Paper PP14

# From Git Init to GitHub Push: A Complete Version Control Workflow for Statistical Programming

Jeetender Chauhan, Merck & Co., Inc., Rahway, NJ, USA
Madhusudhan Ginnaram, Merck & Co., Inc., Rahway, NJ, USA
Sarad Nepal, Merck & Co., Inc., Rahway, NJ, USA
Benjamin Wang, Merck & Co., Inc., Rahway, NJ, USA

## 1. ABSTRACT

Have you ever made a mistake in your code and wished you could go back, or worried about a colleague overwriting your work? Git can solve these problems. It's more than a tool; it's a backup for your code and your team's best friend. It's a version control system that lets you travel back in time to any previous version of your work, compare changes, and work together smoothly. And when you're ready to share your work with the team, GitHub then takes this further by letting your entire team access and contribute to the same codebase.
This paper serves as a simple, practical guide for statistical programmers who use R, Python or SAS to build important programs. We'll explore how Git streamlines workflow, from backup to advanced features like branching and stashing, and ensuring all changes are tracked and recorded in version history for a clean, traceable record. By using Git, we can make our programming lives easier and ensure our projects are always organized, auditable, and ready for regulatory inspection. This isn't just about managing code; it's about confidently creating better, more reliable deliverables that stand up to the most thorough review.
**Keywords:** Git, GitHub, version control, statistical programming, reproducibility, clinical research

## 2. BACKGROUND

In statistical programming, especially in clinical research and pharmaceutical development, it's essential to track every change made to your code. This practice is not only a good habit-it's a regulatory requirement. The FDA's guidance on data integrity emphasizes the need for audit trails and version control in electronic systems used for clinical trials (U.S. Food and Drug Administration, 2018). Additionally, FDA regulations under 21 CFR Part 11 require that electronic records maintain complete audit trails (U.S. Food and Drug Administration, 2003). The NIH provides specific version control guidelines for research documentation and code management (National Institutes of Health, 2015).

Despite these requirements, many programmers still manage code versions manually—by renaming files like script_v1.R or final_final.R—or by storing them on shared drives. These methods can quickly become disorganized and error-prone, especially when multiple team members are involved. Problems like overwriting someone else's work or losing track of changes are common.
This is where modern version control tools become essential. Git addresses the problem of code history and local reproducibility by keeping a complete record of changes and allowing programmers to experiment safely with branching and merging. GitHub extends this capability into the cloud by providing a central, shared space where teams can collaborate, review one another's changes, and manage projects transparently. Together, they provide the infrastructure that supports reproducibility, accountability, and open science in programming (Huskey, 2024).
Increasingly, journals and institutions encourage or even require researchers to share their code and data in public repositories such as GitHub, underscoring its role in reproducibility and open science (De Sterck et al., 2023). By adopting Git and GitHub, statistical programmers not only streamline their own workflows but also ensure that deliverables remain well-documented, auditable, and ready for regulatory or peer review.

## 3. WHAT IS GIT AND GITHUB?

Git is a distributed version control system (VCS) created to track changes in code over time. It allows programmers to capture snapshots of their work (i.e., "commits"), explore new ideas on separate branches, and merge them back while maintaining a permanent, auditable history (Chacon & Straub, 2014). Each developer works with a full copy of the repository locally, which makes Git fast, reliable, and resilient.

GitHub, by contrast, is a cloud-based collaboration platform built on top of Git. It hosts repositories online, making them accessible to teams for sharing, reviewing, and coordinated development. GitHub adds features beyond basic version control: pull requests for peer review, issue tracking, access management, and integration with continuous testing and deployment (GitHub, Inc., 2025).
In simple terms:

- Git = the local tool that tracks and manages changes.
- GitHub = the online service that connects teams and organizes collaborations.

Together, they form the backbone of modern programming workflows—combining local reproducibility and history from Git with global collaboration and transparency from GitHub.

## 4. GETTING STARTED WITH GIT AND GITHUB

For programmers new to version control, the first step is to set up Git locally and connect it to GitHub. This ensures you can manage your code history on your computer while also collaborating through the cloud.

### 4.1 Create a GitHub Account

1. Visit https://github.com and sign up.
2. Choose the free plan (sufficient for most users).
3. Verify your email to activate the account.

### 4.2 Install Git

1. Download Git from https://git-scm.com for Windows, macOS, or Linux.
2. Use default settings during installation.

### 4.3 Configure Git

Open your terminal (or Git Bash on Windows) and enter:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

This information tags each commit with your identity.

### 4.4 Create a Repository

Once Git is installed, the next step is to create a working directory on your local computer. You can create repositories using the GitHub interface or via the command line.

#### *4.4.1 Example: Creating a gitpaper1 Repository*

**Command:**

```
$ mkdir gitpaper1
$ cd gitpaper1
```

After creating and moving into the repository folder, check if it is already initialized as a Git repository:

**Command:**

```
$ git status
```

**Message:**

```
fatal: not a git repository (or any of the parent directories): .git
```

The message “fatal: not a git repository” means that Git has not yet been initialized in this directory.
To initialize a new git repository to track the changes to your project, run:

**Command:**

```
$ git init
```

**Message:**

```
Initialized empty Git repository in C:/Users/<path to directory>/gitpaper1/.git/
```

Git creates a hidden .git folder inside your working directory. This folder contains all the metadata and history needed to track changes. Notice the prompt changes after initialization:

```
user@XYZ MINGW64 ~/<path to directory>/gitpaper (master)
```

The (master) at the end shows that you are now on the default branch of this new repository. In newer versions of Git, the default branch may be called main instead of master.

