PharmaSUG 2026 – Paper ET-327

Building Better Data Science Workflows: Best Practices with Git, GitHub, and Data Version Control (DVC) for Effective Collaboration

Ryan Paul Lafler, Premier Analytics Consulting, LLC

ABSTRACT

This paper presents a practical framework for building reliable and collaborative data science workflows using Git, GitHub, and Data Version Control (DVC). It begins by explaining the importance of version control in data science and introduces Git as the foundation for tracking code changes. GitHub is then presented as a collaboration layer for shared repositories and team-based development. DVC is introduced as an extension to version control that enables efficient tracking and versioning of datasets alongside code. The paper presents practical examples demonstrating structured branching strategies, meaningful commit practices, managing work-in-progress (WIP) safely, and reducing merge conflicts in team environments. DVC workflows are demonstrated to show how dataset changes can be tracked, compared, and restored without storing large files directly in Git repositories. Together, these tools provide a structured and reproducible workflow for collaborative data science, machine learning, and analytics projects. The paper concludes with industry applications in regulated environments where reproducibility, auditability, and collaboration are essential.

Tags: *Git, GitHub, Data Version Control (DVC), MLflow, Reproducible Research, Data Science Workflows, Machine Learning Workflows, Collaboration, MLOps, Data Engineering, Experiment Tracking, Clinical Trials, Pharmaceutical Research, Reproducible Analytics*

INTRODUCTION

Modern data science, analytics, and machine learning projects involve far more than writing a single script or building a single model. Projects evolve over time, datasets change, models are retrained, new features are added, and multiple developers often work on the same codebase simultaneously. Without a structured workflow for managing code, data, and experiments, projects quickly become difficult to reproduce, maintain, and scale. Files get renamed, older versions are lost, changes are overwritten, and it becomes unclear which version of a dataset or model produced a particular result. These problems are extremely common in analytics and machine learning environments where version control practices are not established early.

Version control systems were originally developed for software engineering, but they are now equally important in data science, statistical analytics, and artificial intelligence development. Version control allows developers and analysts to track changes over time, maintain a history of a project, revert to previous versions when necessary, and collaborate without interfering with each other's work. However, modern analytics workflows require version control not only for code, but also for datasets, model artifacts, and experiment tracking. This leads to an ecosystem of tools that work together to manage the full lifecycle of analytics and machine learning projects.

This workflow ecosystem can be summarized as follows:

- **Git** tracks changes to code and project files
- **GitHub** provides remote repositories and collaboration tools
- **Data Version Control (DVC)** tracks datasets and large files outside of Git
- **MLflow** tracks machine learning experiments, models, and performance metrics

Together, these tools allow teams to build reproducible, collaborative, and scalable data science and machine learning workflows. The purpose of this paper is to describe how these tools work together and how they can be used to build structured and reproducible data science workflows that scale beyond individual projects and support team-based development environments.

1. Git and GitHub Fundamentals: The What, Why, and How

Before discussing advanced workflows, branching strategies, data versioning, and machine learning experiment tracking, it is important to understand the foundational tools that support modern development workflows: Git and GitHub. These tools form

the backbone of collaborative software development, analytics workflows, and machine learning project management. Git and GitHub serve different but complementary roles within a development environment.

Git is a version control system that tracks changes to files and directories over time, while GitHub is a platform that hosts remote repositories and enables collaboration between developers and teams. Together, they allow developers and analysts to maintain organized project histories, collaborate on code without overwriting each other's work, and manage development workflows across local machines and shared repositories in the cloud.

In modern development workflows, projects are typically stored in a shared remote repository hosted on a platform such as GitHub. Developers then clone that repository to their local machines, where they perform development work, commit changes locally, and push those changes back to the shared repository. This workflow allows multiple developers to work independently while contributing to the same project in an organized and controlled manner. Understanding how Git manages local repositories and how GitHub manages remote repositories is essential for building structured and collaborative development workflows.

1.1 Git: Distributed Version Control for Managing Code and Project History

Git is a distributed version control system designed to track changes to files and directories over a project's lifecycle. At its core, Git allows developers and analysts to take and save snapshots of a project at specific points in time, called *commits*, and maintain a complete history of a project's contributors and evolution. Each commit represents a saved state of the project and includes information about what changed, when it changed, and who made the change. This commit history allows developers to revert to previous versions of a project, compare changes between versions, and understand how a project developed over time.

One of the most important characteristics of Git lies in its distributed framework. Every developer working on a project has a full copy of the repository, called a *cloned repository*, and its history stored locally on their machine. This enables developers to work independently in parallel environments, create branches, test new features, and commit changes locally without affecting other developers' workflows or requiring constant access to a central server. Changes are only shared with others when they are *tracked*, *committed*, and *pushed* to a centralized remote repository. This distributed structure makes Git flexible, efficient, and resilient, since the repository history exists on multiple machines rather than only on a single central location.

In a typical workflow, a project begins with a remote repository hosted on a platform such as GitHub. Developers *clone* that repository to their local machines, which creates a local repository containing the project files and commit history. Development work is performed locally, where developers modify files, add those changes to version tracking, and create *commits* that represent meaningful updates to the project. These commits are stored in the local repository until they are pushed to the remote repository, where they become available to other developers working on the project.

Git also supports *branching*, which allows developers to create independent lines of development within the same project. Instead of all developers modifying the same files at the same time, developers can create branches to develop new features, fix bugs, or experiment with new ideas without affecting the main version of the project.

Overall, **Git serves as the version control engine that tracks codebase history, manages changes, supports branching workflows, and allows developers to maintain organized and reproducible development environments**. However, Git alone does not provide collaboration tools or shared repositories accessible by teams. This is where platforms such as GitHub come into the workflow.

1.2 GitHub: Remote Repositories and Collaborative Development Platforms

While Git serves as the version control engine that tracks changes and manages project history locally, GitHub provides a platform for hosting repositories remotely and enabling collaboration between developers and teams. GitHub is a cloud-based repository hosting platform built around Git that allows projects to be stored, shared, reviewed, and managed in a centralized environment accessible to multiple contributors. In modern development workflows, GitHub often serves as the central repository where the primary version of a project is stored and maintained.

One of the primary purposes of GitHub is to provide a *shared remote repository* that multiple developers can access and contribute to. Instead of passing project folders between developers or maintaining separate copies of a project on different machines, GitHub allows a team to maintain a single remote repository that acts as the central version of the project. Developers clone this repository to their local machines, perform development work locally, and then push their changes back to the remote repository.

