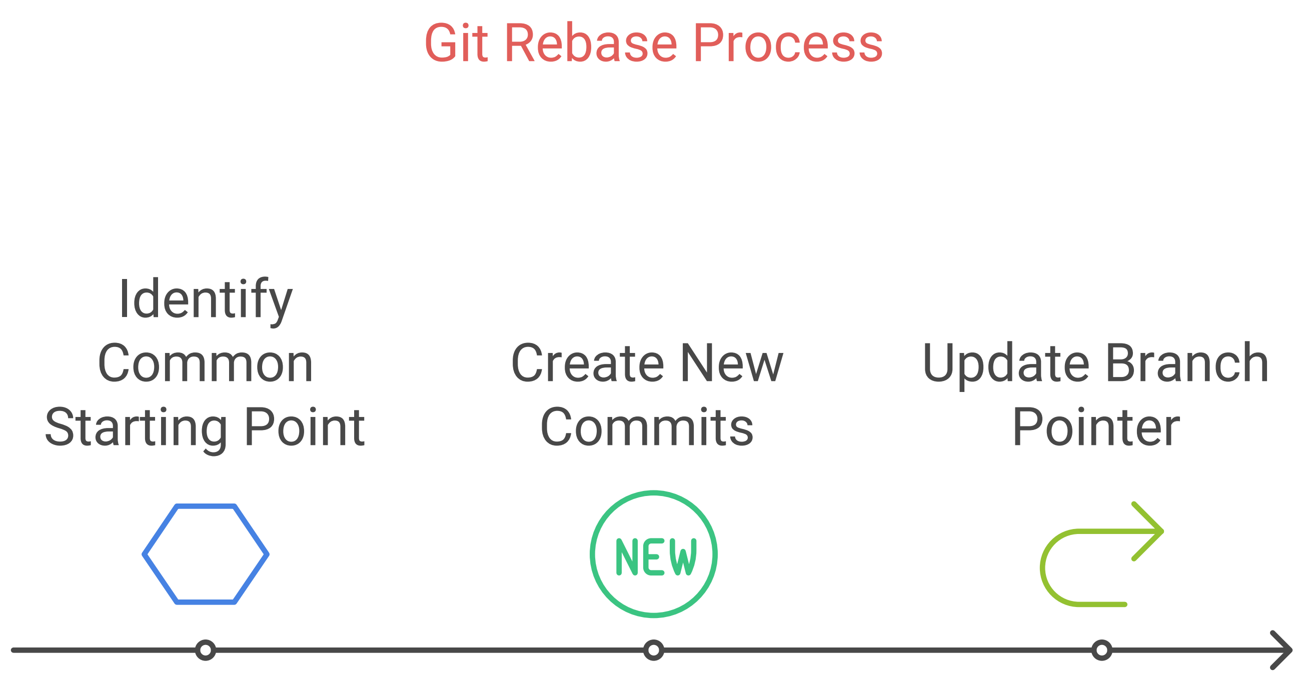
Understanding Git Rebase

What is Git Rebase?

Rebasing in Git rewrites commit history. It does not merge two branches like merging does. Merging creates a merge commit. Rebasing moves the commits from one branch to the top of another branch's HEAD.

How Does Git Rebase Work?

During a rebase, Git picks the changes from your branch and places them on another branch. Git follows a few steps to do this:
1. Git finds the common starting point of both branches.
2. It creates new commits. These commits hold the changes from your branch but start from the latest commit of the other branch.
3. Git updates your branch to point to these new commits.
This process shifts your branch forward.



We will try to use an example to clearly and visually explain git rebase by creating a simple diagram with text.

Scenario

1. You have two branches:
   - **main**: The primary branch.
   - **feature**: A branch created to work on a feature.
2. New commits are added to **main after** you branched off into **feature**.
3. You now want to rebase **feature** to include the latest changes from **main** and align the history.

Before Rebase:

```
main:
A --- B --- C (latest commit on main)

feature:
A --- B --- X --- Y (commits X and Y are your work on feature)
```

Here:

- Commits **A** and **B** are shared by both branches (common history).
- Commits **C** are new on **main**.
- Commits **X** and **Y** are unique to **feature**.