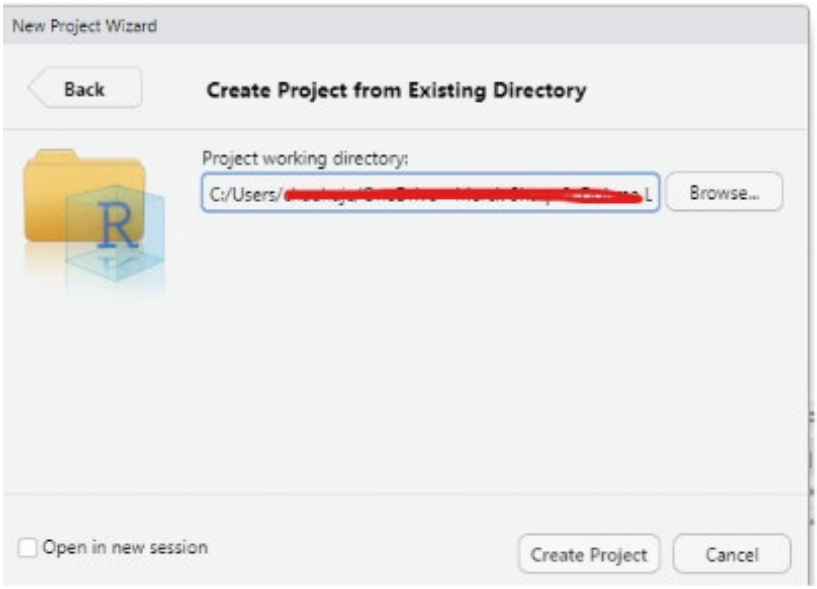
At this point, the directory is a Git repository, but it’s still empty—no files are being tracked yet. The next step is to add files and make your first commit. Before doing so, however, it is good practice to set up an R Project within the repository. This keeps the project clean, reproducible, and ready for version control.

## 5. INTEGRATING R PROJECTS WITH GIT

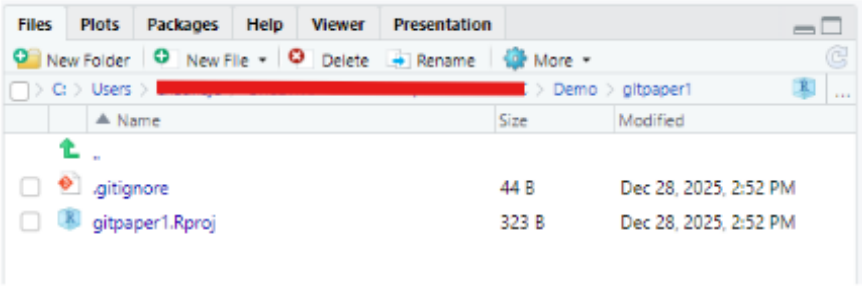
Once a Git repository has been initialized, the next step for R users is to link it with an RStudio Project. An R Project provides a structured environment for analysis by keeping scripts, data, outputs, and settings together. When combined with Git, this setup ensures that both the code and project configuration are version-controlled and reproducible.

The most straightforward way to create an R Project inside an existing Git repository is through the RStudio interface:

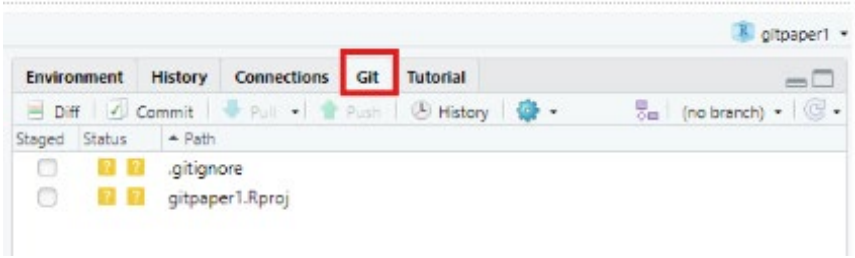
1. Open RStudio.
2. Go to File → New Project → Existing Directory.
3. Browse to your Git repository folder (e.g., gitpaper1).
4. Select the folder and click Create Project.




RStudio creates a .Rproj file in the repository root.




This file contains project metadata and tells RStudio to treat the folder as a project.
Because the .Rproj file is part of the repository, it can be tracked by Git and shared with collaborators.
Once the project is created, RStudio automatically recognizes Git integration.

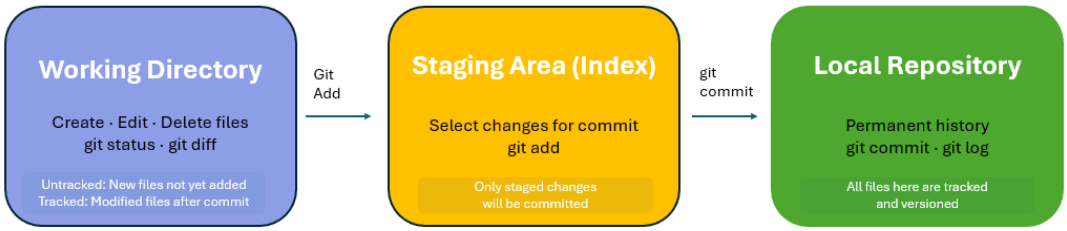



A Git tab appears in the interface, allowing users to stage, commit, and push changes without leaving the IDE (Integrated Development Environment). This workflow combines the organizational benefits of R Projects with the version control power of Git, making it ideal for collaborative and auditable programming in clinical research.

