

DevOps

Git Lesson 4&5

How Git Keeps Track of Files and Changes

Git is a distributed version control system, and at its core, it organises information into a few fundamental object types: **blobs**, **trees**, and **commits**. These objects are tied together using cryptographic hashes (SHA-1) so that Git can reliably track changes across time. Let's unpack how this works.



The Role of Blobs


A **blob** (binary large object) in Git stores the raw content of a file. It does not include metadata such as filenames or permissions; it simply represents the exact snapshot of a file's contents at a given moment. Every blob is identified by a unique SHA-1 hash, which is generated by running the file through Git's hashing function.

For example, if you create a file called `README.md` and run:

```
git hash-object -w README.md
```

Git computes a hash for that file and stores it under `.git/objects`. If you later change even a single character in the file, a completely new hash will be produced, allowing Git to differentiate between the two versions.

This is why blobs are so efficient: Git does not think in terms of line-by-line differences but instead treats each file's entire content as an object. The same file content always maps to the same hash, avoiding duplication.



The Role of Trees

While blobs capture file contents, **trees** represent the directory structure. A tree object contains entries that map filenames to blob objects (for files) or to other trees (for subdirectories). Each entry in a tree records:

- File permissions (e.g., 100644 for a normal file, 040000 for a directory)
- File type (blob or tree)
- The SHA-1 hash of the object
- The filename

For example, running:

```
git cat-file -p <tree-hash>
```


might show:

```
100644 blob <hash1>  file1.txt
```

```
100644 blob <hash2>  file2.txt
```

```
040000 tree <hash3>  src
```

This tells Git that the project directory contains two text files and one subdirectory (src). In this way, trees link blobs together into meaningful directory structures.



The Role of Commits

A **commit** ties everything together. Each commit points to exactly one tree object, which represents the root of the project at that point in time. Commits also store metadata: the author, date, commit message, and references to parent commits. The SHA-1 hash of the commit uniquely identifies it.

This chain of commits allows Git to build a complete history. For example:

- Commit A points to Tree X
- Commit B points to Tree Y and also lists Commit A as its parent

By walking back through this chain, Git reconstructs how the repository evolved. This is what powers commands like `git log`.



Blobs, Trees, and Commits Together

- A **blob** is the snapshot of a file's contents.
- A **tree** organises blobs into directories.
- A **commit** records the state of the project by pointing to a tree, with links to its parent commits.

Think of it this way: **blob = file**, **tree = folder**, **commit = project snapshot**. This three-layer model is the foundation of Git's powerful tracking capabilities.



Exploring Git Objects

Git provides commands to inspect its internal objects:

To see the type of an object:

```
git cat-file -t <hash>
```

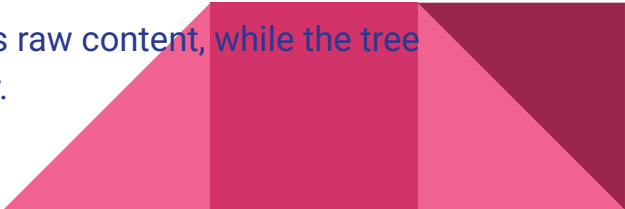
To view its contents:

```
git cat-file -p <hash>
```

To list the internal .git directory structure:

```
tree /f .git
```

For instance, if you hash a file and then inspect it, you'll notice the blob contains raw content, while the tree contains the file's name and permissions, and the commit brings them together.



Collaborating with Others

While blobs, trees, and commits are at the heart of Git, collaboration requires pushing and pulling changes across remote repositories such as GitHub or GitLab.

