

Git is an open-source distributed version control system. It helps you manage your project files easily using local branching, staging, and workflows.

Many developers use Git today. And they're usually familiar with basic Git concepts like:

* How to initiate a repository.
* How to create branches.
* How to stage/unstage changes.
* How to commit changes.
* How to push commits to remote.

However, many developers are confused about concepts like merging and resolving merge conflicts. In this article, we will learn how to resolve merge conflicts in a practical way. This means you will read, understand, and try it out while going through this article.

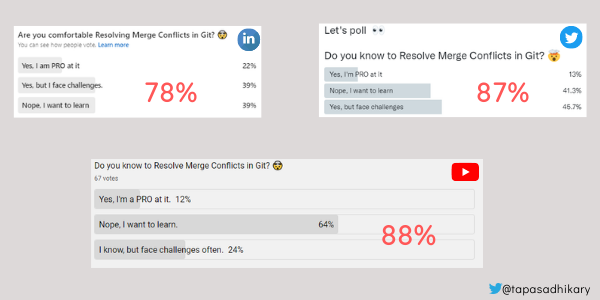
If you like to learn from video content as well, this article is also available as a video tutorial here: 🙂

If you are new to Git and want to learn all the basic concepts, [here is a helpful crash course](https://www.youtube.com/watch?v=vWtu4mzUgQo).

**What are Devs Saying about "Merge Conflicts"?**

Recently I conducted a poll on Twitter, LinkedIn, and YouTube, asking if developers are comfortable with resolving merge conflicts in Git. Guess what I found?

70%-80% of developers shared that they find it challenging to resolve a merge conflict in Git. So this means that "Resolving Merge Conflicts" is an important topic of discussion.

Poll Results - Are you comfortable resolving merge conflicts in Git?

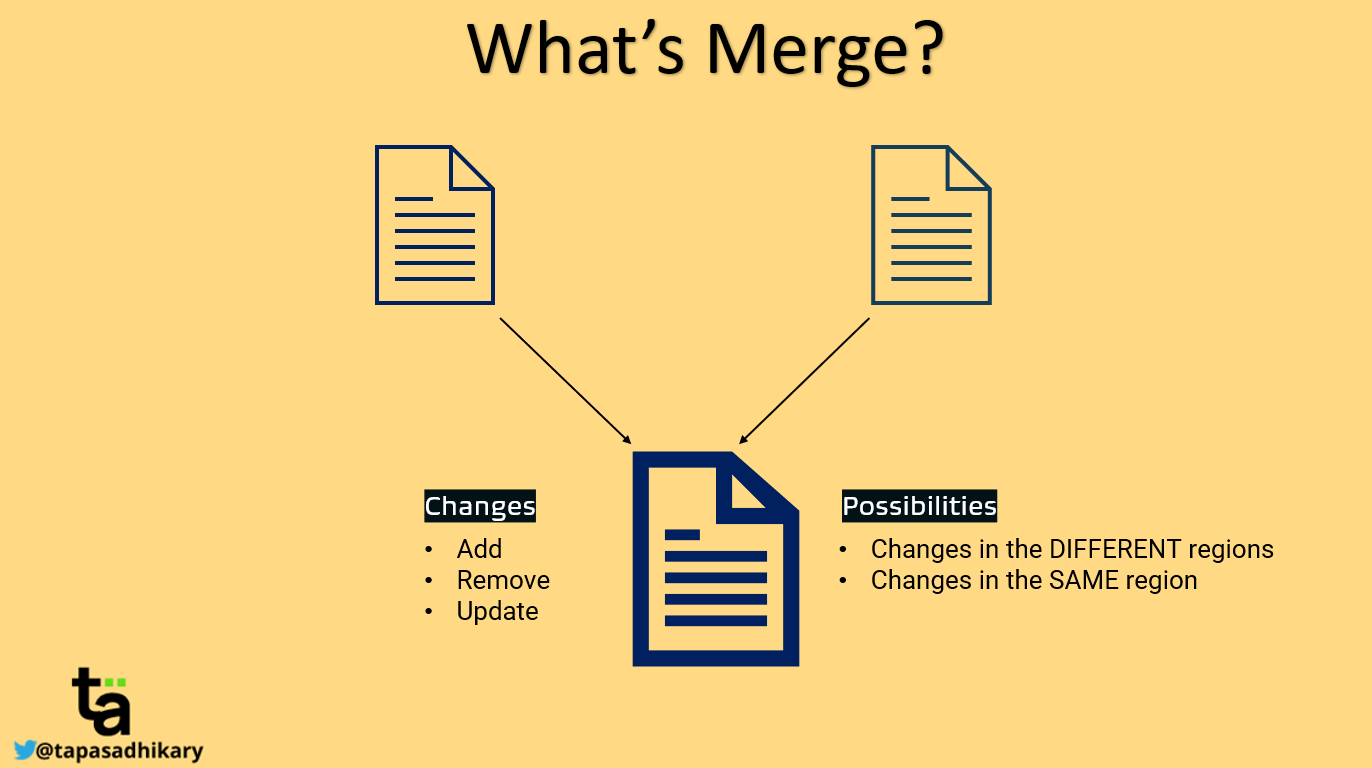
**What is Git Merge and What are Merge Conflicts?**