## 6. CORE GIT CONCEPTS AND TERMINOLOGY

Before working through practical examples, it is helpful to understand the core concepts that define how Git manages changes. This section focuses on Git itself—the version control system responsible for tracking changes and maintaining version history—rather than on GitHub or other hosting platforms. Git is built around a structured, three-stage workflow that gives programmers fine-grained control over what is recorded in version history. This design is particularly valuable in statistical and clinical programming, where not every intermediate edit should be permanently stored.

### 6.1 The Three Core Areas of Git: Working Directory, Staging Area, and Repository




In *Pro Git* (2nd ed.), Scott Chacon and Ben Straub explain that "Git has three primary areas—working directory, staging area, and repository—where files can exist in one of three states: committed, modified, or staged." They clarify what each state means and emphasize that "understanding the role of each area is essential to understanding how Git provides controlled and auditable version management" (Chacon & Straub, 2014).

#### *6.1.1 Working Directory*

The working directory is the local folder on your computer that contains the project files you actively work on—R scripts, SAS programs, documentation, and outputs. This is where you write new code, edit existing code, and delete or reorganize files. Changes made here reflect active development and are often incomplete or exploratory.

A key point is that files in the working directory are not automatically part of the version history. Git can detect that files have changed (for example, a script was edited, or a new file was created), but detection does not mean those changes are permanently recorded. At this stage, Git can help you review what has changed using commands such as git status (to summarize file states) and git diff (to show line-by-line edits that have not yet been staged). This is why the working directory is often described as the "workspace" where development happens before version control decisions are made.

#### *6.1.2 Staging Area (Index)*

The staging area—also called the index—is an intermediate holding area between the working directory and the repository history. Its purpose is to allow a developer to choose exactly what will be included in the next commit. This is one of Git's most important design features: it enables selective and intentional version control.

When you run git add, you are not committing changes yet, you are placing a copy of the selected file content (or selected changes) into the staging area as the candidate snapshot for the next commit. This allows a clean workflow where you can modify multiple files but commit only the subset that belongs together logically (for example, commit updates to lineplot.R while leaving unrelated edits in graph.R unstaged). Git's staging step supports clearer commit history and makes code review and audit trails easier because each commit can be kept focused and well documented.

#### *6.1.3 Local Repository (Commit History)*

The local repository is where Git stores the project's permanent history as a sequence of commits. Once changes are staged, the git commit command records a snapshot into the repository history along with metadata such as author, timestamp, and a message describing the purpose of the change.

Commits form an auditable timeline of development. Because each commit captures a specific state of the project, the repository enables key version-control capabilities such as:

- tracing when and why a change was introduced,
- comparing versions across time, and
- restoring previous states if needed.

Importantly, only staged content becomes part of a commit. Any changes that remain in the working directory (but were not staged) remain outside the version history until they are staged and committed in a later step. This separation is central to Git's controlled workflow and is why understanding these areas is foundational for reproducibility and traceability.

#### *6.1.4 Practical Summary for Readers*

- Working directory = where you edit and create files (not recorded yet).
- Staging area (index) = where you select what will go into the next snapshot.
- Local repository = where committed snapshots are stored as permanent history.

## 7. VERSION CONTROL WORKFLOW IN GIT

At this stage, the working environment has been fully set up: the project directory (gitpaper1) has been created, Git has been initialized, and an R Project has been configured. The next step in the workflow is to inspect the current state of the repository using the git status command.
Running the following command:

**Command:**

```
$ git status
```

produces message indicating that the repository is on the master branch, that no commits have been created yet, and that several files are currently untracked.

**Message:**

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        gitpaper1.Rproj
nothing added to commit but untracked files present (use "git add" to track)
```

This output introduces two fundamental Git concepts: branches and commits.
A branch represents an independent line of development within a Git repository. Every Git repository begins with a single default branch—master—on which development initially occurs. Additional branches can be created later to support parallel development, experimentation, or feature-specific work before changes are merged back into the main line of development. In this workflow, development initially proceeds on the default branch (UBC Library Research Commons, n.d.).

A commit is the mechanism by which Git permanently records changes in the repository. The git commit command captures a snapshot of the staged files and adds it to the project's version history. The git status output indicating "No commits yet" confirms that, although Git has been initialized, no changes have been staged or recorded so far. As a result, the repository currently contains no saved snapshots of work.

After initializing the repository and reviewing its initial state, two new R script files—graph1.r and lineplot1.r—were created in the project directory. Running the git status command again reflects this change in the working directory:

**Command:**

```
$ git status
```

**Message:**

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        gitpaper1.Rproj
        graph1.r
        lineplot1.r
nothing added to commit but untracked files present (use "git add" to track)
```

The output shows that the repository is still on the master branch and that no commits have been created yet. In addition to the previously listed configuration files, the newly created scripts graph1.r and lineplot1.r now appear under Untracked files. This indicates that Git has detected the presence of these files in the working directory but is not yet monitoring them for changes.

At this stage, all listed files exist only in the working directory and remain outside the project's version history. This behavior illustrates an important aspect of Git's version control process: files are not automatically tracked simply because they exist. Instead, programmers must explicitly decide which files should be included in version control.

### 7.1 Git Add (Staging Area)

With the project structure in place and the initial scripts created, the next step is to begin developing code within these files and then selectively stage them using the *git add* command.

**Command:**

```
$ git add lineplot1.r
$ git status
```

**Output:**

```
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   gitpaper1.Rproj
        new file:   lineplot1.r
Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)
        graph1.r
```

At this stage, only *lineplot1.r* was staged, while *graph1.r* intentionally remained untracked. In addition, project-level configuration files such as .gitignore and gitpaper1.Rproj were staged to ensure consistent project behavior and reproducibility across environments. This selective staging highlights a key principle of Git's version control workflow: files are added deliberately, not automatically. By consciously choosing which files to stage, programmers can create focused and meaningful commits, while keeping incomplete or exploratory work outside the version history.