What Happens During Rebase?

1. Switch to **feature** branch: git checkout feature
2. Start rebasing onto **main:** git rebase main
3. **Rebase Process:** Git "replays" the commits from the **feature** branch (**X** and **Y**) **on top of the latest commit in **main** (**C**)

```
main:
A --- B --- C (unchanged)

feature (after rebase):
A --- B --- C --- X' --- Y'
```

After Rebase:

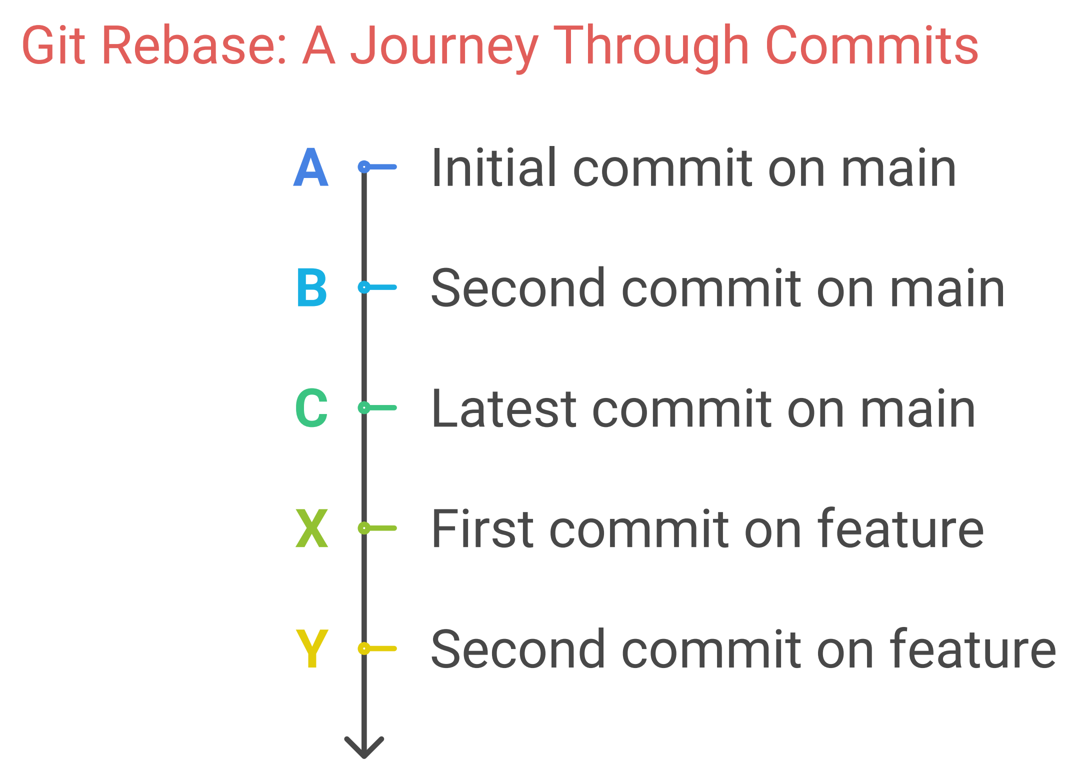
```
main:
A --- B --- C

feature:
            C --- X' --- Y'
```

Rebase with Conflicts

If there are **conflicts** during rebase (e.g., changes in **C** conflict with **X**), Git will stop and ask you to resolve them manually.