To add a remote repository, you can use:

```
git remote add origin https://github.com/username/repository.git
```

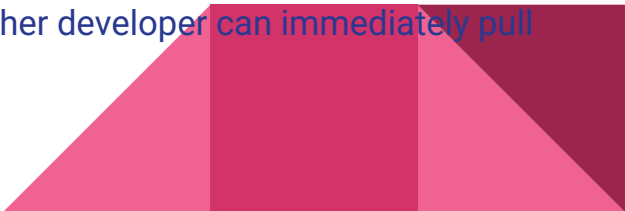
You can then push your local commits to the remote server:

```
git push origin main
```

And to pull in changes made by others:

```
git pull origin main
```

For example, in a team project, one developer may push new features, and another developer can immediately pull them to their own system. This ensures everyone stays in sync.



The Role of .gitignore


The `.gitignore` file tells Git which files or directories to ignore. Ignored files are not tracked, won't be included in commits, and won't be pushed to remote repositories. This prevents unnecessary or sensitive files (like logs, build artifacts, or secrets) from cluttering the repo.

The file is usually placed at the project root, but multiple `.gitignore` files can exist in different directories if needed.

Examples include:

- File: `secret.txt`
- File type: `*.log`
- Directory: `/build/, logs/*`
- Exceptions: `*.log` with `!important.log`
- Wildcards: `*` (many chars), `?` (one char), `**` (multi-level)

Common cases:

- Node.js → `node_modules/, npm-debug.log`
 - Python → `__pycache__/, *.pyc, .venv/`
- 

Git Workflows Overview

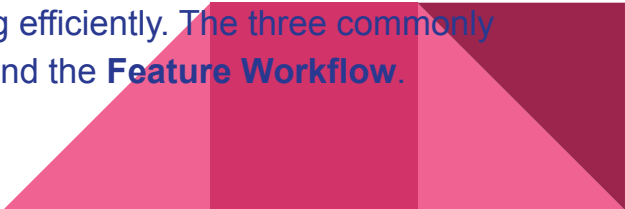
Git workflows provide structured ways to manage code in a project.

Different workflows suit different teams and project needs.

Three popular workflows are:

1. **Centralized Workflow**
2. **Master Workflow**
3. **Feature Workflow**

Git workflows are essentially strategies that define how developers in a team collaborate and manage code changes. They provide a structured way of using Git so that code contributions remain organised, conflicts are minimised, and the project can scale with the needs of the team. Different workflows cater to different types of teams. For instance, a small team or solo developer may prefer something very lightweight, while larger teams with many parallel tasks require a workflow that handles branching and merging efficiently. The three commonly adopted Git workflows are the **Centralized Workflow**, the **Master Workflow**, and the **Feature Workflow**.



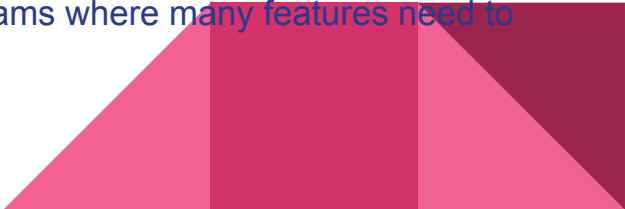
Centralized Workflow

The Centralized Workflow resembles the traditional version control systems such as Subversion, where there is only one central repository and all changes eventually flow into a single main branch. This makes it especially suitable for teams that are transitioning from older centralized systems into Git, as the working style feels familiar.

In this workflow, the central repository has one main branch — often called `main` or `master`. Developers clone this central repository to their local machines. They may choose to create temporary local branches while working, but eventually all changes must be pushed directly to the central branch. Before pushing, developers are responsible for resolving any conflicts locally to ensure that the shared branch remains stable. The process is straightforward:

- Clone the repository.
- Make changes and commit them locally.
- and finally push to the main branch.

This approach keeps everything simple but can become restrictive for larger teams where many features need to be developed in parallel.



Master Workflow

The Master Workflow is even simpler and is designed with the smallest setups in mind — either solo developers or very small teams. In this approach, all development work happens directly on the master (or main) branch without the use of additional branches. Each commit is made directly on this branch, which means there is no overhead of managing merges or synchronising multiple lines of development.