Other developers can then pull those changes to update their own local repositories. This shared repository workflow allows teams to collaborate efficiently while maintaining a consistent and organized project structure.

GitHub also provides collaboration tools that extend beyond basic version control, most notably the pull request workflow. Pull requests allow developers to propose changes before they are merged into the main project branches, creating an environment where team members can review code, discuss modifications, identify potential issues, and approve changes before integration. This review process improves code quality, reduces the likelihood of introducing errors into the main codebase, and supports structured development workflows in team environments.

In addition to pull requests, GitHub provides tools for issue tracking, project management, access control, and workflow automation. Teams can manage tasks and bugs through issues, organize development progress using project boards, control repository permissions, and integrate automated testing and deployment workflows. GitHub also plays a major role in open-source software development, where developers around the world collaborate on shared projects. Overall, **GitHub serves as the collaboration and remote repository platform that complements Git**. While Git manages version control locally through commits, branches, and merges, GitHub provides the shared remote repository and collaboration environment that allows teams to work together on the same project in an organized and controlled manner.

1.3 Integrating Git and GitHub: Local and Remote Repository Workflows

Git and GitHub are often used together as part of a single development workflow, but they serve different roles within that workflow. Git manages version control locally on a developer's machine, while GitHub hosts the remote repository that acts as the shared version of the project. The interaction between local repositories and remote repositories is what enables collaborative development and structured version control workflows across teams.

In a typical Git and GitHub workflow, a project exists in two locations: a remote repository hosted on GitHub and a local repository on a developer's machine. The remote repository serves as the central version of the project, while developers clone that repository to create local repositories where development work is performed. Developers modify files locally and create commits in their local repository, and those commits are only shared with others when they are pushed to the remote repository. Because other developers may also be pushing changes to the remote repository, it is important to regularly pull updates to keep the local repository synchronized with the most recent version of the shared repository. **Committing locally, pushing to the remote repository, and pulling updates from the remote repository allows multiple developers to effectively work on the same project while maintaining a consistent and organized codebase**.

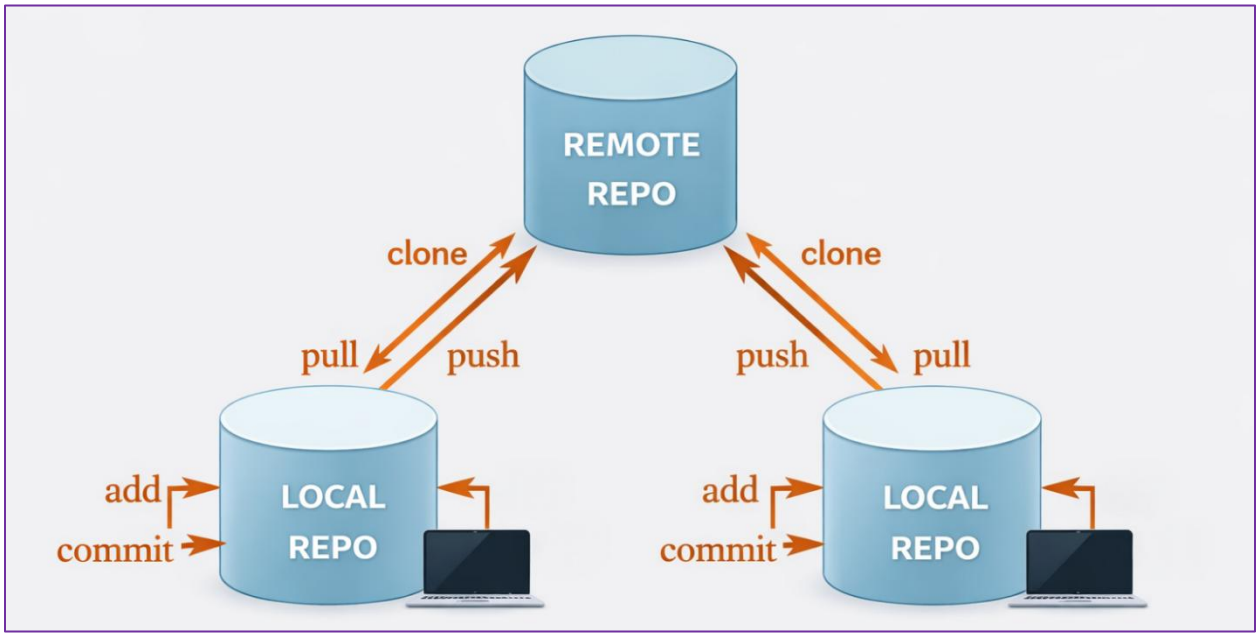

Figure 1. *Local and remote repository workflow in Git and GitHub. Developers clone a remote repository to create local repositories, commit changes locally, push commits to the remote repository, and pull updates to keep local repositories synchronized with the shared codebase. Credit: Ryan Paul Lafler, Premier Analytics Consulting, LLC.*

This relationship between local repositories and a shared remote repository is illustrated in **Figure 1**, where developers clone the remote repository, commit changes locally, and synchronize their work through push and pull operations.

This push and pull workflow allows multiple developers to work on the same project without overwriting each other's work. Developers can work independently on their local machines, commit changes locally, and then synchronize their work with the shared remote repository when ready. In team environments, changes are pushed to feature branches and then merged into main or develop branches through *pull requests*, which allow team members to review changes before integration into the primary codebase. This process maintains code quality and reduces the potential for messy merge conflicts.

Together, Git and GitHub create a development workflow where work is performed locally, codebase version history is tracked through Git, and collaboration occurs through a shared remote repository hosted on GitHub.

2. Data Version Control (DVC): Versioning Data in Analytics Workflows

Git is designed to track changes to code and text-based files, but it is not well suited for large binary files such as datasets, imagery, model artifacts, or large geospatial files. When large files are stored directly in a Git repository, the repository grows quickly because Git stores full versions of binary files rather than incremental differences. This bloats the repository, causing slow performance and issues pushing changes to remote repositories creating inefficient workflows in data science and machine learning projects where datasets and model outputs change frequently.