Git is a version control system that keeps a history of all your file versions. You can go back to any of the versions at any time and retrieve an older version.

Suppose you have created a file called abc.txt and pushed it to a Git repository. At this point, the file has its current version associated with it. Now, if your co-worker changed the same file and pushed it back to the repository, the file has a new version associated.

Git Merge is a feature that allows you to keep the file's current content in sync with other previous versions. This is essential because anyone at any point in time should be working on the most recent content of the file without overriding any changes from the previous versions.

Git merge helps you merge changes from other developers before pushing a new change to the same file.



In the case of Git merge, we need to be aware of two things:

1. **Changes**: What type of operations occurred between two versions of a file? New content is added or removed, or existing content is updated.
2. **Possibilities**: There are two possibilities. The changes happened in the different regions of the file or the changes happened in the same region of the file. Same region means that developers have made changes around the same place (for example, paragraphs, lines, and so on) of a file.

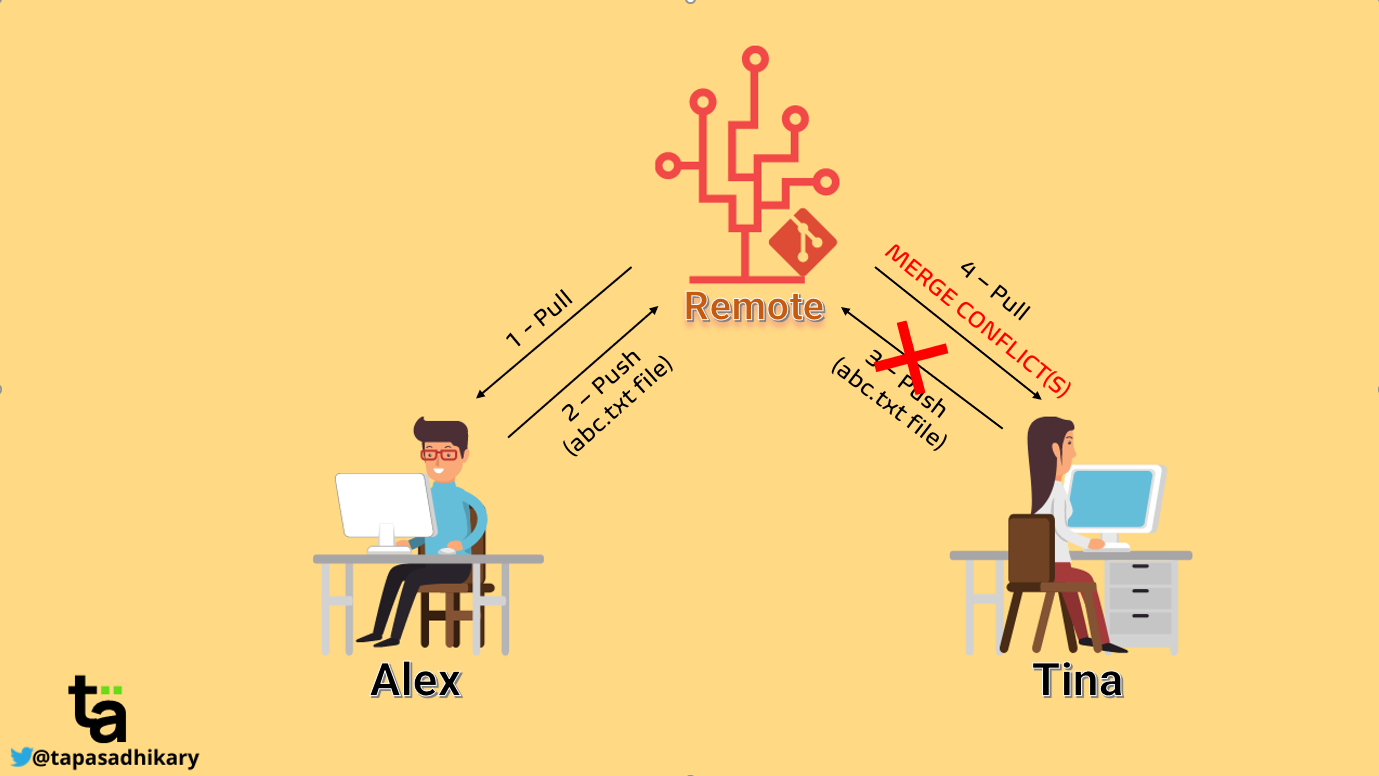
Fortunately, Git automatically takes care of most of these cases using the auto-merge strategy. But when the changes have occurred in the same region of the file, Git won't perform an auto-merge. Instead, it leaves it to you to Resolve the Merge Conflicts.