Although selective staging provides fine-grained control, Git also offers convenience commands such as git add -a, which stage all modified and untracked files at once. This approach can be useful once users are comfortable with Git's workflow; however, it should be applied with caution in regulated environments. In this example, graph1.r was deliberately not staged to illustrate how unfinished work can be excluded from a commit.

It is important to note that staging a file using git add does not move the file into the local repository or remove it from the working directory. The file continues to reside in the working directory and can still be edited. Staging simply records a snapshot of the file's current state in the staging area as a candidate for the next commit. Only when the git commit command is executed are the staged changes permanently recorded in the local repository. This separation between the working directory, staging area, and repository allows Git to support deliberate, controlled version management rather than automatically saving every edit.

**7.2 Modifying a Staged File Before Commit**

After staging lineplot1.r, additional changes were made to the file by updating plot aesthetics and labels. Running git status after saving the file reveals an important aspect of Git's workflow: the same file can simultaneously appear as staged and modified.

**Command:**

```
$ git status
```

**Output:**

```
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   gitpaper1.Rproj
        new file:   lineplot1.r
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   lineplot1.r
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        graph1.r
```

In this situation, Git reports lineplot1.r under both "Changes to be committed" and "Changes not staged for commit." This indicates that Git has retained a snapshot of the file in the staging area, while also detecting newer edits in the working directory that have not yet been staged. The staged version represents what would be included in the next commit if the commit were executed immediately, whereas the unstaged changes reflect more recent modifications that remain outside the commit.

This behavior highlights Git's snapshot-based design as illustrated follows:

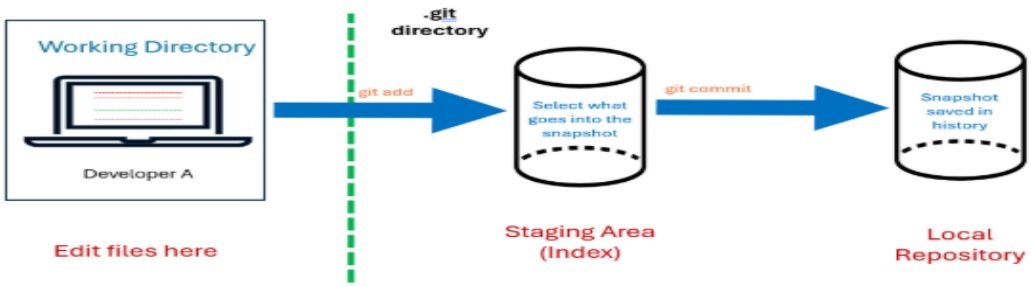



*Figure: From staging to history: git add and git commit.*

The working directory contains editable files. The git add command records a snapshot of selected changes in the staging area (index). The git commit command then saves that staged snapshot into the local repository as a permanent history. This figure is an original illustration adapted from the Git workflow concepts described in Library Carpentry: Introduction to Git and *Pro Git* (Chacon & Straub, 2014).

The git add command does not create a permanent record; instead, it prepares a specific version of a file for inclusion in the next snapshot. Any subsequent edits must be staged again if they are intended to be part of the same commit. This mechanism allows programmers to refine changes incrementally and ensures that only reviewed and intentional updates are permanently recorded.

**7.3 Git Commit (Snapshot Saved in Local Repository)**
A commit represents a permanent snapshot of the project at a specific point in time. When a commit is created, Git records the staged changes into the local repository along with metadata such as the author, timestamp, and a descriptive commit message.

***7.3.1 Creating the Initial Commit***
The following command was used to create the first commit in the repository:

**Command:**
```
$ git commit -m "Initial line plot (Version 1)"
```

**Output:**
```
[master (root-commit) dbaf49a] Initial line plot (Version 1)
3 files changed, 43 insertions(+)
create mode 100644 .gitignore
create mode 100644 gitpaper1.Rproj
create mode 100644 lineplot1.r
```

This command created the repository's root commit, as indicated by the label (root-commit) in the output. The commit message "Initial line plot (Version 1)" documents the purpose of this snapshot and establishes a clear baseline for future development. The commit records three files:

- .gitignore
- gitpaper1.Rproj
- lineplot1.r

The summary confirms that these files were newly created and added to version history, with a total of 43 lines inserted. At this point, the repository contains its first permanent snapshot, representing Version 1 of the analysis along with essential project configuration files.

***7.3.2 Repository State After the Commit***
Running git status immediately after the commit produces the following output:

**Command:**
```
$ git status
```

**Output:**
```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   lineplot1.r
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Graph1.r
no changes added to commit (use "git add" and/or "git commit -a")
```

This output shows that the repository is now on the master branch with a valid commit history. The file lineplot1.r appears under "Changes not staged for commit", indicating that it has been modified again in the working directory after the commit was created. These changes are not part of Version 1 and will not be recorded unless they are explicitly staged and committed in a subsequent step.

File *graph1.r* remains listed as untracked files, confirming that it has not been added to version control. This is intentional: *graph1.r* represents exploratory or in-progress work.

With a committed Version 1 in place, the repository is now ready for the next step: creating a new branch to modify and extend the code without altering the original baseline.

**7.4 Reviewing Uncommitted Changes with git diff**

In many development scenarios, a file is committed to establish a stable version and later modified without immediately creating a new commit. Before deciding whether these changes should be staged, committed, or discarded, it is often necessary to review exactly what has been altered. At this stage, the work remains entirely within Git; no interaction with GitHub is required.