Data Version Control (DVC) addresses this limitation by providing an open-source versioning system tracking datasets and large files while keeping those files outside of the Git repository. Instead of storing the data itself in Git, DVC stores data in external storage such as cloud object storage, network storage, or local storage, and then tracks metadata and file hashes inside the Git repository. This allows Git to continue managing the project structure and code history while DVC manages datasets, model artifacts, and pipeline outputs. As a result, projects can version both code and data without making the Git repository large and difficult to manage.

DVC helps manage, track, and version changing datasets, intermediate outputs generated during processing and modeling, and trained models, particularly for regulated industries requiring reproducible workflows. By versioning datasets and model outputs, DVC helps ensure that analyses can be reproduced using the *exact data and code versions* that were used originally.

2.1 How Data Version Control (DVC) Integrates with Git Repositories

DVC is designed to work alongside Git repositories where Git continues to track code, documentation, and project structure, while DVC tracks datasets and large files stored outside the repository. When a dataset is added to a project and tracked using DVC, the data itself is stored in external storage, and a small metadata file that references the dataset version is added to the Git repository. This allows Git to track the metadata and project structure, with DVC tracking changes to the actual data files.

When developers clone a repository, they receive the code and DVC metadata files through Git. They can then use DVC to pull the associated dataset versions from external storage. This ensures that the correct dataset version is paired with the correct version of the code. Similarly, when data is updated, DVC tracks the new version of the data, and Git tracks the updated metadata files that reference the new dataset version. This workflow allows code and data to be versioned together while stored separately, which is essential for reproducible analytics and machine learning workflows.

By integrating with Git in this way, DVC extends version control beyond code and into the full data science workflow, including datasets, intermediate processing outputs, and trained model artifacts. This allows teams to maintain reproducible workflows where code, data, and models are all versioned and synchronized across different stages of development.

3. MLflow: Tracking Machine Learning Experiments and MLOps Lifecycles

While Git and DVC manage version control for code and datasets, machine learning workflows also require tracking experiments, model parameters, performance metrics, and trained models. Machine learning development is inherently iterative, with many experiments run using different datasets, features, model types, and hyperparameters. Without a structured system for tracking experiments, it becomes difficult to determine which models performed best, what parameters were used, or how a particular model was generated.

MLflow provides an open-source framework for tracking machine learning experiments and managing model optimization and fine-tuning workflows. It allows AI/ML engineers to log experiment runs, including parameters, metrics, and model artifacts, and compare results across multiple experiments. MLflow also includes a model registry that allows teams to manage model versions and track which models are deployed to production environments. In structured analytics and machine learning workflows, MLflow helps organize experimentation and model development so that model training, evaluation, and deployment can be tracked and reproduced over time.

4. Healthy Development Workflows with Branching in Git

Version control alone does not create an organized development environment. Without a structured workflow, developers may commit directly to the main branch, overwriting the stable codebase with untested code, interfere with other developers' works, and introduce difficult-to-resolve merge conflicts. **Structured branching workflows provide a framework for organized development, stable production code, and parallel feature development**.

Branching in Git allows developers to create independent lines of development within the same repository. Instead of all developers working on the same branch at the same time, development work is separated into branches for new features, bug fixes, releases, and experiments. These branches allow development to occur in parallel while protecting the stability of the main codebase. Once development on a branch is completed and tested, it can be merged back into the main project through a controlled process.

A healthy development workflow typically organizes branches based on their role in the development lifecycle. While workflows may vary between organizations, most structured Git workflows include the following branch types:

- **`Main` branch** – stable production code
- **`Develop` branch** – integration branch for completed features
- **`Feature` branches** – development of new features or enhancements
- **`Release` branches** – preparation for deployment and testing
- **`Hotfix` branches** – urgent fixes to production code

A typical healthy branching workflow showing how **`feature`**, **`release`**, and **`hotfix`** branches interact with the **`develop`** and **`main`** branches is illustrated in **Figure 2**.

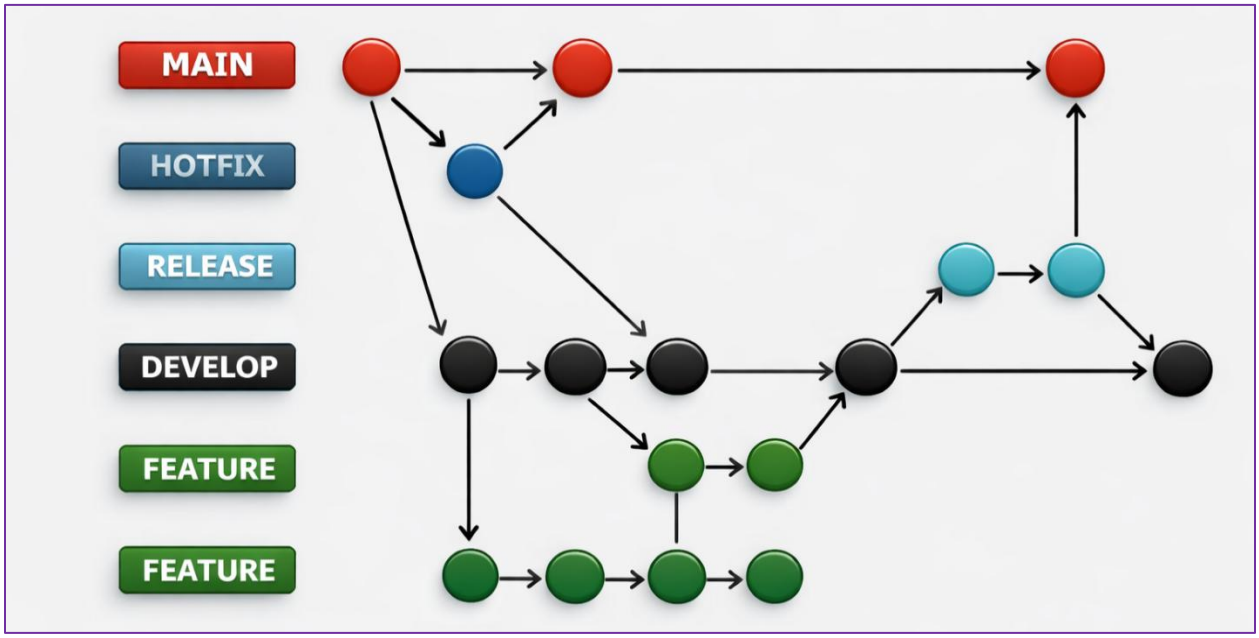

Figure 2. *Structured Git branching workflow showing* ***`main`***, ***`develop`***, ***`feature`***, ***`release`***, *and* ***`hotfix`*** *branches.* ***`Feature`*** *branches are created from the* ***`develop`*** *branch,* ***`release`*** *branches prepare code for deployment, and* ***`hotfix`*** *branches address production issues from the main branch before being merged back into both* ***`main`*** *and* ***`develop`****. Credit: Ryan Paul Lafler, Premier Analytics Consulting, LLC.*

This branching structure allows teams to support parallel development, maintain stable production code, test new features before release, and manage bug fixes without interrupting ongoing development. It also reduces potential merge conflicts and provides a clear structure for how code moves from development to testing to production.

