Paper OS10

The Use of Git in Statistical Programming

Rita Pecuch, Takeda, Ann Arbor, United States
Jim Coates, KSM Technology Partners, Montgomery Country, United States

ABSTRACT

Git has achieved over 90% adoption among programmers, providing powerful capabilities such as code reproducibility, streamlined collaboration, and robust audit trails. Yet, despite these strategic advantages, the pharmaceutical industry has lagged notably behind other sectors in embracing this transformative technology.

This trend is shifting as more major pharmaceutical companies embrace open-source tools, recognizing Git as essential for change tracking, auditability, and transparency in analyses. To accelerate this transition, our PHUSE Emerging Trends and Innovation working group has assembled representatives from diverse organizations to address industry-specific challenges and establish best practices for implementing Git in statistical reporting environments.

This presentation will showcase real-world applications where Git is already enhancing statistical programming workflows, introduce our collaborative working group initiative, and explore our expanding repository of resources—including our content-rich blog and upcoming white paper. Join us as we bridge the gap between Git's proven potential and its practical implementation in pharmaceutical statistical reporting.

INTRODUCTION

This paper emerges from the PHUSE Emerging Trends and Innovation Working Group, which was formed to address Git adoption challenges across the pharmaceutical industry. The working group includes representatives from major pharmaceutical companies, Contract Research Organization (CROs), and technology vendors, all of whom recognized a common pattern: Git adoption in statistical programming felt inevitable.

Git adoption in pharmaceutical statistical programming has moved from "if" to “when and how." The converging forces of efficiency demands, vendor independence, and industry standardization are making this transition increasingly pressing. Organizations that move thoughtfully and proactively can shape their implementation to fit their culture and needs. Organizations that delay will eventually need to adopt reactively, with less control and higher switching costs.

This paper serves as a strategic guide for organizations considering or beginning Git adoption for statistical programming. It aims to establish the business case for why Git adoption is becoming necessary rather than just beneficial. It seeks to demystify Git by providing statistical programmers and their managers with a clear explanation of what Git is and how it applies to their work. It documents some key strategic decisions organizations face during adoption, with considerations for each choice. It shares real-world learnings including actual pain points and solutions from companies that have already implemented Git. Finally, it points to practical resources including training materials, templates, and community support.

This paper addresses multiple stakeholder groups involved in Git adoption for clinical trial programming. Statistical programming leaders will find guidance on adoption decisions and business value. Statistical computing environment (SCE) architects will find technical implementation considerations for integrating Git with clinical workflows. Quality and compliance teams will find regulatory requirements and audit trail considerations. Statistical programmers will understand the rationale behind Git practices and how they align with clinical programming workflows. Training and change management professionals will find foundational concepts to support team transition. While this paper covers key considerations for each stakeholder group, it is not intended as a comprehensive implementation guide.

1. THE GIT IMPERATIVE

1.1 The Pressure Statistical Programmers Face Today

Statistical programmers in the pharmaceutical industry face a fundamental paradox. On one hand, they work in highly regulated environments where every line of code, every data transformation, and every output must be traceable, reproducible, and defensible to regulatory authorities. On the other hand, many are still managing this critical work using approaches designed for a simpler era: shared network drives, manual version control through filename conventions, and ad-hoc documentation of what changed and why.

Consider a typical scenario: A statistical programmer is finalizing Analysis Dataset Subject Level (ADSL) for a pivotal trial. The shared drive contains ADSL_v1.sas, ADSL_v2.sas, ADSL_v2_final.sas, ADSL_v2_final_revised.sas, and ADSL_v3_USE_THIS.sas. The QC programmer needs to review the latest version but isn't sure which file is truly final. An auditor asks to see exactly what changed between the interim and final analysis. The team scrambles to reconstruct the history, hoping their email chains and manual logs are complete.

This is not a failure of statistical programmers. It's a failure of tooling. The industry has been operating with version control approaches that were adequate when studies were smaller, teams were co-located, and regulatory expectations were less stringent. Those days are gone, and many of us are feeling the strain.

