# 1. Fast-forward merge

Wednesday, July 2, 2025 7:13 PM

# 2. Three-Way Merge

Wednesday, July 2, 2025 7:14 PM

Let's **deeply break down** the internal process of a **Three-Way Merge (No Fast-Forward)** using Git, step by step.

## 🧩 Situation Setup (Before the Merge)

We have two branches: master and feature.

scss
نسختحرير
```
A---B---C (master)
     \
      D---E (feature)
```

- **A, B, C** are commits on master.
- **D, E** are commits on feature.
- Both master and feature have diverged after commit B.
- C is a child of B on master, and E is a child of D on feature.

## 🧠 Internally: What Git Does in a Three-Way Merge

Git uses the **three-way merge algorithm** to merge the branches.

### 🧪 1. Find Common Ancestor

Git first finds the **common ancestor** of the two branches being merged.

- git merge will be run on master to merge feature.
- Git compares:
  - **Base commit**: B (common ancestor)
  - **Head1**: C (current master)
  - **Head2**: E (latest on feature)

### 🧬 2. Calculate Changes

Git does a diff:

- diff(B, C) → what changed from base to master.
- diff(B, E) → what changed from base to feature.

These are the **two sets of changes** Git now tries to **combine**.

### ⚔️ 3. Merge Conflicts (if any)

If both C and E modified the **same lines** in the **same files**, you get a **conflict**.
If the changes are in **different areas**, Git merges automatically.

### 🧱 4. Create Merge Commit (M)

Git creates a new commit **M**, which:

- Has **two parents**: C and E.
- Combines changes from both branches.
- Preserves history from **both branches** (unlike fast-forward).

## ✅ Final History After Merge

css
نسختحرير
```
A---B---C-------M (master)
```

```
     \          /
      D---E---/ (feature)
```

- The master branch is now at M.
- The feature branch still ends at E (unchanged).
- Commit M contains combined code from both master and feature.

## 📦 Real Git Commands

**Step-by-step Demo:**

bash
نسختحرير
```
# Start from master
$ git checkout master
# Create a feature branch and add changes
$ git checkout -b feature
# edit file, commit D
$ git commit -m "commit D"
# edit again, commit E
$ git commit -m "commit E"
# Go back to master and make separate changes
$ git checkout master
# edit file, commit C
$ git commit -m "commit C"
# Merge feature into master with merge commit
$ git merge feature
```

- If there are no conflicts, Git will create merge commit M.
- If there are conflicts, Git will pause and ask you to resolve them, then:

bash
نسختحرير
```
$ git commit  # to complete the merge after resolving
```

## 🎯 Benefits of Three-Way Merge

| **Feature** | **Description** |
|---|---|
| 🧠 Maintains History | Both histories are preserved. You can track both sides. |
| 🧱 Clear Merge Points | Merge commits (M) clearly show integration points. |
| 🕵️ Easy to Debug | If bugs appear, you can trace whether they came from master or feature. |

## 🌐 مقارنة مع Fast-Forward

| **Fast-Forward** | **Three-Way Merge** |
|---|---|
| Linear history | Branching + merge commits |
| No merge commit | Creates a merge commit (M) |
| Simpler log | Full history, clear merges |

# 3. Squash Merge process

Wednesday, July 2, 2025 7:04 PM

Let's **demonstrate the Squash Merge process in deeper detail** using:

1. 🔧 Practical Git Commands
2. 📜 Full Example with Files
3. 🔁 Internal View at Each Step
4. 📊 Graph Before vs After

## ✅ Let's Create a Realistic Example

Suppose you're working on a login feature:

### 📁 Initial Project Files (on master)

```
project/
├── index.html
└── style.css
```

### 💡 Goal:

You create a feature/login branch and do **3 commits**:

1. Add login.html
2. Add login.js
3. Add validation code in login.js

You want to **squash all 3 commits into one**, and apply that one to master.