4.1 `Main` Branch: Stable Codebase

The **`main`** branch represents the stable, production-ready version of the codebase. Often, this is referred to as the *primary codebase*. This branch should contain code that is tested, reviewed, and considered safe for deployment or release. In structured workflows, developers typically do not commit directly to the **`main`** branch. Instead, changes are introduced through pull requests after being developed and tested in other branches. This helps ensure that the **`main`** branch always represents a stable version of the project.

Maintaining a stable **`main`** branch is important because it provides a reliable version of the codebase that can be deployed, shared, or used as a reference point. If issues arise in development branches, the **`main`** branch remains unaffected, allowing the team to continue working without disrupting production code.

4.2 `Develop` Branch: Integration and Ongoing Development

The **`develop`** branch is commonly used as the primary integration branch where completed **`feature`** branches are merged before being promoted to the main branch. While the **`main`** branch represents stable production code, the **`develop`** branch represents the most recent version of ongoing development. New features are typically merged into the **`develop`** branch after they are completed and reviewed, allowing multiple features to be integrated and tested together.

Using a **`develop`** branch helps organize the development workflow by separating the primary codebase from active development. This allows teams to continue developing new features while maintaining a stable production version in the **`main`** branch. Once the **`develop`** branch reaches a stable state, it can be merged into the **`main`** branch as part of a release.

4.3 `Feature` Branches: Parallel Development and New Features

`Feature` branches are used to develop new functionality, enhancements, or experimental changes without affecting the main or develop branches. Each new feature or task is typically developed in its own branch, which is created from the **`develop`** branch. Developers can work independently on **`feature`** branches, commit changes locally, and push updates without interfering with other development work.

Once a feature is completed and tested, the feature branch is merged back into the **`develop`** branch through a pull request. This allows team members to review the changes before integration. **`Feature`** branches support parallel development by allowing multiple developers to work on different features at the same time while keeping the main development branches organized and stable.

4.4 `Release` Branches: Preparing for Deployment

`Release` branches are used to prepare a version of the project for deployment or production release. When the **`develop`** branch reaches a stable state and a new release is planned, a **`release`** branch is created to finalize testing, documentation updates, and minor fixes. This allows development of new features to continue in the **`develop`** branch while the **`release`** branch is stabilized for deployment.

Once testing and final adjustments are complete, the **`release`** branch is merged into the **`main`** branch and typically also merged back into the **`develop`** branch to ensure both branches contain the final release changes. **`Release`** branches help separate deployment preparation from ongoing feature development, which improves workflow organization and release stability.

4.5 `Hotfix` Branches: Quickly Fixing Production Issues

`Hotfix` branches are used to quickly address critical issues found in the production version of the code. These branches are typically created directly from the **`main`** branch so that fixes can be applied to the production code without waiting for ongoing development work in the **`develop`** branch to be completed.

After the issue is fixed and tested, the **`hotfix`** branch is merged into both the **`main`** branch and the **`develop`** branch. Merging into the **`develop`** branch ensures that the fix is not lost in future development work. **`Hotfix`** branches allow teams to respond quickly to production issues while maintaining an organized development workflow.

5. Practical Tips & Tricks for Effective Git and GitHub Workflows

Beyond understanding Git commands and branching strategies, effective Git usage comes from developing consistent workflow habits. Many Git problems are not caused by Git itself, but by poor workflow practices such as committing too much at once, not pulling before pushing, working directly on the main branch, or creating unclear commit histories. The following workflow tips help maintain organized repositories, clean commit histories, and smoother collaboration in team environments.

5.1 Tip #1 – Manage Features, Releases, Hotfixes, and Bug Fixes Using Branches and Sub-Branches

One of the most important Git workflow practices is organizing development work using branches and sub-branches. Instead of making changes directly on the **`main`** or **`develop`** branches, new work should be organized into **`feature`**, **`bugfix`**, **`hotfix`**, or **`release`** branches. For larger features or complex updates, sub-branches can be created to focus on specific components or functions within a feature. This approach keeps development organized, reduces merge conflicts, and protects the primary codebase from unstable changes.

Using branches and sub-branches allows teams to divide work into manageable components, support parallel development, and maintain a clean and structured development workflow. It also makes it easier to merge related work together and track the development history of specific features or fixes.

Key ideas for this workflow include:

- Create **`feature`**, **`bugfix`**, **`hotfix`**, or **`release`** branches from the **`develop`** branch
- Use sub-branches for specific components or functions (e.g., login-page, ML-model, db-fix)
- Organize work into manageable units to reduce merge conflicts
- Merge sub-branches into parent branches before merging into develop
- Branches help protect the main codebase from unstable development work
- Supports parallel development and structured project organization

To demonstrate this workflow, consider a scenario where a login page is not scaling correctly across different devices. The fix should be implemented without affecting ongoing development work, so a **`bugfix`** branch is created from the **`develop`** branch. The developer then modifies the relevant CSS and JavaScript files, commits the changes to the **`bugfix`** branch, and finally merges the **`bugfix`** branch back into the **`develop`** branch once the issue is resolved.

The workflow for this example includes creating a new branch from **`develop`**, tracking changes to the affected files, committing changes locally, switching back to the develop branch, and merging the **`bugfix`** branch into **`develop`**.