1.2 Why Now: Three Converging Forces

Most organizations will adopt Git for statistical programming in the next few years. The question is whether that happens deliberately, with a plan that fits your culture and needs, or reactively when business pressures force a hurried transition. We've observed three forces converging that are making this decision point arrive sooner than many expect.

The first force is the efficiency imperative. Technology advancement is outpacing manual processes at an accelerating rate. Statistical programming teams are being asked to deliver more trials, faster, with the same or fewer resources. The complexity of modern clinical trials (adaptive designs, interim analyses, multiple endpoints, integration of real-world evidence) demands automation and scalability that manual version control struggles to provide.

When a programmer spends 20 minutes figuring out which version of a program created which dataset, that's time lost. When a QC programmer has to email back and forth to confirm they're reviewing the right files, that's friction. When a team meeting is spent reconciling who made which changes to avoid overwriting work, that's overhead. These inefficiencies compound across every study, every programmer, every day. Git helps eliminate this entire class of problems by tracking every change automatically, making every version instantly accessible, and allowing contributors to work simultaneously without conflicts.

The second force is vendor independence and portability. Many organizations have learned difficult lessons about vendor lock-in with proprietary systems. When your version control, workflows, and institutional knowledge are tied to a specific SCE vendor, you lose negotiating power, flexibility, and the ability to adopt better tools as they emerge.

Git is open-source and universal. It's not owned by any vendor, does not require a specific platform, and has been the industry standard in software development for over a decade. Whether your organization uses Domino, Viya, or builds a custom SCE, Git works the same way. Your programmers' skills are transferable. Your repository structures and workflows remain portable. Your intellectual property (how you organize studies, enforce quality checks, manage branches) belongs to you, not a vendor. This vendor independence also future-proofs your operation. When new technologies emerge, Git compatibility will be table stakes.

The third force is talent and industry standardization. Over 90% of software developers use Git. Graduates from data science, biostatistics, and computer science programs expect Git as a basic tool, like email or a code editor. When pharmaceutical companies recruit talent from technology companies, academic research, or even other industries, Git proficiency is assumed.

This creates both a challenge and an opportunity. Organizations that don't offer modern development tools may become less attractive to top talent. Organizations that do adopt Git can recruit from a broader pool and reduce onboarding time for programmers who already know the tool. Beyond individual hiring, major pharmaceutical companies have already made this transition. This creates an informal industry baseline that spreads through collaborations, CRO relationships, and regulatory familiarity.

The question isn't whether your organization will adopt Git, but whether you'll do so while you still have the luxury of careful planning.

1.3 What Git Is (and Isn't)

Before discussing adoption strategies, let's establish a common understanding of Git's core functionality. Many statistical programmers have heard of Git but may lack direct experience with it.

What Git Is: Git is a distributed version control system that addresses several critical needs in statistical programming. It records every modification to every code file in your project, creating an automatic audit trail that shows who changed what, when, and why. This complete change tracking enables multiple programmers to work on the same codebase simultaneously without conflicts or overwriting each other's work. Git allows you to create separate workstreams (branches) for development, QC, and production, supporting controlled progression through

your validation process. Any previous version of your work can be restored instantly, eliminating the need for dated file copies or "FINAL_v3_FINAL" naming conventions. Git is open-source technology that functions consistently across all platforms and computing environments.

What Git Is Not: Understanding what Git isn't is equally important. Git itself is the underlying technology; while GitHub, GitLab, and similar platforms host Git repositories, Git is free, open-source, and universal, meaning you're not locked into any vendor. While developers use Git extensively, it's particularly well-suited for regulated programming environments that require robust audit trails and reproducibility, not exclusively a developer tool. Git works within your existing statistical computing environment, complementing rather than replacing it. And while cloud hosting offers collaboration benefits we generally recommend, Git operates locally first and doesn't require cloud services to function.