This workflow works well when the complexity of the project is low and when the risk of conflicts is minimal. Tags are sometimes used in this model to mark important milestones, such as release points or version updates. While it eliminates the overhead of branching, it is also less flexible, as all contributors are effectively working in the same space, which can lead to conflicts if the team grows larger.



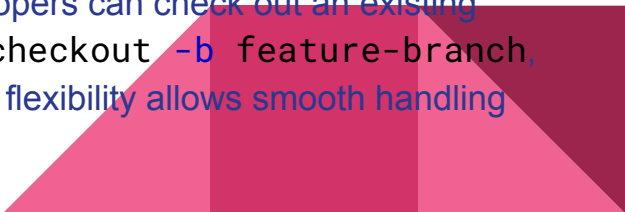
Feature Workflow

The Feature Workflow is the most flexible and is widely used in modern software teams, particularly those working on multiple features or tasks at the same time. In this model, the main branch (often called `main` or `master`) is treated as the stable branch that always reflects production-ready code. Developers do not work directly on this branch. Instead, every new feature, bug fix, or experiment is developed in its own separate branch.

The process begins by creating a new branch from the main branch for the specific feature under development. The developer then commits all changes related to that feature on the feature branch. Once the feature is complete and tested, it is merged back into the main branch, usually through a merge request or pull request so that other team members can review the changes. After the merge, the feature branch can safely be deleted. This keeps the repository clean and ensures that the main branch remains stable while still supporting multiple parallel streams of work.

Merging in this workflow is an essential operation. The `git merge` command allows you to combine the changes from the feature branch into the main branch.

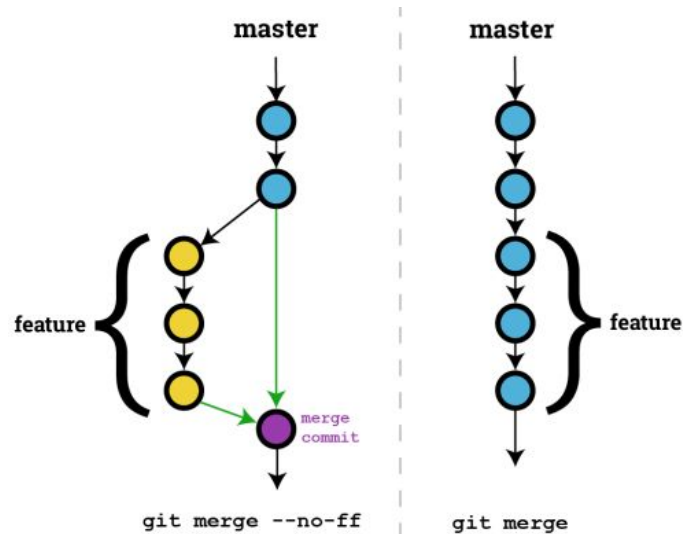
Creating and switching to feature branches is also straightforward in Git. Developers can check out an existing branch with `git checkout feature-branch`, create a new one with `git checkout -b feature-branch`, or use the more modern `git switch -c feature-branch` command. This flexibility allows smooth handling of concurrent work without disrupting the stability of the main branch.



Fast-Forward Merge in Git

A fast-forward merge is the simplest type of merge that Git can perform. It happens when the branch being merged has a direct, linear relationship with the branch you are merging into. In other words, **if no new commits have been made on the target branch since the feature branch was created**, then Git can simply move the pointer of the target branch forward to the latest commit on the feature branch. This avoids creating any new merge commit and keeps the project history clean and linear.

To understand this better, imagine you start a new feature branch from main. You work on this branch, make a few commits, while no one else touches main. When you are ready to merge back, Git sees that main has not progressed since you branched off. Instead of creating a new merge commit, it just advances main to point to the tip of your feature branch. The effect is as if you had committed directly to main, and the history remains a straight line.