## 🔧 Step-by-Step Commands

### 🔷 1. Initialize a Git Repo

```
mkdir squash-demo && cd squash-demo
git init
```

Create index.html:

```
echo "<h1>Home Page</h1>" > index.html
git add index.html
git commit -m "Initial homepage"
```

### 🔷 2. Create a Feature Branch

```
git checkout -b feature/login
```

**First Commit (D)**

```
echo "<form>Login</form>" > login.html
git add login.html
git commit -m "Add login page"
```

**Second Commit (E)**

```
echo "console.log('login');" > login.js
git add login.js
git commit -m "Add login JS"
```

**Third Commit (F)**

```
echo "function validate() { return true; }" >> login.js
git add login.js
git commit -m "Add validation to login"
```

Now the history is:

```
A---B (master)
    \
     D---E---F (feature/login)
```

## 🔁 Now Switch to master and Squash Merge

```
git checkout master
git merge --squash feature/login
```

**Output:**
Git applies all changes from commits D, E, F but does not create a commit.
You now write a new commit message:

```
git commit -m "Add login feature: HTML, JS, validation"
```

Now the commit graph looks like:

```
A---B---S (master)
     \
      D---E---F (feature/login)
```

- S is the **squash commit** (synthesized).
- Git doesn’t remember the squash came from a branch.
- The feature/login branch still exists separately.

## 🧠 Internally, What Happened

1. Git identified the **merge base** (B)
2. Git computed the **diff from B → F**
3. Git **applied that diff** to master (B → S)
4. Git **staged the result**, waiting for your commit
5. You created a **new commit S** with your message

## 💥 What if You Had Merge Conflicts?

If changes in the feature branch touch the **same files or lines** as master, Git will pause and show a conflict.
But in squash merge:

- You fix the conflict manually
- Stage the fixed files
- Then run: git commit

No special "merge commit" will be made.

## 📊 Full Visual (Before & After)

### 🔷 Before:

```
master: A---B
             \
feature:      D---E---F
```

### 🔷 After Squash Merge:

```
less
نسختحرير
master: A---B---S
                 \
feature:          D---E---F
```

- No merge commit.
- Just one clean commit on master.
- Feature branch remains unchanged.

## 🔍 Git Log Comparison

### After Three-Way Merge:

```
git log --oneline --graph
*   f1d2f3 Merge branch 'feature/login'
|\
| * 48fdf3 Add validation
| * e3a5c9 Add login JS
| * d12f01 Add login.html
* 34a8d9 Initial homepage
```

### After Squash Merge:

```
git log --oneline --graph
* b212f3 Add login feature: HTML, JS, validation
* 34a8d9 Initial homepage
```

## 🔚 Summary (Internal + External)

| Action | Description |
|---|---|
| Internally uses diff(B→F) | Yes |
| Merge base used | Yes (common ancestor) |
| Multiple commits preserved | ❌ No |
| One commit created | ✅ Yes (your message) |
| Branch graph changes | ❌ No merge link added |
| History stays linear | ✅ Yes (perfect for clean history) |

## If exist diverged ???

Let's now **demonstrate what happens when doing a squash merge in a *diverged* situation**, and how Git handles that **internally**.

## ✅ Scenario: Squash Merge When Branches Have Diverged

### 🌐 Starting Graph (Before Squash Merge):

scss
نسختحرير

```
A---B---C (master)
     \
      D---E---F (feature)
```

- master has C
- feature has F
- They both diverged after commit B

Now you try to squash feature into master.

## 🔍 Does Squash Merge Work with Diverged Branches?

✅ Yes, **squash merge still works** when branches have diverged.
Unlike a regular merge, Git does **not care** that the branches diverged — it just **applies the diff** from base to feature on top of the current branch.

## 🧠 Internal Process of Squash Merge with Divergence

Let's explain what happens in your case.

### 📍 You're at:

```
$ git checkout master
$ git merge --squash feature
```

Git does this:

### 🔬 Step 1: Find the Common Ancestor

- Common ancestor = B

### 🧪 Step 2: Compute Diff from B → F (feature)

- Git gathers **all changes made from D → F**
- Even if master added its own commits (e.g., C), that doesn't stop squash merge.