After modifying *lineplot1.r* following the initial commit, running the git status command shows:

```
On branch master
Changes not staged for commit:
      modified:   lineplot1.r
```

This output confirms that *lineplot1.r* has been edited in the working directory but that the changes have not yet been staged. To examine the specific differences between the current working version of the file and the last committed snapshot, the git diff command is used:

**Command:**

```
$ git diff
```

```
$ git diff
diff --git a/lineplot1.r b/lineplot1.r
index 2c59f4e..f7818eb 100644
--- a/lineplot1.r
+++ b/lineplot1.r
@@ -1,22 +1,9 @@
-# new_graph.R
-# Create a simple scatter plot using ggplot2
-
-# Load the library
-library(ggplot2)
-
-# Create a small dataset
-data <- data.frame(
-  x = 1:10,
-  y = (1:10)^1.5
-)
-
-# Generate the plot
 ggplot(data, aes(x, y)) +
-  geom_point(color = "blue", size = 3) +
-  geom_line(color = "red", linewidth = 1) +
+  geom_point(color = "darkgreen", size = 4) +
+  geom_line(color = "steelblue", linewidth = 1.2) +
   labs(
-    title = "Simple Scatter Plot",
-    x = "X Values",
-    y = "Y Values"
+    title = "Simple Scatter Plot (Updated)",
+    x = "Input Values",
```

The resulting output displays a line-by-line comparison between the committed version of lineplot1.r and its modified version in the working directory. Lines prefixed with a minus sign (-) indicate content that has been removed or replaced, while lines prefixed with a plus sign (+) indicate newly added or modified content. In this example, the diff output shows changes to plotting aesthetics, including updated point colors, line styles, and revised axis labels and title.
Importantly, git diff does not change the state of the repository. It is a read-only inspection tool that allows programmers to review modifications before deciding on the next action. This capability is especially valuable in regulated or collaborative environments, where understanding and justifying code changes is essential.

At this point in the workflow, the changes to lineplot1.r exist only in the working directory. They are not part of the staging area and have not been recorded in version history. Based on the diff output, the programmer can now make an informed decision to either stage the updated file using git add, discard the changes using git restore, or continue refining the code before committing.

**7.5 Discarding Local Changes with git restore**

Unlike git add, which prepares changes for inclusion in a commit, git restore explicitly discards local modifications. As a result, it should be used with care, as discarded changes cannot be recovered unless they were previously committed. As we don't need the changes, we can restore original snapshot using following command which will also update our lineplot1.r file in working directory:

**Command:**

```
$ git restore lineplot1.r
$ git status
```

**Output:**

```
On branch master
nothing to commit, working tree clean
```

In contrast to git restore, which reverts files in the working directory, git reset operates on the staging area or commit history and should be used with additional caution.

This step completes the local Git review cycle—edit, inspect, and decide—before any commits are created, or changes are shared through GitHub, which will be introduced later in the workflow.

## 8. CREATING GITHUB REPOSITORY

### 8.1 Creating a GitHub Account and Authentication Methods

GitHub is a cloud-based platform that hosts Git repositories and enables collaboration, backup, and code sharing. Creating a GitHub account is a prerequisite for hosting repositories or contributing to shared projects. Account registration is performed through the GitHub website (https://github.com) by providing a username, email address, and password, followed by email verification (GitHub, Inc., n.d.-d).

When interacting with GitHub from a local Git repository, authentication is required to securely push and pull changes. GitHub supports multiple authentication mechanisms, of which HTTPS and SSH are the most commonly used (GitHub, Inc., n.d.-a). With HTTPS authentication, users authenticate using their GitHub username and a personal access token (PAT) rather than a password. This approach is straightforward to configure and is widely supported in corporate and regulated environments, where SSH access may be restricted by security policies or firewalls (GitHub, Inc., n.d.-a, n.d.-c).

In contrast, SSH authentication relies on a cryptographic key pair generated on the local machine and registered with the GitHub account. Once configured, SSH allows password-less authentication and is convenient for frequent GitHub users. However, it requires additional setup and may not be permitted in all organizational environments (GitHub, Inc., n.d.-b).

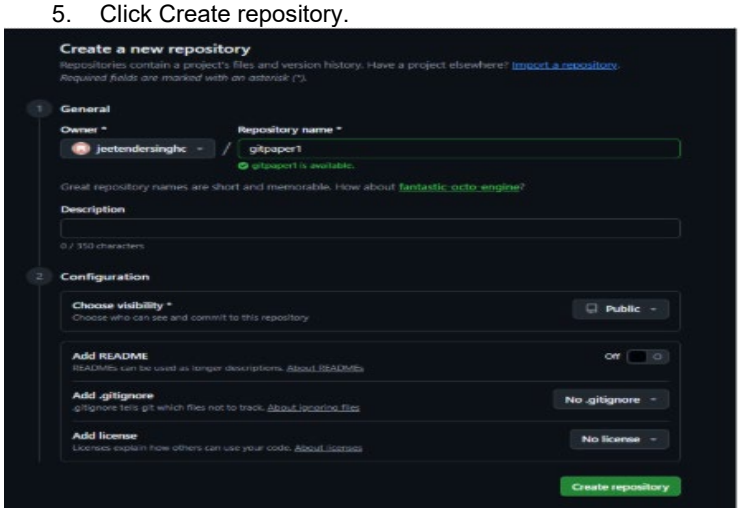
In this paper, HTTPS authentication is used for connecting the local repository to GitHub, as it provides a simple, secure, and broadly compatible method for sharing code. Readers interested in detailed instructions for account creation or authentication setup are referred to the official GitHub documentation.