Workflow Steps:

- Create a new branch from the **`develop`** branch
- Modify relevant CSS and JavaScript files
- Add files to tracking
- Commit changes to the local **`bugfix`** branch
- Switch back to the **`develop`** branch
- Merge the **`bugfix`** branch into the **`develop`** branch

Example 1. Fixing a Login Page Scaling Issue Using a Bugfix Branch

```
git checkout -b bugfix/login-scaling develop
git add login.css login.html
git commit -m "WIP: adjust flexbox layout for better scaling"
git add login.css
git commit -m "WIP: refine media queries for mobile viewing"
git log --oneline
```

Example 1 demonstrates creating a bugfix branch from the develop branch, making changes and committing them on the bugfix branch, and then merging the bugfix branch back into the develop branch. This workflow keeps development organized, protects

the main branches from unstable changes, and maintains a structured project history that clearly shows when features and fixes were introduced into the project.

5.2 Tip #2 – Commit with Purpose Using Meaningful, Prefixed Messages

One of the most important Git habits is committing with purpose. Commits should not be random snapshots of work, and they should not be so large that multiple unrelated changes are grouped together. Each commit should represent a single meaningful change to the project. At the same time, developers should not wait until an entire feature is complete to commit changes. Instead, commits should be made locally as meaningful progress is achieved on a feature branch.

Commits should never be made directly on the main, develop, or release branches. All development work should occur on **`feature`** or **`bugfix`** branches, and commits should remain local to that branch until the feature or fix is ready to be merged into a shared branch. Keeping commits atomic and purpose-driven makes commit history easier to read, easier to debug, and easier to manage when rebasing or squashing commits later.

A useful practice is to use simple prefixes in commit messages to indicate the type of change being made. This creates a structured and readable commit history.

Common commit message prefixes include:

- **WIP**: Work in progress
- **feat**: New feature
- **fix**: Bug fix
- **hotfix**: Immediate production fix
- **refactor**: Code restructuring or optimization

Using prefixes makes commit history easier to search, review, squash, and reorganize when cleaning up commit history before merging branches.

To demonstrate this workflow, consider a scenario where a responsive scaling issue is discovered on a login page. The issue needs to be fixed without affecting other parts of the project, so a **`bugfix`** branch is created from the **`develop`** branch. The developer then makes a series of meaningful commits while working through the fix, using WIP and fix commit messages to document progress and changes. After the work is completed, the commit history can be reviewed to clearly see the sequence of changes made to resolve the issue.

The workflow for this example includes creating a **`bugfix`** branch, tracking changes to the affected files, committing meaningful updates with structured messages, and reviewing the commit history before merging the branch.

Example 2. Creating Meaningful Commits on a Bugfix Branch

```
git checkout -b bugfix/login-scaling develop
git add login.css login.html
git commit -m "WIP: adjust flexbox layout for better scaling"
git add login.css
git commit -m "WIP: refine media queries for mobile viewing"
git log --oneline
```

Example 2 demonstrates creating a **`bugfix`** branch from the **`develop`** branch, making meaningful WIP commits while implementing the fix, and reviewing the commit history. By committing locally with purposeful messages, the commit history clearly reflects the development process and makes future debugging and project maintenance much easier.

5.3 Tip #3 – Squash Multiple WIP Commits Before Pushing to the Remote Repository to Maintain a Clean Version History

When developing features or fixing bugs, it is common to create multiple WIP (work in progress) commits while implementing and testing changes. These commits are useful locally because they allow developers to track progress and save incremental work.

However, WIP commits should typically not be pushed directly to the remote repository because they can clutter the commit history and make the project history difficult to read and manage.

Instead, multiple WIP commits should be squashed into a single meaningful commit before pushing changes to the remote repository. This ensures that the commit history on the remote repository reflects completed, functional changes rather than intermediate development steps. Maintaining a clean and readable commit history is especially important in collaborative environments where multiple developers rely on commit history to understand project changes.

Key workflow ideas for squashing commits include:

- WIP commits are useful for tracking local progress
- WIP commits should typically not be pushed to the remote repository
- Multiple WIP commits can be squashed into a single commit
- Squashing creates a cleaner and more readable commit history
- Each commit pushed to the remote repository should represent a completed change
- Use interactive rebase to squash commits together.

To demonstrate this workflow, consider a scenario where multiple WIP commits were created while fixing a responsiveness issue on a login page. Before pushing the branch to the remote repository, the developer wants to combine these WIP commits into a single commit representing the completed fix. This is done using interactive **`rebase`** to **`squash`** the commits together, followed by safely pushing the updated commit history to the remote repository.

The workflow for this example picks up from Tip #2, retrieving recent commits, interactively rebasing and squashing commits, editing the final commit message, and pushing the cleaned commit history to the remote repository.

Workflow Steps:

- Retrieve the previous WIP commits
- Start an interactive rebase
- Squash the commits together
- Edit the new commit message
- Save and exit the rebase editor
- Safely push the rewritten commit history to the remote repository

Example 3. Squashing Multiple WIP Commits into a Single Commit

```
git rebase -i HEAD~2
```

Interactive rebase will open a commit history editor showing the most recent commits:

```
pick abc1234 WIP: refine media queries for mobile viewing
pick def5678 WIP: adjust flexbox layout for better scaling
```

Change the second commit from **`pick`** to **`squash`**:

```
pick abc1234 WIP: refine media queries for mobile viewing
squash def5678 WIP: adjust flexbox layout for better scaling
```

After saving and closing the editor, Git will prompt for a new commit message representing the combined changes:

```
fix: corrected issue with responsiveness on mobile devices
```

Finally, safely push the updated commit history to the remote repository:

```
git push --force-with-lease origin bugfix/login-scaling
```

Example #3 demonstrates squashing two WIP commits into a single meaningful commit before pushing changes to the remote repository. Using interactive rebase allows developers to maintain a clean, organized commit history where each commit represents a completed change. This improves repository readability, simplifies project history, and supports better collaboration in team development environments.

5.4 Tip #4 – (Alternative) Multiple WIPs? Stash Work in Progress Locally and Only Commit Final Changes

An alternative to creating multiple WIP commits and later squashing them is to use Git **`stash`** to temporarily store work in progress on the local repository without committing those changes to version history. Stashing allows developers to save incomplete work, switch branches, test other changes, or continue experimenting without affecting commit history. Once the work is complete and ready to be committed, the stashed changes can be reapplied and committed as a single meaningful commit.

This approach helps maintain a clean version history because only completed changes are committed and pushed to the remote repository. It also avoids the need to squash multiple WIP commits later. Stashing is particularly useful when experimenting with code, testing different implementations, or temporarily switching tasks while working on a feature or bug fix.