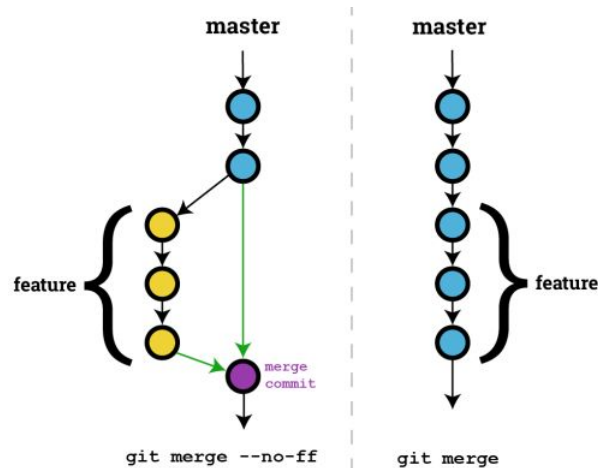


Fast-Forward Merge in Git

The process of performing a fast-forward merge involves two steps. First, you switch to the branch you want to update — usually **main** — using **git checkout main**. Then, you run the merge command **git merge feature-branch**. If a fast-forward is possible, Git will simply advance the pointer of main to the last commit on feature-branch.

One key aspect of fast-forward merges is that they preserve a very clean and linear commit history. This makes it easy to trace through the sequence of changes without any branching or merge commits cluttering the history. However, some teams prefer to avoid fast-forward merges by using the **--no-ff** option. This creates an explicit merge commit even when a fast-forward would have been possible. The advantage of **--no-ff** is that it preserves the fact that a feature was developed in its own branch, making the project history clearer when reviewing development patterns.

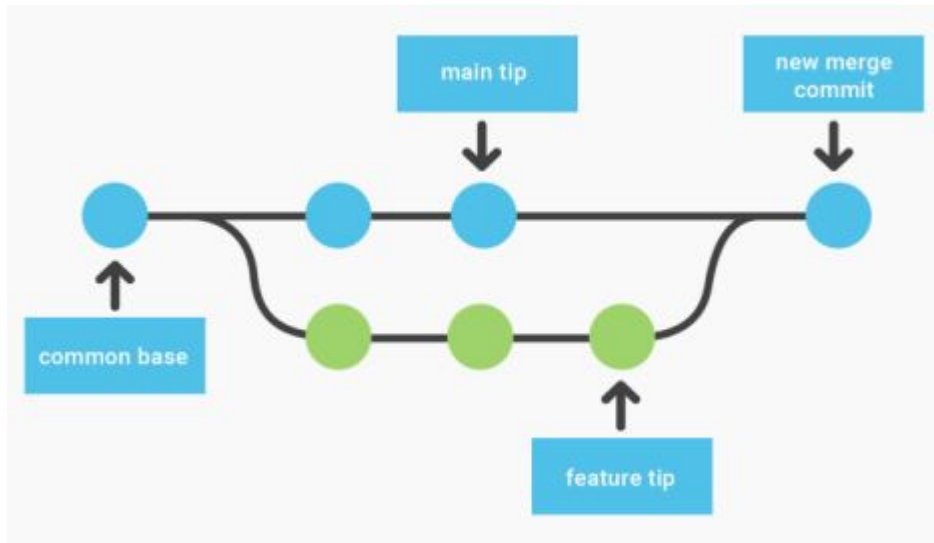
In short, fast-forward merges are excellent for simple scenarios where you want to keep history minimal and linear. On the other hand, disabling them with **--no-ff** can provide better visibility into how different features were developed and integrated.



Three-Way Merge in Git

A three-way merge happens when the **target branch** and the **source branch** have both diverged — meaning new commits have been added to each branch since they split from a common base. Unlike a fast-forward merge, which simply moves the branch pointer ahead, a three-way merge must actively combine two different lines of history into one.