### 8.2 Create a Repository on GitHub

#### *8.2.1 Step 1: Create an Empty Repository on GitHub*

1. Log in to GitHub.


2. Click the “+” icon in the top-right corner and choose New repository.
3. Enter a repository name (example: gitpaper1).
4. Do not initialize with a README, .gitignore, or license at this stage (start with an empty repo so it matches your local repository).
5. Click Create repository.



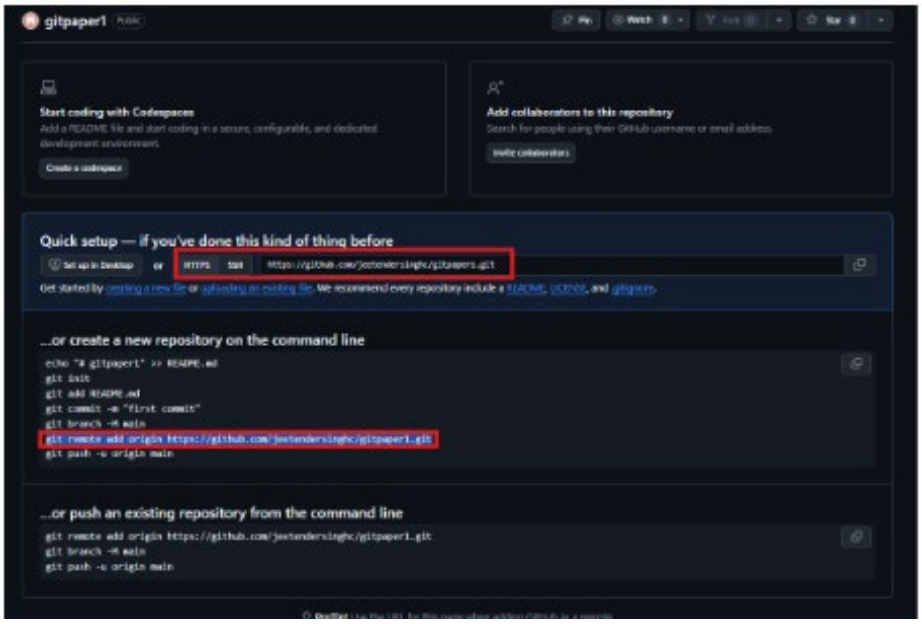
### *8.2.2 Step 2: Copy the Remote URL (SSH or HTTPS)*

On the GitHub setup page, choose either:

- SSH (URL starts with git@github.com:...)
- or HTTPS (URL starts with https://github.com/...)

GitHub will show a “Quick setup” page with connection commands.

### *8.2.3 Step 3: Add the GitHub Repository as a Remote (origin)*




In Git Bash / terminal, go to your local repository folder (gitpaper1) and run:

```
git remote add origin git@github.com:<your_github_username>/gitpaper1.git
```

- remote add tells Git you want to connect to a remote repository.
- origin is the conventional nickname for the primary remote.
- Replace <your_github_username> with your actual GitHub username.

Verify it was added:

```
git remote -v
```

```
$ git remote -v
origin  https://github.com/[redacted]/gitpaper1.git (fetch)
origin  https://github.com/[redacted]/gitpaper1.git (push)
```

You should see origin listed for both fetch and push.

### *8.2.4 Step 4: Push Your Local Commits to GitHub over HTTPS*

Once the remote is configured, push your current branch to GitHub. For example, if your branch is master:

**Command:**

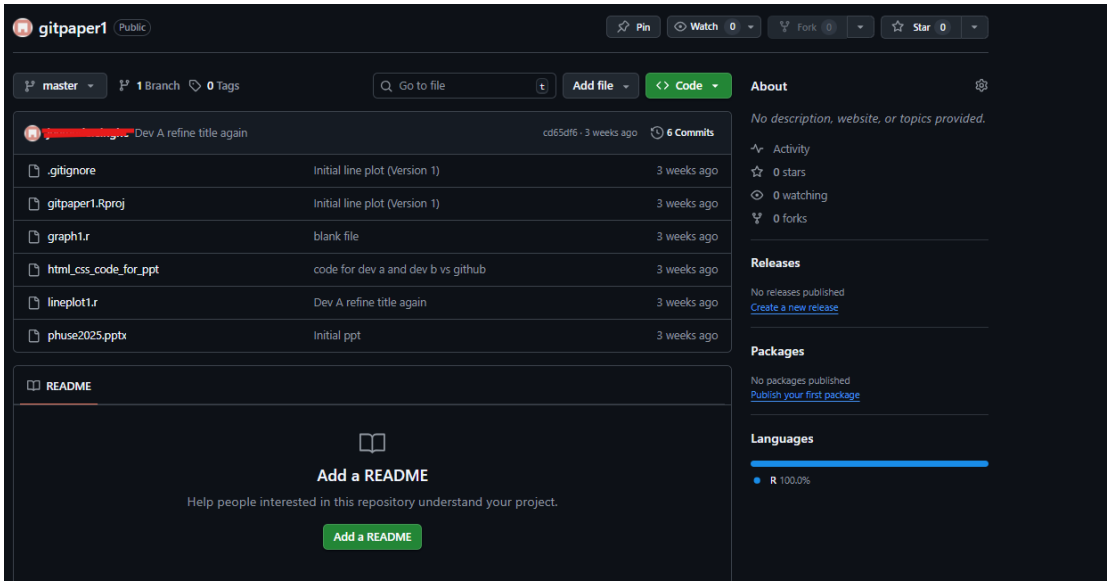
```
git push -u origin master
```

**Output:**

```
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (11/11), 41.37 KiB | 6.89 MiB/s, done.
Total 11 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/jeetendersinghc/gitpaper1.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

This command instructs Git to push the local master branch to the remote repository named origin. The -u (or --set-upstream) option establishes a tracking relationship between the local branch and its remote counterpart, allowing future updates to be pushed with a simple git push command.

The output generated during this process provides insight into how Git transfers data to the remote repository.
After this step, refresh GitHub in the browser—your committed files should now appear online.




***8.2.5 Step 5: Pull Changes from GitHub (when collaborating)***
If changes are made on GitHub (for example, by a collaborator or by editing a file via the GitHub website), you bring them down to your computer using:

```
git pull origin master
```

git pull is effectively:

- fetching remote updates and
- merging them into your local branch.

## 9. EXAMPLE SCENARIO: DEVELOPER A AND DEVELOPER B (PUSH/PULL AND A MERGE CONFLICT)

To demonstrate how collaboration conflicts occur, two separate working copies of the same GitHub repository were created on a single machine to simulate two developers:

- Developer A workspace: gitpaper1-devA
- Developer B workspace: gitpaper1-devB

Both clones are connected to the same remote repository on GitHub (origin/master). This setup mimics a real team environment where each developer has an independent local copy of the repository history.

### 9.1 Step 1: Developer A Makes a Change and Pushes it to GitHub

In the gitpaper1-devA directory, lineplot1.r was modified (refining the plot title). Git detected the change as a modification in the working directory. The updated file was staged and committed:

```
git add lineplot1.r
git commit -m "Dev A refine title again"
```

Developer A then pushed the commit to GitHub:

```
git push
```

The successful push updated the remote branch, moving origin/master forward (from commit f82b1c9 to cd65df6). At this point, GitHub contained Developer A's newer commit, while Developer B's local copy had not yet incorporated it.

### 9.2 Step 2: Developer B Edits the Same File Without Pulling First

In the separate gitpaper1-devB working copy, Developer B also edited the same line(s) in lineplot1.r (updating the title differently). The file was staged and committed locally:

```
git add lineplot1.r
git commit -m "Dev B: Update title"
```

At this stage, Developer B's repository contained a local commit that diverged from the remote branch, because GitHub already had Developer A's newer commit.

**9.3 Step 3: Developer B Attempts to Push and Git Rejects it ("fetch first")**
When Developer B attempted to push:

```
git push
```

Git rejected the update with a message indicating that the remote contains work that the local repository does not have.

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/[redacted]/gitpaper1.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This rejection is expected and is a core safety feature: Git prevents a developer from overwriting remote history when the remote branch has moved ahead. The correct next step is to pull the remote updates into the local branch and resolve any conflicts if both developers changed the same lines.

**9.4 Step 4: Developer B Pulls, Resolves Conflict, Then Pushes the Merged Result**
To integrate the remote changes, Developer B runs:

```
git pull
```

```
[redacted] ~/OneDrive - [redacted] LLC/Demo/gitpaper1-devB (master)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 303 bytes | 11.00 KiB/s, done.
From https://github.com/[redacted]/gitpaper1
   f82b1c9..cd65df6  master     -> origin/master
Auto-merging lineplot1.r
CONFLICT (content): Merge conflict in lineplot1.r
Automatic merge failed; fix conflicts and then commit the result.
```

Once developer B runs git pull, GitHub is sending the commits

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 303 bytes | 11.00 KiB/s, done.
```

that exist on the remote branch but not in your local repository. These include Developer A's recent commit.

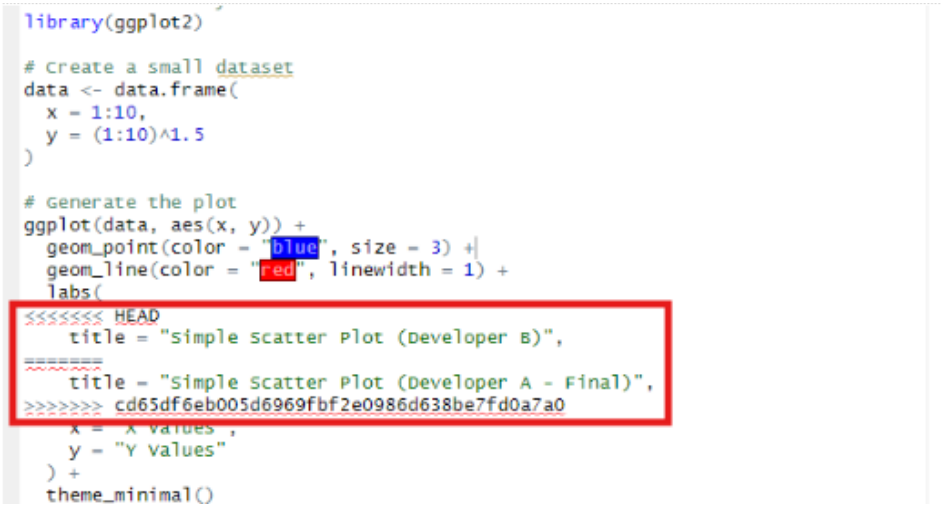
```
From https://github.com/[redacted]/gitpaper1
   f82b1c9..cd65df6  master     -> origin/master
```

The output tells that origin/master moved forward and GitHub now has commits your local master did not previously contain. At this point, Git tries to merge those remote changes into your local branch.

```
Auto-merging lineplot1.r
CONFLICT (content): Merge conflict in lineplot1.r
Automatic merge failed; fix conflicts and then commit the result.
```

Git attempted to merge the changes automatically but paused the process because both Developer A and Developer B modified the same section of the file, leaving Git unable to determine which version to keep; as a result, the repository entered a conflicted state with no merge commit created and conflict markers inserted into lineplot1.r for manual resolution.