How Git Works: Git manages file versions through "commits"—snapshots of your project at specific points in time. Each commit captures what changed, who made the change, when it occurred, and a message explaining why. These commits form a complete, auditable history that cannot be lost or accidentally overwritten*, providing the foundation for regulatory compliance and reproducible research.

1.4 Why Git Over Traditional File Systems

Statistical programmers are already doing version control, just manually. Understanding why Git offers meaningful advantages requires examining the specific pain points of traditional approaches.

In a traditional shared network drive workflow, a programmer creates ADSL.sas on a shared drive. As changes are needed, they save ADSL_v2.sas to preserve the original. After QC review comes ADSL_v2_final.sas. A protocol amendment requires ADSL_v3.sas. After more edits: ADSL_v3_revised.sas, ADSL_v3_final.sas, ADSL_v3_FINAL_USE_THIS.sas. We've all been there.

This approach creates several persistent problems. It's often unclear what's current when multiple "final" versions exist. Two programmers editing simultaneously overwrite each other's work. There's no record of what changed between v2 and v3. Determining who made which changes requires detective work. Manual documentation is error-prone and often incomplete. Finding and restoring a specific previous version is difficult. Duplicate files consume significant storage.

The Git equivalent workflow looks different. A programmer works on ADSL.sas in a feature branch. They make changes and commit with a message: "Added baseline BMI calculations per SAP v3.0." They create a pull request for QC review. The QC programmer reviews the exact changes (what lines were added or modified) and approves. The code merges to the main branch, establishing the definitive version. When a protocol amendment is needed, a new branch is created with commits like "Updated randomization date per Protocol Amendment 2." All history is preserved. Any previous version can be instantly retrieved.

This solves the core problems. There's always clarity about what's current (one file, current version on main branch). Multiple programmers work simultaneously with Git merging changes intelligently. Every change is preserved with explanation. Attribution is automatic with every commit showing who made the change and when. The audit trail is built-in and immutable*. Recovery is instant with a single command. Storage is efficient because Git stores only changes (deltas), not duplicate files.

The difference extends beyond efficiency to capability. Git enables workflows that are difficult or impossible with manual version control. You can safely experiment by branching, testing, and abandoning if needed. QC processes can compare exact changes rather than entire files. Regulatory compliance becomes design rather than documentation burden because the audit trail is automatic.

1.5 The Regulatory Alignment

While regulatory requirements haven't fundamentally changed, Git naturally satisfies them more completely and reliably than manual approaches. Consider 21 CFR Part 11 requirements. Authority checks are managed through access controls determining who can commit to which branches. Operational checks happen through pull requests that enforce review before production. The audit trail is an immutable* record of all changes with user ID and timestamp. System validation is straightforward because Git behavior is deterministic and well-documented.

The principles of Good Documentation Practices align similarly well with Git. Documentation is contemporaneous because commit messages describe changes as they're made. Changes are attributable with every commit linked to a specific user. The record is legible with changes viewable in human-readable format. History is permanent and cannot be altered or deleted. Accuracy is ensured because the system records exactly what changed, eliminating transcription errors.

The key insight is that Git doesn't add regulatory burden. It reduces it. The audit trail is automatic. The documentation is built into the workflow. The validation evidence is generated as a byproduct of normal work.

1.6 Common Misconceptions Addressed

Several concerns prevent organizations from adopting Git, and they're worth addressing directly. These concerns are understandable given how different Git appears from traditional approaches.

The first concern is that Git is too technical for statisticians and SAS programmers. This perception exists because Git emerged from software development culture, which can feel foreign to statistical programmers. Git does have a learning curve, but so did SAS when programmers first learned it. Modern Git interfaces (both graphical and command-line) are designed for non-developers. Organizations that have taught Git to programmers with no command-line experience report that basic proficiency develops within days or weeks, not months. The investment is comparable to learning a new SAS procedure or statistical method.