Key workflow ideas for using Git **`stash`** include:

- Save work in progress locally without committing to version history
- Avoid cluttering commit history with multiple WIP commits
- Experiment and test changes without affecting commit history
- Apply stashed changes later and commit a final completed change
- Stashes can be listed, applied, or deleted
- Supports clean commit history and organized development workflows.

To demonstrate this workflow, consider the same scenario where a login page is not scaling correctly across different devices. Instead of creating multiple WIP commits while testing layout changes, the developer temporarily stores work using Git **`stash`**. After experimenting and completing the fix, the stashed changes are applied back to the working directory and committed as a single meaningful commit before pushing the branch to the remote repository.

The workflow for this example includes creating a bugfix branch, modifying files, stashing work in progress, reviewing stashed changes, applying the stashed work, committing the final changes, and pushing the completed fix to the remote repository.

Workflow Steps:

- Create a bugfix branch from the develop branch
- Modify relevant CSS and HTML files
- Stash work in progress instead of committing WIP changes
- Continue working and create additional stashes if needed then view stash history
- Apply stashed changes back to the working directory
- Commit final completed changes
- Push final commit to the remote repository

Example 4. Creating Meaningful Commits on a Bugfix Branch

```
git checkout -b bugfix/login-scaling develop

git stash push -m "WIP: adjusting flexbox layout for scaling"

git stash push -m "WIP: refine media queries for mobile"

git stash list

git stash pop

git add login.css login.html

git commit -m "fix: resolved login page scaling across devices"

git push origin bugfix/login-scaling
```

Example 4 demonstrates using Git **`stash`** to temporarily store work in progress instead of creating multiple WIP commits. Stashing allows developers to experiment and make incremental changes locally without affecting commit history. Once the work is complete, the stashed changes can be applied, committed as a single meaningful commit, and pushed to the remote repository. This approach keeps commit history clean and reduces the need to squash commits before pushing changes.

5.5 Tip #5 – Git Merge vs. Git Rebase: Preserving History vs. Rewriting History

One of the most important concepts in Git workflows is understanding the difference between merging and rebasing branches. Both merge and rebase are used to integrate changes from one branch into another, but they handle commit history differently. Git merge preserves the full commit history of both branches by creating a merge commit, while Git rebase rewrites commit history by placing commits from one branch on top of another branch to create a linear history.

Choosing between **`git merge`** and **`git rebase`** depends on the workflow and whether the branch is shared with other developers. In general, rebasing is useful for cleaning up local feature branch history before merging, while merging is preferred for integrating branches into shared repositories since it preserves commit history and avoids rewriting history that other developers may rely on.

Key workflow ideas for merge and rebase include:

- **`git merge`** preserves commit history and creates a merge commit (leads to non-linear histories)
- **`git rebase`** rewrites commit history to create a linear history
- Rebase is useful for cleaning up local feature branches before merging
- Merge is preferred for shared branches such as develop or main
- Rebasing shared branches can cause conflicts for other developers
- Use rebase for local/private branches and merge for shared/public branches
- Merge preserves history; rebase creates a cleaner, linear history

When do you use Git Rebase vs. Git Merge? A common workflow is to rebase local feature branches onto the latest version of the develop branch before merging the feature branch into develop. This ensures that the feature branch includes the most recent updates and maintains a clean commit history. After the feature branch is rebased and tested, it can be merged into the develop branch using a merge commit to preserve the project history.

A useful rule of thumb is:

- Use rebase for local feature branches
- Use merge for shared branches

To demonstrate this workflow, consider the same login scaling bugfix branch. The developer first updates the local feature branch by rebasing it onto the latest version of the develop branch. This ensures the feature branch includes recent changes from develop. After rebasing and testing, the feature branch is merged into the develop branch, and the updated develop branch is pushed to the remote repository.

Workflow Steps:

- Switch to local feature branch
- Fetch latest updates from remote repository
- Rebase feature branch onto develop branch
- Switch to develop branch
- Merge feature branch into develop branch
- Push updated develop branch to remote repository

Example 5. Rebase Local Feature Branch and Then Merge into Develop Branch

```
git checkout bugfix/login-scaling

git fetch origin

git rebase develop
```

This rebases the local bugfix branch on top of the most recent version of the **`develop`** branch so that the **`feature`** branch includes the latest changes before merging.

```
git checkout develop

git merge bugfix/login-scaling

git push origin develop
```

This switches to the **`develop`** branch, merges the **`bugfix`** branch into develop to incorporate the completed fix, and then pushes the updated develop branch to the remote repository so the changes are shared with the team.

Example 5 demonstrates rebasing a local feature branch onto the latest **`develop`** branch to incorporate recent changes and maintain a linear commit history, followed by merging the feature branch into the **`develop`** branch to preserve project history. Using **`git rebase`** for local branches and merge for shared branches provides a balanced workflow that maintains both clean commit history and complete project history.

6. Tracking Data Histories with Data Version Control (DVC)

While Git and GitHub provide structured version control for code and collaborative development workflows, many data science and machine learning projects—especially in life sciences, healthcare, finance, environmental science, and other regulated industries—require reproducibility not only of code, but also of the datasets used for analysis and model development. In regulated environments, it is often necessary to reproduce analytical results months or years later, demonstrate exactly which dataset was used for a model or statistical analysis, and maintain a complete audit trail of data changes over time.

In many analytical workflows, datasets are updated periodically, cleaned, transformed, or augmented with new samples. Without structured data version control, teams often end up with multiple copies of datasets stored with inconsistent naming conventions, making it difficult to determine which dataset version was used for a specific analysis, model, or report. This creates reproducibility challenges and can be problematic in regulated environments where traceability and auditability are required.

Data Version Control (DVC) extends Git-based version control workflows to datasets, large files, and machine learning artifacts while integrating directly with Git repositories. Instead of storing large datasets directly inside Git, DVC stores metadata in the Git repository while the actual data is stored in external storage systems such as local storage, network storage, cloud storage, or object storage. This allows entire analytical workflows including code, datasets, and model outputs to be version controlled together.

This type of workflow is particularly important in clinical trials, pharmaceutical research, statistical analysis planning, and healthcare analytics where datasets change over time and analyses must be reproducible and auditable. Clinical trial data often undergo multiple data cuts, cleaning processes, and database locks, and statistical analyses must be tied to specific dataset versions defined in a Statistical Analysis Plan (SAP). Being able to track which dataset version was used for a particular analysis or model is critical for reproducibility, validation, and regulatory review. Using Data Version Control alongside Git allows teams to track dataset versions, analysis code, and model outputs together, ensuring analytical workflows can be reproduced and audited when necessary.