### ⚙️ Step 3: Apply That Diff on Top of C

- It applies the **combined changes** from feature *on top of* C
- Like you copied all new/changed files from feature and applied them to master's latest version

## 💥 Step 4: Possible Conflicts?

- If master and feature **touched the same lines or files**, you may get a **conflict**
- Resolve conflicts → git add the resolved files → then git commit

## 🧱 Step 5: Commit as One

After resolving conflicts (or if none), you run:

bash
نسختحرير
```
git commit -m "Add login feature (squashed)"
```
Final graph:

scss
نسختحرير
```
A---B---C---S (master)
     \
      D---E---F (feature)
```

## 🧪 Example in Real Commands

### Step-by-Step:

```
# Start from scratch
mkdir squash-diverged && cd squash-diverged
git init
echo "v1" > file.txt
git add file.txt
git commit -m "A: Add file.txt"
echo "v2" > file.txt
git commit -am "B: Update file to v2"
# Create feature branch
git checkout -b feature
echo "feature v3" > file.txt
git commit -am "D: Add feature v3"
echo "feature v4" >> file.txt
git commit -am "E: Append feature v4"
# Go back to master and diverge
git checkout master
echo "master v3" >> file.txt
git commit -am "C: Append master v3"
```
Now master has C, and feature has E.

### Now Do Squash Merge:

bash
نسختحرير
```
git merge --squash feature
```
If no conflicts → Git stages the combined diff.
If conflict appears (likely here since both changed file.txt), Git will say:

bash
نسختحرير
```
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
```
Fix the file manually, then:

bash
نسختحرير
```
git add file.txt
git commit -m "Squashed feature branch with login changes"
```

## 🧠 Internal Summary (When Diverged)

| Internal Step | Explanation |
|---|---|
| Base Commit | Git finds the **common ancestor** (e.g., B) |

| | |
|---|---|
| Target of Merge | Current HEAD (e.g., C on master) |
| Applied Diff | Changes from B → F |
| Commit Created | ✅ One manual commit (you write the message) |
| Merge Commit? | ❌ No (no merge trace in history) |
| Conflicts Possible? | ✅ Yes, if overlapping changes |

## 🧠 BONUS: Why This is Useful

- You keep your **master history clean** (just one commit).
- You can **rebase** or delete the feature branch easily.
- You don't bring over "messy WIP commits" from the feature branch.

# 4. Octopus Merge

Wednesday, July 2, 2025 7:33 PM

## 📍 What is an Octopus Merge?

An **Octopus Merge** is when you merge **more than two branches** into one **in a single command**. Git internally supports this as long as there are **no conflicts** between them.

## 🌐 Initial Structure (Before Merge):

```
      D---E (feature1)
     /
A---B (master)
     \
      F---G (feature2)
```

- feature1 branch has commits D and E
- feature2 has commits F and G
- Both branches diverged from common ancestor B

## 🔍 What Git Sees:

You're trying to merge both feature1 and feature2 into master in **one shot**:

```
git checkout master
git merge feature1 feature2
```

## 🧠 Internally: How Octopus Merge Works

### 🔬 Step 1: Find the Common Ancestor

Git finds:

- Common ancestor = B

### 🧪 Step 2: Validate Clean Merge

Git:

- Tries to combine changes from:
  - B → E (feature1)
  - B → G (feature2)
- Checks if **any conflicts** exist between the two changesets.

  ⚠️ **Important Rule**:
  If Git finds **any conflicts**, it will **abort** the merge.
  Octopus merge is **only for conflict-free merges.**

### 🧱 Step 3: Create a Single Merge Commit M

If successful, Git creates **one merge commit M**:

```
       D---E
      /       \
A---B----------M (master)
     \         /
       F---G
```

- M has **multiple parents**:
  - First parent: B (master base)
  - Others: E, G

This is the "octopus" – **one head with many arms** 🐙

## 🔧 Real Command Example

### Step 1: Setup

```
git init octopus-merge-demo
cd octopus-merge-demo
echo "start" > file.txt
git add file.txt
git commit -m "Initial commit (A)"
echo "line from B" >> file.txt
git commit -am "B: master continues"
```

### Step 2: Create Branches

**Feature 1:**

```
git checkout -b feature1
echo "feature1: add login" > login.txt
git add login.txt
git commit -m "D: Add login.txt"
echo "feature1: update login" >> login.txt
git commit -am "E: Update login.txt"
```

**Back to master → Create Feature2:**

```
git checkout master
git checkout -b feature2
echo "feature2: add dashboard" > dashboard.txt
git add dashboard.txt
git commit -m "F: Add dashboard.txt"
echo "feature2: update dashboard" >> dashboard.txt
git commit -am "G: Update dashboard.txt"
```

### 🔁 Step 3: Perform Octopus Merge

```
git checkout master
git merge feature1 feature2
```

✅ If there are **no file overlaps**, Git creates a merge commit like:

```
Merge branches 'feature1' and 'feature2'
```