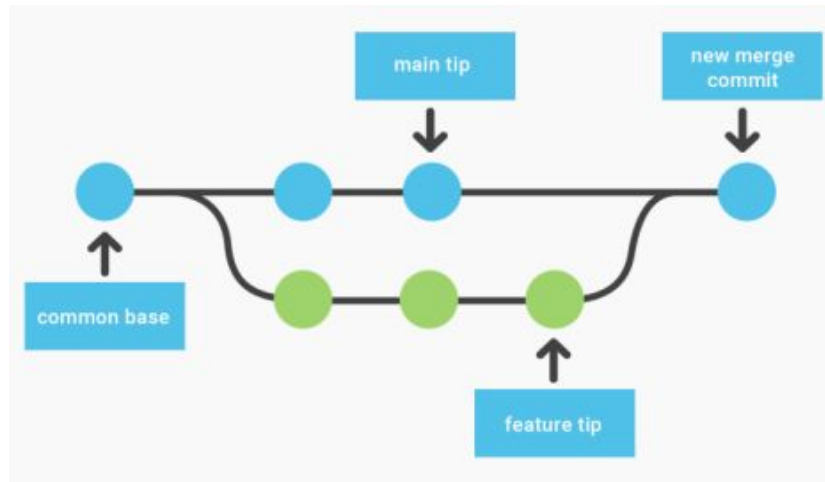
When Git performs a three-way merge, it looks at three points: the commit where the two branches originally diverged (the common base), the last commit on the target branch (for example, `main`), and the last commit on the source branch (for example, `feature`). Using these three points, Git creates a new **merge commit** on the target branch. This merge commit has two parent commits: one from the target branch and one from the source branch. The new commit represents the integration of changes from both branches.



Example to Understand Three-Way Merge

Imagine you are working on a project with a teammate. You both start from the same main branch. You create a new branch called feature to add a login page, while your teammate stays on main and updates the project's README file. Both of you commit changes independently. Now, when you try to merge your feature branch back into main, Git cannot simply fast-forward, because main has commits (the README update) that are not present in your feature branch.

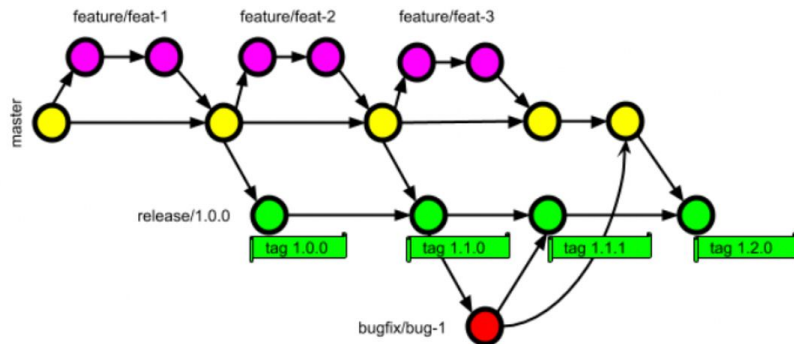
Here, Git performs a three-way merge. It looks at the common ancestor (the commit before you branched), your commits on the feature branch (login page code), and the new commits on the main branch (README update). Git then creates a new merge commit on main that brings both changes together: the login page code and the updated README. The history now shows that two different lines of work came together, and this is captured by the merge commit with two parents.



Git Tagging

In Git, tags are a way of marking specific points in a project's history that you want to remember or reference later. Unlike branches, which move forward as new commits are added, tags are static markers that stay tied to a single commit forever. This makes them especially useful for denoting milestones in a project, such as marking the commit where version 1.0 of the software was released.

For example, once you tag a commit as `v1.0`, no matter how many new commits are added to the repository, the `v1.0` tag will always point back to that exact commit. This gives teams a reliable way to refer to releases, fixes, or other important points in the project's lifecycle.