6.1 The Problem with Data Tracking and Management

Version control works extremely well for code and text files, but data science and machine learning projects involve datasets that are often large, frequently updated, and stored in multiple locations. Over time, datasets change, new samples are added, features engineered, missing values imputed, and models retrained using different versions of the data. Without a structured way to track dataset versions, it's difficult to determine which dataset produced what specific model or analysis result.

What happens without data version control? Teams often end up with multiple versions of the same dataset stored in different folders with names such as **`dataset_final.csv`**, **`dataset_v2.csv`**, or **`dataset_final_final.csv`**. It may become unclear which dataset version was used to train a machine learning model or generate a report. Large datasets may also be too large to store directly in Git repositories, email, or shared folders. These issues often lead to reproducibility problems, where code runs correctly on one machine but produces different results on another because a different dataset version was used.

Data Version Control (DVC) addresses these problems by extending tracking, versioning, and reproducibility to datasets and large files while integrating directly with Git-based workflows.

6.2 What is Data Version Control and How Does it Extend Git?

Data Version Control (DVC) is an open-source tool that allows developers and data scientists to track datasets, large files, and machine learning artifacts in a way that integrates with Git repositories. Instead of storing large data files directly inside Git, DVC stores metadata in the Git repository while the actual data is stored in external storage such as local storage, network storage, cloud storage, or object storage systems.

DVC tracks datasets using hash-based identifiers, similar to how Git tracks commits using hashes. When a dataset changes, DVC detects the changes, stores the new version in a data storage location, and updates the metadata files tracked by Git. This allows Git to manage the project structure and metadata while DVC manages the actual large files and dataset versions.

DVC adopts Git commands such as **`add`**, **`push`**, **`pull`**, **`checkout`**, and **`diff`** making it similar for users already familiar with Git workflows. As described, DVC is not a replacement for Git; rather, it extends Git so that *entire data science projects*, including code, datasets, and model artifacts, can be version controlled for reproducible submissions and workflows.

Using Git and DVC together allows teams to track:

- Code changes with Git
- Dataset versions with DVC
- Model artifacts and outputs with DVC
- Project history and metadata with Git
- Data storage in local, network, or cloud storage systems

This combination creates a reproducible and structured data science workflow where both code and data histories are tracked together.

6.3 Versioning a Housing Price Dataset

To demonstrate how DVC tracks dataset versions, consider a simple machine learning project that uses a housing price dataset stored in a CSV file. The dataset will be added to DVC tracking, then new rows will be appended to the dataset, and DVC will track the changes between dataset versions. This example demonstrates how Git and DVC work together to track both project history and dataset history over time.

The workflow for this example includes initializing a Git and DVC repository, creating a dataset, adding the dataset to DVC tracking, modifying the dataset by adding new rows, tracking the updated dataset version, and comparing differences between versions.

Workflow Steps:

- Initialize Git and DVC repository
- Create tabular CSV dataset
- Add CSV dataset to DVC tracking
- Modify dataset by adding new samples
- Track updated dataset version
- Compare differences between dataset versions
- Restore the previous dataset version

Example 6. Initializing, Tracking, Updating, and Viewing Differences Between Datasets

```
mkdir real_estate_project
cd real_estate_project


git init
dvc init
git commit -m "Initialized Git and DVC for real estate project"
mkdir data


echo "house_id,price,sqft,location" > data/house_prices.csv
echo "1,250000,1600,San Francisco" >> data/house_prices.csv
echo "2,320000,2000,New York" >> data/house_prices.csv
dvc add data/house_prices.csv
```

This workflow initializes a Git repository and a DVC repository, creates a dataset, and adds the dataset to DVC tracking. When the dataset is added, DVC creates a **`.dvc`** metadata file that is tracked by Git, while the actual dataset is stored in the DVC data storage location.

```
echo "10001,275000,1800,Los Angeles" >> data/house_prices.csv
echo "10002,500000,2500,Chicago" >> data/house_prices.csv


dvc add data/house_prices.csv
git add data/house_prices.csv.dvc .gitignore
git commit -m "Version 2: Added new housing data records"


dvc push
```

This updates the dataset, tracks the new dataset version with DVC, and stores the updated dataset version in the DVC storage location. Git tracks the metadata file representing the dataset version.

```
dvc diff
git log --oneline
```

The **`dvc diff`** command shows differences between dataset versions, such as rows added or removed, while **`git log`** shows the commit history associated with dataset version changes.

To restore a previous dataset version attached to that Git commit:

```
git checkout <commit_id>
dvc checkout
```

This restores the dataset to the version associated with the selected Git commit, allowing users to reproduce previous experiments or analyses.

Example 6 demonstrates how DVC tracks dataset versions over time while Git tracks project history and metadata. When a dataset is modified, DVC stores the new dataset version in data storage while Git tracks the metadata describing that dataset version. Commands such as **`dvc add`**, **`dvc push`**, **`dvc diff`**, **`and dvc checkout`** allow users to track dataset versions, compare dataset changes, and restore previous dataset versions. Git and DVC provide a reproducible workflow where both code and data histories are tracked and version controlled.

7. Discussion and Industry Applications

The open-source workflows described and presented in this paper extend beyond software development and are highly applicable to data science, machine learning, statistical analysis, and analytical modeling workflows across many industries. Version control for code alone is not sufficient in environments where datasets change over time, models are retrained, analyses are updated, and results must be reproducible months or years later. Combining Git for code version control, GitHub for collaboration, Data Version Control (DVC) for dataset versioning, and MLflow for experiment and model tracking creates a structured workflow that supports reproducibility, traceability, collaboration, and long-term project maintenance.

In many analytical workflows, results depend not only on the code used, but also on the exact dataset version, model parameters, and experiment configuration. Without version control for data and experiments, it becomes difficult to reproduce results, validate models, or explain differences between analytical outcomes over time. Using structured version control workflows allows teams to track changes to code, datasets, and models together, creating a complete project history and enabling reproducible analytical workflows.