A second concern is that existing SCE platforms already provide version control. This is partially true. Some SCE platforms offer basic versioning, typically snapshots at the project level rather than granular file-by-file tracking with complete history. More importantly, vendor-specific version control locks you into that platform. Git provides universal version control that transcends any single tool and keeps your workflows portable.

Some view Git as designed for software developers rather than regulated environments. The opposite is actually true. Git's immutable* audit trail and clear change history make it particularly well-suited for regulated work. The pharmaceutical companies furthest along in Git adoption have successfully validated its use for GxP environments. Git provides more complete auditability than manual version control.

Concerns about the learning curve slowing down operations during adoption are legitimate. Initial adoption does require training and adjustment. However, programmers typically become productive within weeks, and the efficiency gains compound quickly. The short-term investment yields long-term returns.

Finally, some believe their current approach works adequately. This is survivorship bias. Current approaches work until they do not. Most programmers can recall times when work was accidentally overwritten, QC reviewed the wrong version, or an audit required scrambling to reconstruct changes. These are not hypothetical risks. They are recurring problems that Git systematically eliminates when best practices are employed.

2. GIT ADOPTION CONSIDERATIONS AND EARLY DECISIONS

During Git adoption, an organization or team will need to make several key strategy decisions along the way. Every organization has different constraints, existing systems, and cultures. Rather than prescribing one correct approach, this section presents options and trade-offs to inform your organization's decisions. This is meant to be a high-level summary of concepts and is not a Git tutorial. Numerous excellent resources exist for learning Git commands and workflows. Additionally, this is meant to summarize key points that are relevant to all users as a starting point, and a future white paper will address topics more specific to statistical programming.

2.1 Commit Frequency: Building Your Audit Trail

A commit - a saved snapshot of your changes recorded in the repository history - turns your work into a series of named, immutable* checkpoints in Git’s timeline. Each commit captures exactly what changed (and when, and by whom), giving Git stable reference points to restore a repository to a known good state (by reverting to a specific commit) and to compare evolution over time (by comparing one commit against another). Without commits, Git has no reliable milestones to anchor these operations, making rollbacks, audits of how a file evolved, and precise change comparisons far less effective, effectively wiping out the core advantages Git is designed to provide.

Therefore, emphasizing the importance of making commits frequently is essential to instill confidence that important work will not be lost, but how teams interpret “frequently” is the decision point here. When committing changes, programmers can add associated messages. Although a unique identifier is assigned to each commit by default, this will not be meaningful in an audit trail to the individual programmer attempting to look for an old version, making commit messages essential for tracking. Because of this, commits should be made for changes that can be summarized in a short sentence, which generally corresponds to a single functional change. For statistical programming this could mean committing after finishing code to generate a specific table, listing, or figure (TLF), or changes made to an existing program after a protocol amendment.

Strategy and guidelines will vary slightly by team, but the importance of establishing them early cannot be understated in order to take full advantage of Git's core capabilities. Trying out different strategies and checking which has the best compliance and which is most intuitive to the specific team may be the best approach, as the highest compliance will create the best Git audit trail.

2.2 Branching Strategy and Best Practices

A branch in Git is an independent line of development that allows you to work on code changes without affecting the stable codebase. Think of it as creating a separate workspace where you can develop new programs, fix bugs, or test approaches while the main codebase remains intact and functional. Branches are beneficial because they enable isolation of work-in-progress, support parallel development by multiple programmers, facilitate structured peer review through clear comparison points, and provide a safety net for experimentation without risk to validated code.

Branching strategies need not be uniform across all repositories. Different repository types within a clinical programming organization may warrant different approaches, and these strategies can operate independently without conflict. A standard macro library used across multiple studies might benefit from a conservative branching approach with rigorous review gates, while a study-specific analysis repository might employ a more flexible strategy aligned with reporting event timelines. The key is selecting strategies that match the risk profile, collaboration patterns, and regulatory requirements of each repository type. Some organizations apply a single branching strategy enterprise-wide through policy and training. Others categorize their repositories by type; such as infrastructure code, study-level deliverables, or exploratory analyses, and define appropriate strategies for each category. Statistical programming teams producing consistent deliverable types across trials often find that standardizing on one well-defined strategy reduces cognitive overhead and supports quality through consistency.