## 🔍 Git Log After Merge

```
git log --oneline --graph --all
```

Shows something like:

sql
نسختحرير

```
*   abc123 Merge branches 'feature1' and 'feature2'
```

```
|\
| * g7g8g9 G: Update dashboard.txt
| * f5f6f7 F: Add dashboard.txt
* | e3e4e5 E: Update login.txt
* | d1d2d3 D: Add login.txt
|/
* b1b2b3 B: master continues
* a0a1a2 Initial commit (A)
```

## ⚠️ What If Conflicts Exist?

Let's say:

- Both feature1 and feature2 **edit the same file** (e.g., file.txt)

Then this will happen:

bash
نسختحرير

```
$ git merge feature1 feature2
error: Merging is not possible because you have unmerged files.
fatal: merge failed
```

❌ Git **aborts the octopus merge** — unlike regular merge, **you can't resolve conflicts manually** here.

## ✅ When Should You Use Octopus Merge?

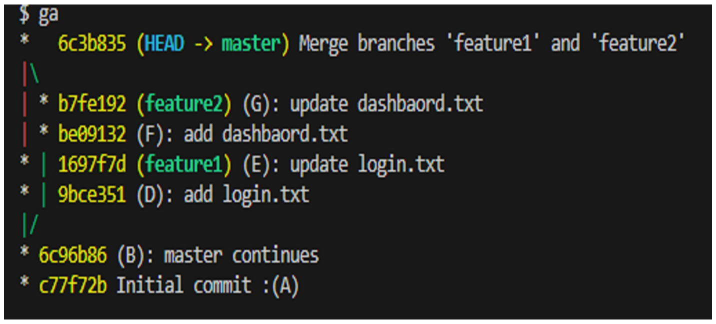
| **Use Case** | **Suitability** |
|---|---|
| Merging multiple branches at once | ✅ YES |
| No conflicts between the branches | ✅ YES |
| Clean feature integration (e.g., plugins) | ✅ YES |
| Complex, conflicting branches | ❌ NO |

## 📚 Summary (Internals + Output)

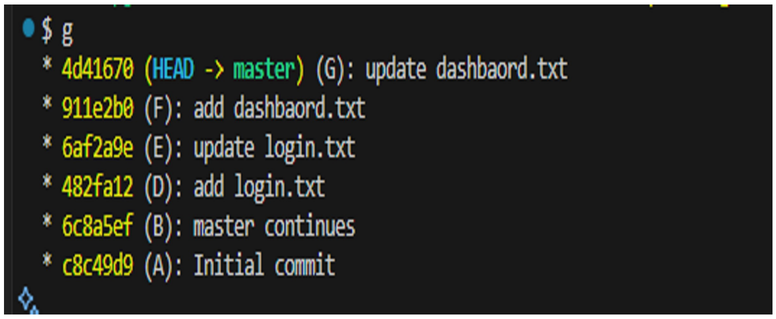
| **Step** | **What Happens** |
|---|---|
| Common ancestor | Git finds B |
| Check for conflicts | Git checks all incoming branches |
| Apply diffs | All branches diffed from B |
| Merge commit | Git creates M with multiple parents |
| Conflict present? | ❌ Merge is aborted (can't resolve manually) |

# 4.1 change the message of the initial/root commit

Wednesday, July 2, 2025 8:20 PM



```
$ ga
*   6c3b835 (HEAD -> master) Merge branches 'feature1' and 'feature2'
|\
| * b7fe192 (feature2) (G): update dashbaord.txt
| * be09132 (F): add dashbaord.txt
* | 1697f7d (feature1) (E): update login.txt
* | 9bce351 (D): add login.txt
|/
* 6c96b86 (B): master continues
* c77f72b Initial commit :(A)
```



```
$ g
* 4d41670 (HEAD -> master) (G): update dashbaord.txt
* 911e2b0 (F): add dashbaord.txt
* 6af2a9e (E): update login.txt
* 482fa12 (D): add login.txt
* 6c8a5ef (B): master continues
* c8c49d9 (A): Initial commit
```

## 🟠 Question:

How can I change the message of the **initial/root commit**:

c77f72b Initial commit :(A)
to:

(A): Initial commit

## 📌 Git History Before Rewriting

Here's the state of your repository **before adjusting the message**:

```
*   6c3b835 (HEAD -> master) Merge branches 'feature1' and 'feature2'
|\
| * b7fe192 (feature2) (G): update dashboard.txt
| * be09132 (F): add dashboard.txt
* | 1697f7d (feature1) (E): update login.txt
* | 9bce351 (D): add login.txt
|/
* 6c96b86 (B): master continues
* c77f72b Initial commit :(A)   👈 You want to change this
```

## 🛠️ Solution: Using git rebase --root

To rewrite the first commit message:

### 🔷 Step 1: Run Root Rebase

```
git rebase -i --root
```

### 🔷 Step 2: Change pick to reword

Editor opens with:

```
pick c77f72b Initial commit :(A)
pick 6c96b86 master continues
pick 9bce351 add login.txt
pick 1697f7d update login.txt
pick be09132 add dashboard.txt
pick b7fe192 update dashboard.txt
pick 6c3b835 Merge branches 'feature1' and 'feature2'
```

✅ Modify the first line:

```
reword c77f72b Initial commit :(A)
```
💾 Then save and exit the editor.

## 🔷 Step 3: Git Prompts for New Message

A second editor opens showing:

```
Initial commit :(A)
```
✏️ Change it to:

```
(A): Initial commit
```

## 💾 Save and Exit

- **Nano**:
  - Ctrl + O, Enter, then Ctrl + X
- **Vim**:
  - Esc, type :wq, then Enter

## ✅ Git History After Rewriting

Your log will now show the new message:

Your actual commit log will look like this:

```
* 83a2f0a (HEAD -> master) (G): update dashboard.txt
* b50dd6e (F): add dashboard.txt
* 1697f7d (feature1) (E): update login.txt
* 9bce351 (D): add login.txt
* 6c96b86 (B): master continues
* abcd123 (A): Initial commit     ✅ message updated!
```
🧠 **Note**: The commit hash of the initial commit (1a2b3c4) changed because commit content was modified.

## ⚠️ Important Notice

If you already pushed your repo to a remote (GitHub, etc.), you'll need to **force-push**:

```
git push --force
```

# 5. Rebase then Merge – Internal Process

Wednesday, July 2, 2025 9:49 PM

✅ **Rebase Then Merge – Internal Process**
...when the feature branch was created after commit B (not after C)

## 🧱 Situation Overview

You have two branches:

- master: continues from B → C
- feature: branched from B, adds D and E

## 🧭 Git History Before Rebase

```
A---B---C (master)
    \
     D---E (feature)
```

This means:

- feature branched **before** the latest commit on master (C)
- So Git considers the two branches **diverged**

## 🎯 Goal:

You want to make the feature branch look as if it was developed **after C**, not B.
This creates a **linear history** and avoids a merge commit.

## 🛠️ Step-by-Step Process

### 🔷 Step 1: Rebase feature onto master

bash
نسختحرير

```
git checkout feature
git rebase master
```

### 🧠 What Git Does Internally:

1. **Finds the base** of feature = commit B
2. **Takes all new commits** after B in feature → D and E
3. **Copies** them
4. **Reapplies** them on top of C

🧪 The new commits are called:

- D': new version of D
- E': new version of E

## 📈 Git History After Rebase

mathematica
نسختحرير

```
A---B---C---D'---E' (feature)
    \
    D---E        ❌ abandoned (old path)
```

- feature is now cleanly on top of master
- D and E are discarded (not deleted, just orphaned)

## 🔷 Step 2: Merge Feature (Fast-Forward)

```
git checkout master
git merge feature
```

- Since master ends at C, and feature starts at C and continues → **Git fast-forwards** master.

## ✅ Final Git History (After Rebase + Merge)

```
A---B---C---D'---E' (master, feature)
```

- No merge commit created
- Clean, linear history
- Looks like all work happened in one straight line

## 📦 Summary Table

| **Step** | **Action** | **Result** |
|---|---|---|
| git rebase | Rewrites feature onto master | Makes commits D, E become D', E' |
| git merge | Fast-forwards master | No merge commit |
| Final history | Linear (C → D' → E') | Easier to understand |

## ✅ Why This Is Useful

| **Benefit** | **Why It Helps** |
|---|---|
| No merge commit | Simpler history |
| Linear commit flow | Easy to read with git log --oneline |
| Avoids diverging branches | Keeps your project tidy |
| Great for feature branches | Especially before pushing to remote |

# 6. Merge with Conflict – Internal Process

Thursday, July 3, 2025 12:26 AM

✅ **6. Merge with Conflict – Internal Process**
Illustrated with commit graphs, internal steps, and terminal behavior.