These workflows are especially useful in industries where data is updated periodically, models are retrained, and analytical results must be documented and reproducible. The combination of Git, GitHub, DVC, and MLflow provides an open-source solution for managing the full lifecycle of data science and machine learning projects, from data collection and preprocessing to model development, evaluation, deployment, and long-term maintenance.

These industries include:

Clinical Trials and Pharmaceutical Research

- Track different data cuts and database locks used for statistical analysis
- Maintain reproducibility for Statistical Analysis Plans (SAPs)
- Version control analysis code, datasets, and model outputs together
- Maintain audit trails for regulatory review and submissions

Healthcare Analytics and Hospitalization Data

- Track updates to patient datasets and observational studies
- Version control risk models, readmission models, and outcome models
- Maintain data lineage for healthcare analytics and reporting

Environmental Science, Climate Science, and Geospatial Analytics

- Track large environmental datasets such as NetCDF, GRIB, or raster data
- Maintain reproducible workflows for research and policy analysis
- Manage large datasets stored in cloud or object storage systems

Manufacturing and Industrial Analytics

- Track sensor data and process monitoring datasets over time
- Reproduce analytical results used for process optimization
- Maintain traceability for quality audits and production analytics

Across these industries and applications, the common requirement is the ability to reproduce analytical results using the exact code, dataset version, and model configuration used at the time of analysis. Open-source tools such as Git, GitHub, Data Version Control, and MLflow provide a flexible and scalable framework for managing reproducible analytical workflows. By version controlling code, datasets, and experiments together, teams can build structured, collaborative, and reproducible data science workflows that support long-term project development, model maintenance, and regulatory or research requirements.

CONCLUSION

Modern data science, analytics, and machine learning projects require structured workflows that manage not only code, but also datasets, experiments, and model development over time. As projects evolve, datasets change, models are retrained, and analytical workflows become more complex. Without version control and structured development practices, projects become difficult to reproduce, maintain, and collaborate on. Git and GitHub provide the foundation for version controlling code and managing collaborative development, while structured branching strategies and commit workflows support organized development and clean project history.

Data Version Control extends version control workflows to datasets and large files, allowing teams to track dataset versions alongside code, while MLflow supports experiment tracking, model versioning, and performance tracking for machine learning workflows. Together, these tools create a complete workflow for managing the full lifecycle of data science and machine learning projects, from data collection and preprocessing to model development, evaluation, and deployment. By version controlling code, datasets, and experiments together, teams can reproduce analytical results, track changes over time, and maintain structured project histories.

These workflows are particularly valuable in research environments and regulated industries such as clinical trials, pharmaceutical research, healthcare analytics, environmental science, and finance, where reproducibility, traceability, and auditability are critical. The combination of Git, GitHub, Data Version Control, and MLflow provides a powerful open-source framework for building reproducible, collaborative, and scalable analytical workflows, making these tools essential components of modern data science and machine learning development environments.

REFERENCES

Barreto Simedo Pacheco, L., Rahman, M., & Rabbi, F. (2024). DVC in open source ML-development: The action and the reaction. *Proceedings of the IEEE/ACM International Conference on AI Engineering (CAIN 2024)*. https://dl.acm.org/doi/10.1145/3644815.3644965

DeWitt, P. E., et al. (2024). Open source and reproducible workflows for clinical data challenges and model evaluation. *Scientific Data, Nature Publishing Group*. https://www.nature.com/articles/s41597-023-02854-0

Li, Z., Kesselman, C., Nguyen, T. H., Xu, B. Y., Bolo, K., & Yu, K. (2025). From data to decision: Data-centric infrastructure for reproducible machine learning in collaborative eScience. *arXiv*. https://arxiv.org/abs/2506.16051

Schlegel, M., et al. (2025). Capturing end-to-end provenance for machine learning pipelines. *ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S0306437924001534

ACKNOWLEDGEMENTS

The author would like to acknowledge the PharmaSUG 2026 Conference Committee, including the Academic Chair, Operations Chair, and Section Chairs, for their acceptance of this submission and for the opportunity to present this work at PharmaSUG 2026.

AUTHOR'S BIO

Ryan Paul Lafler is the Founder and Lead Consultant of Premier Analytics Consulting, LLC, a California-certified small business based in San Diego specializing in applied AI and machine learning systems, distributed data engineering, statistical analysis, enterprise GIS, and custom full-stack analytics platform development. As a principal architect and consultant, he designs and delivers infrastructure-aware AI/ML solutions, production-grade analytics platforms, scalable data infrastructure, GIS and spatial analytics systems, and statistical modeling workflows for enterprise organizations, public-sector agencies, and research institutions. Through consulting and contracting roles, he has cross-industry programming expertise in Python, R, SQL/NoSQL, SAS®, and modern JavaScript frameworks, and implements structured quality control, validation, and governance practices for automated analytics and AI-assisted workflows. He also serves as an Adjunct Professor in the Big Data Analytics Graduate Program, the Department of Mathematics and Statistics, and the Global Campus Program at San Diego State University. He earned his Master of Science in Big Data Analytics (2023) following the defense and publication of his thesis, and his Bachelor of Science in Statistics with a Minor in Quantitative Economics (2020), both from San Diego State University.

CONTACT INFORMATION

Comments, suggestions, and/or any questions are encouraged and may be sent to:

Ryan Paul Lafler
President, CEO, and Lead Consultant
Premier Analytics Consulting, LLC
Email: rplafler@premier-analytics.com
Website: www.Premier-Analytics.com
LinkedIn: www.linkedin.com/in/RyanPaulLafler

COMPANY INFORMATION

Premier Analytics Consulting, LLC is a California Certified Small Business founded and led by Ryan Paul Lafler that provides cross-industry services including infrastructure-aware AI and machine learning systems, distributed data engineering, statistical analysis and modeling, enterprise GIS solutions, and custom full-stack platform development for organizations working with complex data environments. The firm partners with enterprise organizations, public-sector agencies, consulting firms, and research institutions through prime contracts, subcontracts, consulting and advisory engagements, and technical partnerships in flexible remote and hybrid environments. Premier Analytics Consulting focuses on designing and implementing reliable, secure, and production-ready AI, analytical, statistical, data engineering, and GIS systems that support open-source modernization and enterprise decision-making, research, operations, and long-term organizational data strategy across industries and technical domains.

➤ ***Learn more about our services and engagement structures:*** **www.Premier-Analytics.com**

TRADEMARK CITATIONS