Several branching patterns have emerged that align well with clinical programming workflows. Trunk-based development maintains a single main branch with very short-lived feature branches, emphasizing frequent integration and continuous validation. This approach works well for teams with strong automated testing and mature collaboration practices. Feature branching creates a dedicated branch for each distinct development task; whether that's a new analysis program, a macro enhancement, or a bug fix. Branch names should be strategic and meaningful, such as "csr-efficacy-tables" or "fix-ae-sorting-logic," clearly indicating the branch's purpose. Deliverable-based branching organizes work around regulatory milestones or reporting events, with branches like "database-lock-1" or "dmc-report-q2" that encapsulate all programming activities for that deliverable. Some teams employ a staging or integration branch as an intermediate step between feature branches and the main branch, which proves particularly valuable when multiple programmers' changes need to be tested together before final integration.

Regardless of strategy selected, the importance of clearly defining workflows and enforcing branching practices cannot be overstated. Git provides powerful capabilities, but it cannot prevent poor practices without discipline and adherence to established processes. For example, if a team organizes a study repository with folders for each reporting event and does not establish clear guidelines, programmers may still copy and paste code across folders rather than using Git's merge capabilities to share validated code. Git makes recovery from such practices easier, merging branches rather than manually duplicating files, but prevention requires consistent training and compliance with defined workflows. At minimum, the main (or master) branch should have protections in place to ensure that changes are never incorporated without peer review and appropriate validation, maintaining the integrity of your production-ready codebase.

2.3 Git Tooling

Git interaction can occur within your statistical computing environment through either a command-line interface (CLI) or graphical user interface (GUI), though available options depend on your SCE's capabilities. Not all SCEs support Git integration natively, and those that do may offer only CLI access, only GUI access, or both. Understanding your SCE's Git support is a critical first step in planning your implementation approach.

When both options are available, teams must decide which approach to standardize on or whether to allow individual programmer choice. The GUI approach tends to be more intuitive for programmers new to command-line tools, offering visual representations of repository status, branch structures, and change histories that can accelerate initial comprehension. The CLI offers greater control and precision over Git operations, often reducing multi-step GUI workflows to single commands. For routine operations like committing changes or creating branches, CLI efficiency can minimize both time and potential for error.

An important but often overlooked factor is transferability. CLI commands are universal, the same Git commands function identically across SAS, R, Python, or any other environment. A programmer who learns `git commit`, `git branch`, or `git merge` carries that knowledge to any future platform. GUI implementations, however, vary by tool.

The Git interface in one SCE may differ substantially from another, requiring retraining during platform transitions or when collaborating across systems.

This transferability consideration should be weighed against your organization's stability and strategic direction. Teams using a well-established SCE with no anticipated changes may prioritize minimizing the learning curve through GUI adoption, potentially improving initial compliance with new workflows. Teams undergoing or planning SCE transitions, working across multiple platforms, or valuing portable skill development may find the CLI investment worthwhile despite the steeper initial learning curve.

The perceived difficulty of CLI adoption, however, deserves realistic assessment. Everyday Git work relies on a core set of perhaps six to eight commands that programmers use repeatedly. Extensive documentation, cheat sheets, and online resources make command lookup straightforward when needed. Many teams find that after an initial adjustment period, CLI efficiency becomes preferable even to programmers who initially resisted it. The decision ultimately balances immediate accessibility against long-term efficiency, error reduction, and skill portability within your organization's specific context and timeline.

2.4 Cloud Hosting Options

While using Git as an individual programmer provides valuable change tracking benefits, collaborative use is strongly recommended to leverage centralized peer review, workflow automation, and team coordination features. Collaborative Git requires selecting a hosting platform where repositories are stored and shared. Many organizations already use Git-based platforms for other technical teams: such as Bitbucket, Azure DevOps, GitHub Enterprise, or GitLab, which may offer licensing advantages or streamlined procurement if extended to statistical programming groups.