Types of Git Tags

Git provides two main types of tags: **lightweight tags** and **annotated tags**.

A **lightweight tag** is the simpler of the two. It is nothing more than a pointer to a specific commit in the repository. It does not carry any extra information such as the author of the tag, the date, or a description. In practice, this means a lightweight tag is just a name attached to a commit. For example, running `git tag v1.0` creates a tag named `v1.0` pointing to the current commit. You could also tag an earlier commit directly by specifying its hash, for instance `git tag v1.0 <commit_hash>`. Lightweight tags are useful when you want a quick, no-frills marker for a commit.

An **annotated tag**, on the other hand, is much richer. Git treats it as a full object in its database, which means it is stored permanently with metadata. Annotated tags contain information such as the tagger's name, email, date, and an optional message describing the tag. For example, you might run:

```
git tag -a v1.0 -m "Version 1.0 release"
```

This creates an annotated tag `v1.0` with a message describing the release. Annotated tags are typically used in real-world projects because they provide context. If someone looks back months later, they will see not just the commit but also who created the tag and why. Think of it like this: **lightweight tags are like a sticky note** you place on a page in a book, while **annotated tags are like a bookmark with a written note about why that page matters**.

Pushing Tags to a Remote Repository

By default, when you push commits to a remote repository like GitHub or GitLab, your **tags are not pushed** along with them. This is because **tags are considered separate objects from commits**. If you want your teammates to see and use the same tags, you have to push them explicitly. If you want to push a single tag, you use:

```
git push origin v1.0
```

This sends the `v1.0` tag to the remote. If you have created several tags and want to push all of them at once, you can run:

```
git push --tags
```

This is especially common after tagging multiple release points. Sometimes tags also need to be deleted, either because they were created by mistake or because the release process has changed. To delete a tag locally, you can run:

```
git tag -d v1.0
```

However, this only removes the tag from your machine. If the tag already exists on the remote repository, you also need to delete it there using:

```
git push origin --delete v1.0
```

This ensures consistency across the team and avoids confusion.




Git Best Practices – Writing Clean Code with Git

One of the most important habits is writing **clear commit messages**. A good commit message helps both you and your teammates understand why a change was made. A widely used convention is to have a short subject line of no more than 50 characters, followed by an optional body. The body can describe the change in detail, separated by a blank line. For example, a subject line might be *“Fix login bug”*, and the body might explain what exactly was changed and why.

Another best practice is to **make atomic commits**. This means each commit should represent one logical change only. For instance, if you fix a bug and also update documentation, these should be two separate commits rather than bundled together. This makes it easier to trace changes, revert specific fixes if necessary, and review history without confusion.

Branches should also be used **effectively**. Instead of committing everything directly to `main`, it is better to create separate branches for each new feature or bug fix. Following a clear naming convention, such as `feature/login-page` or `bugfix/cart-issue`, helps everyone on the team understand the purpose of a branch at a glance.



Git Best Practices – Writing Clean Code with Git

Before merging a branch back into the main codebase, it is best practice to **review the code**. This is often done using a merge request or pull request, where other developers can review the changes, suggest improvements, or catch mistakes before the code reaches production.

Finally, it is essential to **avoid committing sensitive or unnecessary information**. Files like build artifacts, temporary files, or IDE-generated configurations should be excluded using a `.gitignore` file. More importantly, sensitive data such as passwords, API keys, or personal information should never be committed to Git. Not only does this keep the repository clean, but it also avoids serious security risks.

Together, these practices make a repository professional, maintainable, and safe for long-term collaboration.



Git Advanced Commands – Cherry-pick

Sometimes, instead of merging an entire branch, you may only want a single commit from it. This is where **Git cherry-pick** comes in. There are several common **use cases** for cherry-pick. One is when you want a bug fix or feature commit from another branch without merging its entire history. Another is when a hotfix was applied to main, and you want to apply that same fix to your feature branch. A third case is when a teammate's commit contains something useful that you need, and you want to import only that specific commit.