## 🌐 Scenario: Merge with Conflict

## 🧱 Initial Commit Graph (Before Merge)

```
A---B---C (master)
     \
      D---E (feature)
```

- master has new commit C
- feature has new commits D → E
- Both **modified the same line** in the same file (let's say hello.txt)

## 🔍 What Git Sees:

- Common base commit = B
- It tries a **three-way merge**:
  - Base: B
  - Head1: C (from master)
  - Head2: E (from feature)
- It detects a **conflict**:
  ➤ C and E both changed the **same line** in hello.txt

## 🧠 Internal Git Process:

1. Git **stops the merge**
2. Adds **conflict markers** inside the conflicting file:

   ```
   <<<<<<< HEAD
   version from master (C)
   =======
   version from feature (E)
   >>>>>>> feature
   ```
3. Git **marks the merge as incomplete**
4. You must **manually resolve** the conflict

## 🧑‍💻 Your Role:

### 🔧 Step 1: Resolve the Conflict

- Open the file (e.g., hello.txt)
- Edit it to keep the correct version

Example resolved file:

✅ final version after resolving

## 🔧 Step 2: Mark as Resolved

```
git add hello.txt
```

## 🔧 Step 3: Complete the Merge

```
git commit -m "M: Merge feature into master (resolved conflict)"
```

## 📈 Final Commit Graph (After Merge)

```
A---B---C-------M (master)
     \         /
      D---E---/ (feature)
```

- M is a **merge commit**
- It has **two parents**: C (master) and E (feature)
- The merge only succeeds **after manual resolution**

## 📦 Summary Table

| Step | Action | Result |
|---|---|---|
| Git tries to merge | Uses common base B | Finds conflict between C and E |
| Conflict detected | Stops and adds conflict markers | Merge paused |
| Developer resolves | Edits file, runs git add | Marks as resolved |
| Developer commits | Runs git commit | Creates merge commit M |

## ⚠️ When This Happens

- Both branches edited **the same line**
- Git can’t decide which version to keep
- Very common in collaborative teams or long-lived branches

Git Rebase Commands

Thursday, July 3, 2025 10:23 AM

## ✅ 1. Basic Rebase Commands – Demonstrated

### 🔷 Example Setup

mkdir rebase-demo && cd rebase-demo
git init
Create base commits on master:

bash
نسختحرير
```
echo "v1" > app.txt
git add app.txt
git commit -m "A: Initial commit"
echo "v2" >> app.txt
git commit -am "B: Update v2"
echo "v3" >> app.txt
git commit -am "C: Update v3"
```
Create a new branch:

bash
نسختحرير
```
git checkout -b feature
echo "login 1" > login.txt
git add login.txt
git commit -m "D: Add login page"
echo "login 2" >> login.txt
git commit -am "E: Improve login form"
```
Now you have:

scss
نسختحرير
```
A---B---C (master)
   \
    D---E (feature)
```

### 🔷 1.1 git rebase master

bash
نسختحرير
```
git checkout feature
git rebase master
```
Git moves the feature branch commits D and E **after** C:

mathematica
نسختحرير

A---B---C---D'---E' (feature)

## 🔷 1.2 git rebase --abort

If a conflict happens:

bash
نسختحرير
```
git rebase master
# Conflict occurs!
git rebase --abort  # Restores branch back to before rebase
```

## 🔷 1.3 git rebase --continue

After resolving conflict:

bash
نسختحرير
```
# Fix file manually
git add file.txt
git rebase --continue
```

## 🔷 1.4 git rebase --skip

If a commit causes problems and you want to **ignore it**:

bash
نسختحرير
```
git rebase master
# Conflict in E
git rebase --skip  # E will be dropped
```

## ✅ First: git rebase --abort

### 📌 Purpose:

If a **conflict** happens during a rebase and you don't want to continue or you can't resolve it, this command allows you to **cancel the rebase** and go back to how things were before it started.

