About merge methods on GitHub

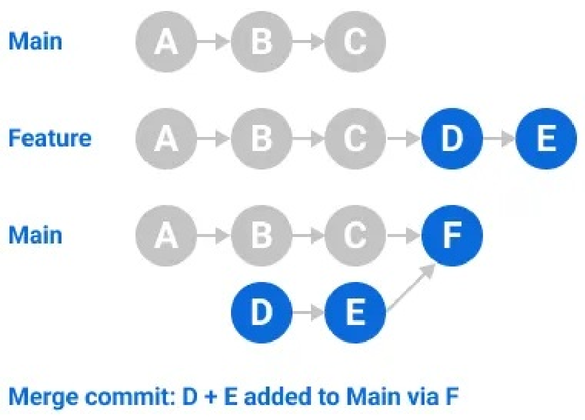
In this article

You can allow contributors with push access to your repository to merge their pull requests on your GitHub Enterprise Server instance with different merge options or enforce a specific merge method for all of your repository's pull requests.

You can configure pull request merge options on your GitHub Enterprise Server instance to meet your workflow needs and preferences for managing Git history. For more information, see "Configuring pull request merges." You can enforce one type of merge method, such as commit squashing or rebasing, by only enabling the desired method for your repository.

When you click the default **Merge pull request** option on a pull request on your GitHub Enterprise Server instance, all commits from the feature branch are added to the base branch in a merge commit. The pull request is merged using the `--no-ff` option.

To merge pull requests, you must have write permissions in the repository.

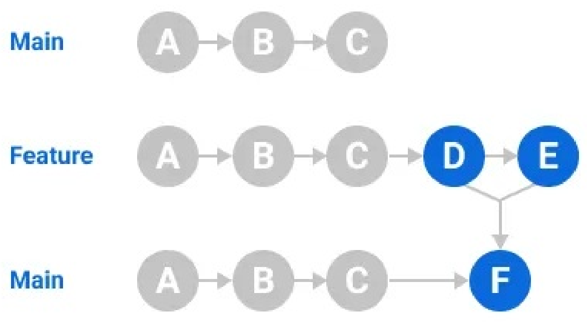


The default merge method creates a merge commit. You can prevent anyone from pushing merge commits to a protected branch by enforcing a linear commit history. For more information, see "About protected branches."

Squashing your merge commits

When you select the **Squash and merge** option on a pull request on your GitHub Enterprise Server instance, the pull request's commits are squashed into a single commit. Instead of seeing all of a contributor's individual commits from a topic branch, the commits are combined into one commit and merged into the default branch. Pull requests with squashed commits are merged using the fast-forward option.

To squash and merge pull requests, you must have write permissions in the repository, and the repository must allow squash merging.



Squash and merge: D + E into F in Main

You can use squash and merge to create a more streamlined Git history in your repository. Work-in-progress commits are helpful when working on a feature branch, but they aren't necessarily important to retain in the Git history. If you squash these commits into one commit while merging to the default branch, you can retain the original changes with a clear Git history.

Before enabling squashing commits, consider these disadvantages:

- You lose information about when specific changes were originally made and who authored the squashed commits.
- If you continue working on the head branch of a pull request after squashing and merging, and then create a new pull request between the same branches, commits that you previously squashed and merged will be listed in the new pull request. You may also have conflicts that you have to repeatedly resolve in each successive pull request. For more information, see "About pull request merges."
- Some Git commands that use the "SHA" or "hash" ID may be harder to use since the SHA ID for the original commits is lost. For example, using `git rerere` may not be as effective.

For more information, see "Configuring commit squashing for pull requests."

Rebasing and merging your commits

When you select the **Rebase and merge** option on a pull request on your GitHub Enterprise Server instance, all commits from the topic branch (or head branch) are added onto the base branch individually without a merge commit. In that way, the rebase and merge behavior resembles a fast-forward merge by maintaining a linear project history. However, rebasing achieves this by re-writing the commit history on the base branch with new commits.

The rebase and merge behavior on GitHub Enterprise Server deviates slightly from `git rebase`. Rebase and merge on GitHub will always update the committer information and create new commit SHAs, whereas `git rebase` outside of GitHub does not change the committer information when the rebase happens on top of an ancestor commit. For more information about `git rebase`, see git-rebase in the Git documentation.

To rebase and merge pull requests, you must have write permissions in the repository, and the repository must allow rebase merging.

For a visual representation of `git rebase`, see The "Git Branching - Rebasing" chapter from the *Pro Git* book.

Before enabling commit rebasing, consider these disadvantages:

- Repository contributors may have to rebase on the command line, resolve any conflicts, and force push their changes to the pull request's topic branch (or remote head branch) before they can use the **rebase and merge** option on your GitHub Enterprise Server instance. Force pushing must be done carefully so contributors don't overwrite work that others have based their work on. To learn more about when the **Rebase and merge** option is disabled on your GitHub Enterprise Server instance and the workflow to re-enable it, see "About pull request merges."

- When using the **Rebase and Merge** option on a pull request, it's important to note that the commits in the head branch are added to the base branch without commit signature verification. When you use this option, GitHub creates a modified commit, using the data and content of the original commit. This means that GitHub didn't truly create this commit, and can't therefore sign it as a generic system user. GitHub doesn't have access to the committer's private signing keys, so it can't sign the commit on the user's behalf.

  A workaround for this is to rebase and merge locally, and then push the changes to the pull request's base branch.

For more information, see "Configuring commit rebasing for pull requests."