**Git Merge Conflicts: A Horror Story**

Let's understand the above situations with a story of two developers, Alex and Tina.

One fine day,

* Alex pulled changes from the remote repository to his local repository.
* He changed the file called abc.txt, staged it, committed it, and finally pushed it back to the remote repository.
* In the meantime, Tina, unaware of Alex's changes in the abc.txt file, made some changes in the same region of the file and tried pushing it to the remote repository.
* Git is a version control system, so it warned Tina that she had changed the version older than what it was in the remote (as Alex's changes were already in the remote).
* Now, Tina needs to first pull the changes from the remote, update the file, and then try pushing again.
* Tina did this. However, in her wildest nightmare, she got the warning that auto-merge failed, and so she needs to now Resolve the merge conflicts.



Does this story ring any bells? Is the above story related to you? There's a chance that you've been in Tina's shoes in the past. If not, you will get there eventually! So, let's understand how Tina has to deal with this situation efficiently.

**How to Resolve Merge Conflicts in Git**

Resolving merge conflicts is not as tricky as it may sound. In 90% of cases, it is easier once you have a clear understanding of the changes and a peaceful mind.

**Thought Process**

Once Tina pulls the changes, Tina's local file has her changes plus Alex's changes. Now Tina can do one of these four things:

* She can keep Alex's changes and remove hers.
* She can remove Alex's changes and keep hers.
* She can keep both Alex's and her changes.
* She can remove both Alex's and her changes.

Alright, but which one she should be doing? That is entirely dependent on the project's needs and the use-cases. Tina will understand the incoming changes and do whatever is relevant to the situation.

So, what are incoming changes? How's Tina going to identify that? How does Tina make the changes? I know you have got many such questions. Let's get the answers to all of them by taking a couple of real-life examples in the section below.

**Steps to Resolve Merge Conflicts in Git**

Let's take a couple of real-life examples of merge conflicts, and learn how to resolve them.

At any point in time, if you want to learn these concepts interactively, please check out [this section of the video](https://www.youtube.com/watch?v=OulZeVtZhZQ&t=397s) I have mentioned at the beginning of this article.