### 🧪 Example:

1. You have two branches:
   - master
   - feature (with some new changes)

bash
نسختحرير
```
# Switch to the feature branch
$ git checkout feature
# Start the rebase process
$ git rebase master
```

🎯 A conflict happens:

bash
نسختحرير
```
Auto-merging file.txt
```

CONFLICT (content): Merge conflict in file.txt

## 🛠️ To cancel the rebase and go back to the original state:

bash
نسختحرير
$ git rebase --abort
📌 This will cancel the rebase and return the feature branch to how it was before the rebase started.

## ✅ Second: git rebase --continue

### 📌 Purpose:

After you **manually resolve a conflict**, you use this command to continue the rebase process.

### 🧪 Steps:

1. During git rebase master, a conflict occurs in file.txt.

bash
نسختحرير
CONFLICT (content): Merge conflict in file.txt

1. You open the file and **resolve the conflict manually**, removing the conflict markers:

text
نسختحرير
<<<<<<< HEAD
// from master
=======
// from feature
>>>>>>> commit-hash

1. After resolving, stage the file:

bash
نسختحرير
$ git add file.txt

1. Then continue the rebase:

bash
نسختحرير
$ git rebase --continue
🔥 Git will proceed to the next commit and continue the rebase.

## ✅ Third: git rebase --skip

### 📌 Purpose:

If a certain commit causes issues and **is not important**, you can **skip** it and continue with the rest.

### 🧪 Example:

1. During rebase:

bash

نسختحرير

```
$ git rebase master
```

1. A conflict occurs in a commit that modifies E.txt.
2. Instead of fixing it, you choose to ignore that commit:

bash
نسختحرير

```
$ git rebase --skip
```

🔁 Git will skip that commit entirely and continue with the next ones.

## 🎨 Diagram (Before and After)

### 💡 Before Rebase:

mathematica
نسختحرير

```
master:   A---B---C
             \
feature:       D---E
```

### ✅ During git rebase master from feature:

Git tries to replay D and E on top of C:

mathematica
نسختحرير

```
A---B---C---D'---E'
```

If:

- A conflict happens in E
- You run git rebase --skip

📌 Result:

mathematica
نسختحرير

```
A---B---C---D'   ← E is removed
```

## 🧠 Summary:

| Command | Purpose | When to Use |
|---|---|---|
| git rebase --abort | Cancel the rebase | When a conflict happens and you don't want to continue |
| git rebase --continue | Resume after resolving a conflict | After fixing the conflict and staging the changes |
| git rebase --skip | Skip a problematic commit | If a commit causes issues and you decide to ignore it |

# Loacal not track remote

Tuesday, July 1, 2025 8:32 PM

Why doesn't Git say: “Your branch is ahead of 'origin/prolog-feature' by 1 commit”?
Even though prolog-feature has a commit (875ae94) that does **not** exist in origin/prolog-feature (2d6d08d).

## ✅ The Actual Reason (in detail):

You have the following:
* 875ae94 (HEAD -> prolog-feature) ← new local commit
* 2d6d08d (origin/prolog-feature) ← last known commit on the remote

So your local branch is **definitely ahead by one commit**.
But the question is:
**Why doesn't git status say: "Your branch is ahead of 'origin/prolog-feature' by 1 commit"?**

## 🔍 Possible Reasons:

### ✅ 1. Your local branch is not actually tracking the remote one

Even though the branch name is the same (prolog-feature), it might not be **linked** to origin/prolog-feature.
To check this, run:

```
git branch -vv
```

You should see something like:

```
prolog-feature 875ae94 [origin/prolog-feature] add hi.txt
```

If you **don’t see [origin/prolog-feature]**, that means the branch is **not tracking** the remote branch.

## ✅ Solution:

If your branch is not tracking the remote branch, run this:

```
git branch --set-upstream-to=origin/prolog-feature
```

Then check again with:

```
git status
```

Now you’ll see:

```
Your branch is ahead of 'origin/prolog-feature' by 1 commit.
```

## ✅ Quick Notes:

| Command | Purpose |
|---|---|
| git fetch | Updates your remote-tracking branches |
| git branch -vv | Shows whether your branch is tracking a remote |
| git status | Only compares to the **upstream** branch (if set) |

Would you like a visual diagram to help explain this case more clearly?

From <https://chatgpt.com/c/6863fe83-0050-8003-8583-58798d79b4d4>

From <https://chatgpt.com/c/6863fe83-0050-8003-8583-58798d79b4d4>