At this stage, Git has paused the merge process and placed the repository into a conflicted state. No merge commit has been created, and the file lineplot1.r now contains conflict markers indicating competing changes from the local and remote branches.



```
library(ggplot2)

# Create a small dataset
data <- data.frame(
  x = 1:10,
  y = (1:10)^1.5
)

# Generate the plot
ggplot(data, aes(x, y)) +
  geom_point(color = "blue", size = 3) +
  geom_line(color = "red", linewidth = 1) +
  labs(
<<<<<<< HEAD
    title = "Simple Scatter Plot (Developer B)",
=======
    title = "Simple Scatter Plot (Developer A - Final)",
>>>>>>> cd65df6eb005d6969fbf2e0986d638be7fd0a7a0
    x = "X Values",
    y = "Y Values"
  ) +
  theme_minimal()
```

Running git status confirms this condition by listing lineplot1.r as “both modified,” signaling that manual intervention is required. When the file is opened in an editor, Git clearly delineates the conflicting sections using standard markers: the block labeled HEAD represents the local version from Developer B, while the block labeled origin/master represents the version fetched from GitHub that was previously committed by Developer A.

To resolve the conflict, the programmer must edit the file to produce a final, agreed-upon version and remove all conflict markers. Once the file reflects the intended outcome, the resolution is recorded by staging the file with git add, completing the merge with git commit, and pushing the result to GitHub using git push. After these steps, the remote repository contains both developers’ original commits as well as a merge commit that transparently documents how the conflict was resolved.

## 10. CONCLUSION

This paper demonstrated how Git supports reliable collaboration by tracking changes, preventing accidental overwrites, and enforcing explicit conflict resolution through its push–pull workflow. Using a practical example, we illustrated how concurrent edits to the same file can lead to merge conflicts and how Git clearly identifies and manages such situations to preserve a transparent and auditable history. This example reflects one of the most common conflict scenarios encountered in collaborative programming and highlights the importance of disciplined version control practices.

While many additional conflict patterns and branching strategies exist, it is not feasible to cover all possible scenarios within a single paper. More advanced workflows—such as detailed branch management, HEAD movement across multiple commits, and complex merge scenarios involving multiple contributors—will be explored in a subsequent edition. Together, these topics will build on the foundational concepts presented here and further demonstrate how Git enables scalable, controlled, and reproducible software development.

## 11. REFERENCES


Chacon, S., & Straub, B. (2014). *Pro git* (2nd ed.). Apress/Springer Nature. https://link.springer.com/book/10.1007/978-1-4842-0076-6

De Sterck, H., Shu, C.-W., & Abgrall, R. (2023). Enhancing reproducibility of research papers in SISC, JSC and JCP. *Journal of Scientific Computing*, *95*(77). https://doi.org/10.1007/s10915-023-02193-7

GitHub, Inc. (n.d.-a). *About authentication to GitHub*. https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/about-authentication-to-github

GitHub, Inc. (n.d.-b). *Connecting to GitHub with SSH*. https://docs.github.com/en/authentication/connecting-to-github-with-ssh

GitHub, Inc. (n.d.-c). *Creating a personal access token*. https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

GitHub, Inc. (n.d.-d). *Signing up for GitHub*. https://docs.github.com/en/get-started/signing-up-for-github/signing-up-for-a-new-github-account

GitHub, Inc. (2025). *GitHub documentation*. https://docs.github.com

Huskey, S. J. (2024). Committing to reproducibility and explainability: Using git as a research journal. *International Journal of Digital Humanities*, *6*, 9–21. https://doi.org/10.1007/s42803-023-00076-9

National Institutes of Health. (2015). *Version control guidelines (version 2.0)*. https://files.nccih.nih.gov/s3fs-public/CR-Toolbox/Version_Control_Guidelines_ver2_07-17-2015.pdf

UBC Library Research Commons. (n.d.). *Getting started with git*. https://ubc-library-rc.github.io/intro-git/content/02_getting_started.html

U.S. Food and Drug Administration. (2003). *Part 11, electronic records; electronic signatures – scope and application*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application

U.S. Food and Drug Administration. (2018). *Data integrity and compliance with drug CGMP: Questions and answers*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/data-integrity-and-compliance-drug-cgmp-questions-and-answers

## 12. ACKNOWLEDGMENTS

The authors would like to thank management teams from Merck & Co., Inc., Rahway, NJ, USA, for their advice on this paper/presentation The authors are also grateful for reviewing the paper and providing feedback by Erica Davis, Changhong Shi, Abhilash Vasu Chimbirithy, Chao Su and our department head Amy Gillespie.

## 13. CONTACT INFORMATION

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

**Jeetender Chauhan**
Merck & Co., Inc., Rahway, NJ, USA
Email: jeetender.chauhan@merck.com
**Madhusudhan Ginnaram**
Merck & Co., Inc., Rahway, NJ, USA
Email: madhusudhan.ginnaram.reddy@merck.com
**Sarad Nepal**
Merck & Co., Inc., Rahway, NJ, USA
Email: sarad.nepal@merck.com
**Bingjun Wang**
Merck & Co., Inc., Rahway, NJ, USA
Email: benjamin.wang@merck.com



***Disclaimer: The views and opinions presented in the poster are our own and do not necessarily reflect the official position of the company***