Among dedicated Git hosting services, GitHub and GitLab are the most common. A poster presented at the PHUSE Computational Science Symposium in Utrecht highlights key differences between these platforms [1]. GitHub offers lower cost and a substantially larger user community [1], which can be advantageous for organizations without extensive internal Git expertise, as community resources, third-party integrations, and troubleshooting guidance are more abundant. GitLab, while typically higher cost with a smaller community, provides certain enterprise features that may align better with regulated environments.

One notable technical difference concerns audit trail retention, which matters specifically for teams leveraging automated workflow features. GitHub retains a complete history of workflow run dates, times, and pass/fail results indefinitely, but detailed execution logs expire after 400 days (or sooner if configured by your organization). If your validation or regulatory strategy depends on indefinite retention of detailed workflow logs, such as test output, environment specifications, or execution details needed during FDA inspections, GitLab's longer retention may justify its higher cost despite the smaller community. This consideration primarily affects teams implementing continuous integration, automated validation, or advanced workflow automation, but warrants evaluation even if such features aren't immediately planned.

Before selecting a platform based solely on publicly available services, investigate whether your organization already licenses enterprise versions of Git platforms for other departments. Extending existing infrastructure to statistical programming may offer both cost savings and IT support advantages. Several PHUSE presentations have documented how pharmaceutical organizations are implementing GitHub and GitLab workflows specifically for clinical programming contexts [2,3], providing valuable industry-specific guidance for platform evaluation and adoption.

2.5 Code Version to Data Version Traceability

Git is fundamentally a code version control system, not a comprehensive data lineage solution. While Git excels at tracking every change to programming code, it is not designed to manage clinical trial datasets, which are typically large, binary, and change through well-defined database lock processes rather than iterative development. Attempting to version control datasets in Git creates repository bloat, performance issues, and doesn't align with how clinical data is actually managed and validated.

Within a Git repository, programming teams define patterns for files that should be ignored so Git does not attempt to version control them; typically datasets, log files, generated outputs, environment-specific configurations, or files containing local settings. While Git can technically track changes to any file type, its line-by-line change visualization is only meaningful for text-based code files. For binary files like SAS datasets or Excel outputs, Git can detect that a file changed but cannot show what changed, limiting its value for data tracking.

Ensuring complete traceability in statistical programming requires linking every derived output back to the exact code version, input data, and execution context that produced it. Git addresses the code component of this requirement but cannot solve it alone. A comprehensive traceability approach combines Git for code versioning with other elements:

your SCE's native logging capabilities, controlled data management processes, execution metadata capture, and disciplined ways of working.

Many statistical computing environments already provide robust audit trails and logging mechanisms. SAS logs capture program execution details, data inputs, and processing steps. Validated data management systems track dataset versions through database snapshots and locks. The challenge is connecting these existing capabilities with Git's code versioning to create a unified traceability story suitable for regulatory inspection.

One effective approach pairs each program execution with a lightweight run record capturing critical linkage information: the Git commit identifier (or tag) for the exact code version used, identifiers for input datasets including database lock or snapshot version, references to specifications or mapping documents that governed the analysis, execution timestamps and environment details, and unique run identifiers that connect to your SCE's detailed logs. This run record can be as simple as a standardized header written to log files or a separate metadata file generated alongside outputs.

Parameters and environment variables offer additional mechanisms for traceability. Program headers can reference Git tags corresponding to validated code releases. Environment variables can store the current Git commit hash and inject it into log files automatically. Validated production runs can be tagged in Git and cross-referenced in data management systems, creating bidirectional linkage between code versions and dataset versions without commingling code and data in the same repository.