The syntax is straightforward. To cherry-pick a single commit, you run:


```
git cherry-pick <commit-hash>
```

If you need a **range of commits**, you can specify a start and end:

```
git cherry-pick <start-commit>^..<end-commit>
```

And if you want multiple specific commits, you can list them:

```
git cherry-pick <commit1> <commit2> <commit3>
```

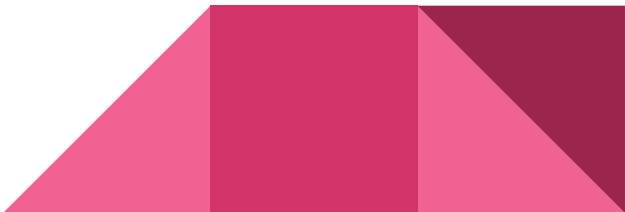


Git Rebase

Rebase is a powerful Git command that allows you to rewrite the history of your branch so it looks as if your work was built directly on top of the latest version of another branch. Instead of merging two branches with a merge commit, rebase takes the commits from your branch and re-applies them one by one on top of the target branch.

Imagine you start a branch `feature-x` from `main` when `main` is at commit A. On your feature branch, you add commits D, E, and F. Meanwhile, other developers update `main` with commits B and C. Now your history looks like this:

```
main:    A---B---C
          |
feature-x: D---E---F
```

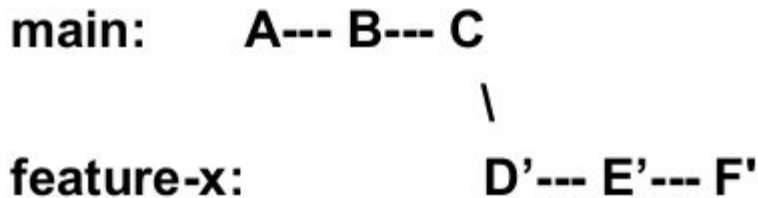


Git Rebase

If you run `git rebase main` while on `feature-x`, Git takes your commits (D, E, F), rewinds your branch back to `main`, and then replays those commits on top of C. The new history looks like the diagram on the right of this slide.

Notice that the commits on `feature-x` now appear as if they were created after C. In fact, Git creates new commits (D', E', F') that contain the same changes as your originals but with a new base.

Rebase can be thought of as *“I want my work to appear as if I started from the latest version of main.”*



Why Rebase Is Useful

Suppose you are working on a login feature (feature-x) while your teammate improves the UI on main. If you merge your branch into main later, the history will show a **merge commit**, which keeps track of the two branches diverging and then converging. While accurate, this can clutter the history when many small branches are merged.

By rebasing instead, you rewrite your commits so it looks like you developed the login feature after the UI updates. This keeps the history linear and easier to follow, as if everything was developed in a straight line.



Git Merge vs Git Rebase

Both merge and rebase are tools to integrate changes from one branch into another, but they differ in philosophy and outcome.

- **Merging** preserves the actual history. It shows the branches diverging and then coming back together with a merge commit. This is like a CCTV recording — it shows exactly what happened, even if the history is messy.
- **Rebasing** rewrites history to make it look like development happened in a neat, linear sequence. This is like a well-edited novel — it tells the story cleanly without showing all the detours.

In practice, teams choose based on their needs. Individual developers and open-source projects often prefer rebasing followed by a fast-forward merge, because it keeps the repository tidy and easy to navigate. For example, many open-source projects like React or the Linux kernel encourage contributors to rebase before submitting patches.

Large enterprise teams, however, often prefer merge commits, especially three-way merges, because they value traceability and an audit trail. The merge commit clearly shows when two lines of development were combined, which can be important for debugging or compliance. GitHub itself defaults to merge commits for pull requests because it is safer for collaboration and avoids rewriting shared history.