J. Proteome Res. 2026, 25, 1191−1197

Complete contact information is available at:
https://pubs.acs.org/10.1021/acs.jproteome.5c00984

Notes

The example platform was developed entirely by prompting an AI, and the first draft of this manuscript was written entirely by GPT-5, based on prompts that described the results I wanted to present. The figure layout plan was my own. I edited the manuscript and am responsible for its contents. Grammarly was also used to edit and refine the language herein.
The author declares no competing financial interest.

■ ACKNOWLEDGMENTS

The NIGMS (R35GM142502) supported this work.

■ REFERENCES

(1) Kong, A. T.; Leprevost, F. V.; Avtonomov, D. M.; Mellacheruvu, D.; Nesvizhskii, A. I. MSFragger: ultrafast and comprehensive peptide identification in mass spectrometry−based proteomics. *Nat. Methods* **2017**, *14* (5), 513−520.
(2) Demichev, V.; Messner, C. B.; Vernardis, S. I.; Lilley, K. S.; Ralser, M. DIA-NN: neural networks and interference correction enable deep proteome coverage in high throughput. *Nat. Methods* **2020**, *17* (1), 41−44.
(3) Tyanova, S.; Temu, T.; Sinitcyn, P.; Carlson, A.; Hein, M. Y.; Geiger, T.; Mann, M.; Cox, J. The Perseus computational platform for comprehensive analysis of (prote)omics data. *Nat. Methods.* **2016**, *13* (9), 731−740.
(4) Nisar, M.; Soman, S. P.; Sreelan, S.; John, L.; Pinto, S. M.; Kandasamy, R. K.; Subbannayya, Y.; Prasad, T. S. K.; Kanekar, S.; Raju, R.; Devasahayam Arokia Balaya, R. ProteoArk: A One-Pot Proteomics Data Analysis and Visualization Tool for Biologists. *J. Proteome Res.* **2025**, *24* (3), 1008−1016.
(5) Kohler, D.; Kaza, M.; Pasi, C.; Huang, T.; Staniak, M.; Mohandas, D.; Sabido, E.; Choi, M.; Vitek, O. MSstatsShiny: A GUI for Versatile, Scalable, and Reproducible Statistical Analyses of Quantitative Proteomic Experiments. *J. Proteome Res.* **2023**, *22* (2), 551−556.
(6) Camara-Fuentes, S de la, Gutierrez-Blazquez, D., Hernaez, M. L., Gil, C. TraianProt: a user-friendly R shiny application for wide format proteomics data downstream analysis [Internet]. *arXiv*; 2024 [cited 2025 Oct 9]. Available from: http://arxiv.org/abs/2412.15806.
(7) Hutton, A.; Ai, L.; Meyer, J. G. PSCS: Unified Sharing of Single-Cell Omics Data, Analyses, and Results. *J. Proteome Res.* **2025**, *24* (9), 4825−4830.
(8) Olabisi-Adeniyi, E.; McAlister, J. A.; Ferretti, D.; Cox, J.; Geddes-McAlister, J. ProteoPlotter: An Executable Proteomics Visualization Tool Compatible with Perseus. *J. Proteome Res.* **2025**, *24* (6), 2698−2708.
(9) Schneider, M.; Zolg, D. P.; Samaras, P.; Ben Fredj, S.; Bold, D.; Guevende, A.; Hogrebe, A.; Berger, M. T.; Graber, M.; Sukumar, V.; Mamisashvili, L.; Bronsthein, I.; Eljagh, L.; Gessulat, S.; Seefried, F.; Schmidt, T.; Frejno, M. A Scalable, Web-Based Platform for Proteomics Data Processing, Result Storage and Analysis. *J. Proteome Res.* **2025**, *24* (3), 1241−1249.
(10) Afgan, E.; Baker, D.; Batut, B.; van den Beek, M.; Bouvier, D.; Čech, M.; Chilton, J.; Clements, D.; Coraor, N.; Grüning, B. A.; Guerler, A.; Hillman-Jackson, J.; Hiltemann, S.; Jalili, V.; Rasche, H.; Soranzo, N.; Goecks, J.; Taylor, J.; Nekrutenko, A.; Blankenberg, D. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Res.* **2018**, *46* (W1), W537−W544.
(11) Heming, S.; Hansen, P.; Vlasov, A.; Schwörer, F.; Schaumann, S.; Frolovaitė, P.; Lehmann, W. D.; Timmer, J.; Schilling, M.; Helm, B.; Klingmüller, U.; Bateman, A. MSPypeline: a python package for streamlined data analysis of mass spectrometry-based proteomics. *Bioinf. Adv.* **2022**, *2* (1), No. vbac004.
(12) Strauss, M. T.; Bludau, I.; Zeng, W. F.; Voytik, E.; Ammar, C.; Schessner, J. P.; Ilango, R.; Gill, M.; Meier, F.; Willems, S.; Mann, M. AlphaPept: a modern and open framework for MS-based proteomics. *Nat. Commun.* **2024**, *15* (1), 2168.
(13) Gabriels, R.; Declercq, A.; Bouwmeester, R.; Degroeve, S.; Martens, L. psm_utils: A High-Level Python API for Parsing and Handling Peptide-Spectrum Matches and Proteomics Search Results. *J. Proteome Res.* **2023**, *22* (2), 557−560.
(14) Proteobench/ProteoBench [Internet]. ProteoBench; **2025** [cited 2025 Nov 21]. Available from: https://github.com/Proteobench/ProteoBench.