**Example 1: Changes are in the Same Region of the File**

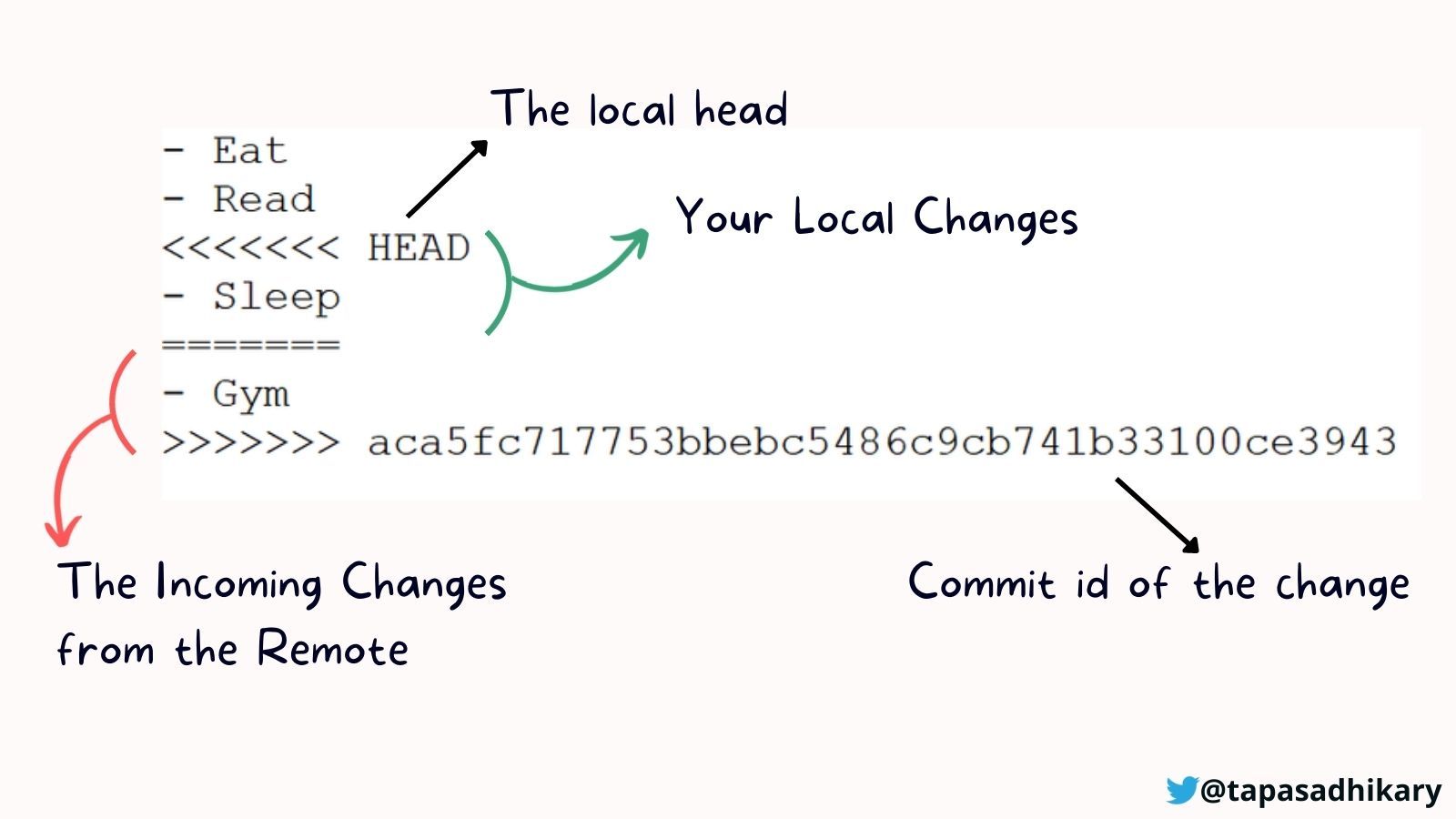
When Git cannot perform an auto-merge because changes are in the same region, it indicates the conflicting regions with special characters. The character sequences are like this:

* <<<<<<<
* =======
* >>>>>>>

Everything between <<<<<<< and ======= are your local changes. These changes are not in the remote repository yet. All the lines between ======= and >>>>>>> are the changes from the remote repository or another branch. Now you need to look into these two sections and make a decision.

The image below shows the content of a file indicating that auto-merge didn't occur and there is a conflict. The conflict is in the line where we have modified the file locally by adding a line - Sleep. But in the meantime, someone else pushed a change by adding the line - Gym in the same region.

So, the line - Sleep is marked as the local change and - Gym as the incoming changes from the remote repository or another branch.

Merge Conflict due to Changes in the Same Region

Based on your use case and project needs, you will make the call to resolve the conflict. If you need to keep only the line with - Sleep, you will keep that and remove the rest of the conflicting texts. In that case, the file content becomes:

- Eat

- Read

- Sleep

On the contrary, you can keep the line - Gym and remove the - Sleep changes:

- Eat

- Read

- Gym

If you need to keep both lines, remove the lines related to the conflict indicators:

- Eat

- Read

- Sleep

- Gym

If you think none of the changes are required, remove them all.

- Eat

- Read

It is entirely up to you to decide what changes are relevant to the situation. After your changes, you need to make sure that none of the conflict-indicating characters exist (<<<<<<<,  =======, >>>>>>>) in the file. Once you settle with the changes, do the following:

Stage the changes:

git add <files>

Commit the changes with a message:

git commit -m "Message"

Finally, push the changes to the remote:

git push

That's all there is to it to resolve the merge conflict in this scenario.

**Example 2: The File is Removed at the Remote/Other Branch**

In removed file merge conflicts, a dev deletes a file in one branch while another dev edits the same file in another branch. In this case, you need to decide if you want to keep the file or if it was right to delete it.

To add the deleted file back to your branch, do this:

git add <file-name>

To go ahead with the deletion of the file, do this:

git rm <file-name>

Then commit your changes with a message:

git commit -m "Message"

Finally, push it:

git push