The key insight is that establishing sound workflows and leveraging your existing SCE capabilities enables reliable traceability, adopting Git alone does not make this automatic. Git provides the code versioning component, but complete lineage requires intentional integration with your data management practices, execution logging, and validation processes. Organizations should assess how Git fits within their broader quality and compliance framework rather than expecting it to replace existing traceability mechanisms.

3. ADOPTION AND CHANGE MANAGEMENT CHALLENGES

Adopting Git represents a fundamental shift in how statistical programmers work, and acknowledging this reality is essential for successful implementation. Git introduces new concepts, unfamiliar terminology, and a different mental model for managing code compared to traditional file-based approaches that many programmers have used for their entire careers. This is not just about learning new commands, it is about changing ingrained habits around how code is saved, shared, reviewed, and validated. Resistance is not irrational; it is a natural response to disruption of established workflows that have successfully delivered regulatory submissions for years. The organizations that succeed with Git adoption are those that approach it as a change management challenge first and a technical implementation second.

The foundation of successful adoption is clearly articulating why Git matters specifically for statistical programming in clinical trials, not just repeating generic version control benefits. Programmers need to understand how Git addresses their actual pain points: the 4:47 PM Friday panic when yesterday's working code now fails, the proliferation of "FINAL_v3_REALLY_FINAL" file names, the hours spent reconciling conflicting changes from multiple QC programmers, or the stress of regulatory inspections when audit trails are scattered across emails and shared drives. Understanding the rationale once is not enough, the "why" needs reinforcement throughout the adoption journey as teams encounter frustrations with the learning curve. When a programmer struggles with branch management or merge conflicts, they need to remember why this struggle is worthwhile. Leaders should consistently connect Git practices back to concrete benefits: time saved, errors prevented, audit readiness improved, and confidence gained.

One of the most common adoption failures occurs when technology decisions are made by leadership or IT departments without meaningful input from the programmers who will use the tools daily. Statistical programmers are sophisticated technical professionals who deserve a voice in decisions affecting their workflows. Involving programmers early through pilot groups, feedback sessions, or working committees serves multiple purposes. It surfaces practical concerns that leadership might not anticipate, such as integration challenges with existing validation workflows or conflicts with standard operating procedures (SOPs). It creates champions within the programming team who feel ownership over the approach rather than resentment of imposed changes. And it provides opportunities to address the real tradeoffs: for example, recognizing that while CLI commands are transferable across platforms, many programmers find GUI interfaces more intuitive initially, and both perspectives have merit. When programmers feel heard, compliance improves not through enforcement but through genuine buy-in.

Beyond the broad strokes of change management, specific tactical considerations affect adoption success. Not all decisions carry equal weight. Technology platform choices often come from enterprise IT or compliance considerations beyond programming team control, and that's acceptable if communicated transparently. What matters more is giving programmers influence over decisions that directly impact their daily work: branching strategies, repository organization patterns, and which Git interface to standardize on. Formal endorsement from leadership, not

just subject matter experts, significantly improves compliance with chosen approaches, though be cautious about codifying strategies in SOPs if they may need to evolve as the team gains experience. Training deserves special attention. Not everyone will attend initial sessions or practice immediately, so plan for ongoing support through office hours, champions within teams, and easily accessible documentation. Common pain points will emerge: programmers forgetting which branch they're working in, accidentally committing to main, or confusion about handling merge conflicts. Rather than viewing these as failures, treat them as expected parts of the learning curve and ensure clear, judgment-free guidance exists for recovering from mistakes. For teams implementing automation like GitHub workflows or validation checks, establish processes for applying improvements to existing repositories, not just new ones. Finally, it is important to be realistic about timelines. Achieving comfortable proficiency with Git takes months, not weeks, and measuring success by perfect compliance in the first quarter sets everyone up for frustration. Celebrate progress, learn from struggles, and adjust your approach based on what your specific team needs.

4. COMMUNITY RESOURCES AND SUPPORT

4.1 PHUSE Working Group Context

Multiple organizations were independently attempting Git adoption and facing the same questions. What branching strategy makes sense for clinical trial deliverables? How should we organize repositories (by study, by compound, by deliverable type)? Should we track data outputs in Git or only code? How do we train programmers with varying technical backgrounds? How do we enforce quality without creating bureaucratic overhead? Rather than each organization solving these problems in isolation, our PHUSE working group formed to share learnings, document approaches that have worked, and create resources that could accelerate adoption across the industry.

The working group is developing additional resources beyond this paper. Including:

- Follow-up papers to address Git-specific technical topics in statistical programming
- A blog series to provide practical examples and code snippets
- Templates for repository structures, branching strategies, and training curricula
- Documentation of case studies from organizations at various stages of adoption

The goal is to create an industry knowledge base that accelerates adoption and helps organizations learn from each other's experiences rather than repeating the same challenges. Our working group provides a collaborative and supportive environment to discuss industry-specific considerations and challenges of Git adoption and usage.

4.2 Open Source Resources

Git thrives because of a wide, active community of maintainers, educators, tooling vendors, and everyday developers who contribute code, documentation, tutorials, issue triage, and integrations—making it easier for newcomers to learn and for teams to adopt consistent workflows. Helpful starting points include the official Git documentation, the free online book Pro Git, Atlassian's Git Tutorials, and GitHub's Git Guides for practical, example-driven learning.

https://git-scm.com/doc
https://git-scm.com/book/en/v2
https://www.atlassian.com/git/tutorials
https://docs.github.com/en/get-started/getting-started-with-git/guides

CONCLUSION

Git is a foundational tool for statistical programming because it turns analysis into a transparent, reproducible, and auditable workflow: code, data-processing steps, and results can be tracked over time, reviewed, and reliably restored, which reduces errors and strengthens confidence in findings. Beyond individual productivity, Git enables effective collaboration by supporting parallel development, structured review, and clear accountability for changes, capabilities that become essential as projects scale in complexity, team size, and regulatory or publication expectations. Successful adoption, however, depends on deliberate choices: selecting an appropriate branching strategy, establishing consistent commit and code review standards, and choosing supporting tooling and training that fit the team's needs and statistical stack. By aligning these decisions with project risk, governance needs, and day-to-day analyst workflows, teams can realize Git's benefits without unnecessary friction, making analyses both faster to iterate and easier to trust.

FOOTNOTES

* Git commits are effectively immutable; changing one produces a new commit. Although not trivial, branch history can be rewritten. However, branches in shared repositories can be protected to reject these types of events, and a record of the rejected event will be retained.

REFERENCES

1. Opalka, A. (2025). GitLab vs GitHub: The Battle of the Repositories [Poster PP05]. Roche. https://phuse.s3.eu-central-1.amazonaws.com/Archive/2025/CSS/EU/Utrecht/POS_PP05.pdf
2. Li, D. (2025). Introduction to CI/CD SDTM aCRF.pdf application using GitHub [Presentation PRE_Beijing05]. Bayer. https://phuse.s3.eu-central-1.amazonaws.com/Archive/2025/SDE/APAC/Beijing/PRE_Beijing05.pdf
3. Chénedé, X., Charliquart, P. (2025). A Git-Driven Automated Workflow for Efficient ADaM and TFL Validation and Quality Control Documentation [White Paper SM01]. https://phuse.s3.eu-central-1.amazonaws.com/Archive/2025/Connect/EU/Hamburg/PAP_SM01.pdf

ACKNOWLEDGMENTS

We would like to acknowledge all of the members of the Git in Statistical Programming PHUSE working group, especially the leads, Eleanor Sparling and Kieran Martin.

CONTACT INFORMATION

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

Author Name: Rita Pecuch
Company: Takeda
Email: rita.pecuch@takeda.com

Author Name: Jim Coates
Company: KSM Technology Partners
Email: jcoates@ksmppartners.com

To join the Git in Statistical Programming PHUSE working group, contact the leads: The Use of Git in Statistical Programming - WORKING GROUPS - PHUSE Advance Hub