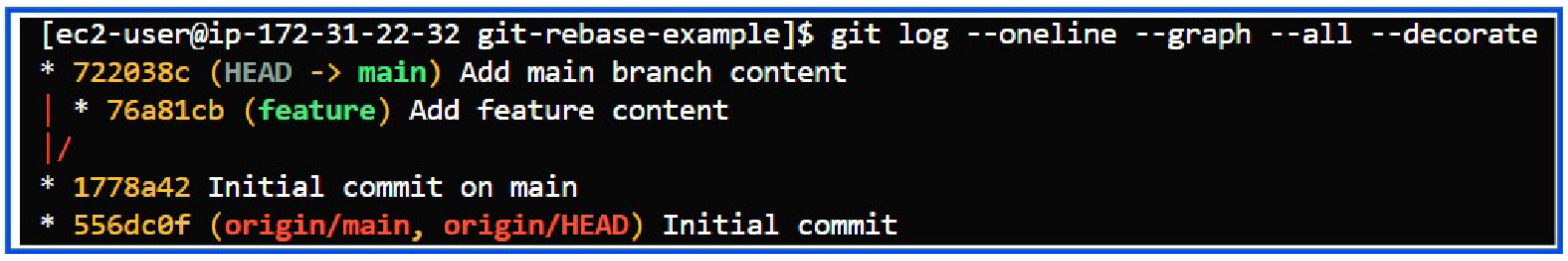
Summary Diagram:

```
Before Rebase:
main:    A --- B --- C
feature: A --- B --- X --- Y

Rebasing feature onto main:
Step 1: Detach X and Y from B.
Step 2: Attach X and Y after C.

After Rebase:
main:    A --- B --- C
feature:             C --- X' --- Y'
```


Git Rebase: A Journey Through Commits
A Initial commit on main
B Second commit on main
C Latest commit on main
X First commit on feature
Y Second commit on feature

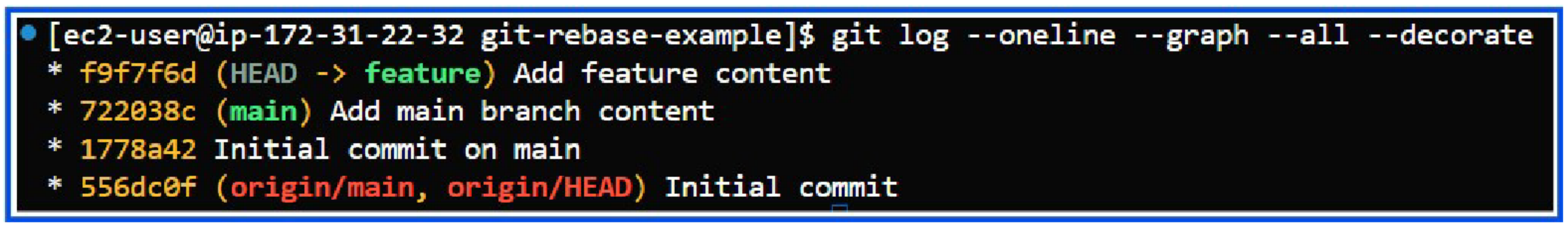
Practical Guide:

Before Rebase:



```
[ec2-user@ip-172-31-22-32 git-rebase-example]$ git log --oneline --graph --all --decorate
* 722038c (HEAD -> main) Add main branch content
| * 76a81cb (feature) Add feature content
|/
* 1778a42 Initial commit on main
* 556dc0f (origin/main, origin/HEAD) Initial commit
```

After Rebase:



```
[ec2-user@ip-172-31-22-32 git-rebase-example]$ git log --oneline --graph --all --decorate
* f9f7f6d (HEAD -> feature) Add feature content
* 722038c (main) Add main branch content
* 1778a42 Initial commit on main
* 556dc0f (origin/main, origin/HEAD) Initial commit
```

Note:

The commit ID for the feature branch changed after the rebase because Git rebase rewrites history. Git rebase modifies past commits. This process alters commit IDs in the branch. So, the commit ID is different in the feature branch. This is very important to remember.

Benefits of Using Git Rebase

- Simpler History: Rebase creates a straight commit history. This helps people understand the project's growth better. A clear history is easier to read.
- Easy Navigation: A straight history simplifies navigating through commits. Commands like git log and git bisect become simpler to use.
- No Merge Commits: Rebasing removes the messy merge commits in history. This keeps the commit log tidy.

Potential Drawbacks

- **Rewriting History**: Rebase changes commit history. This becomes a problem if you already sent commits to a shared repository. It may lead to confusion. Other collaborators might face conflicts.

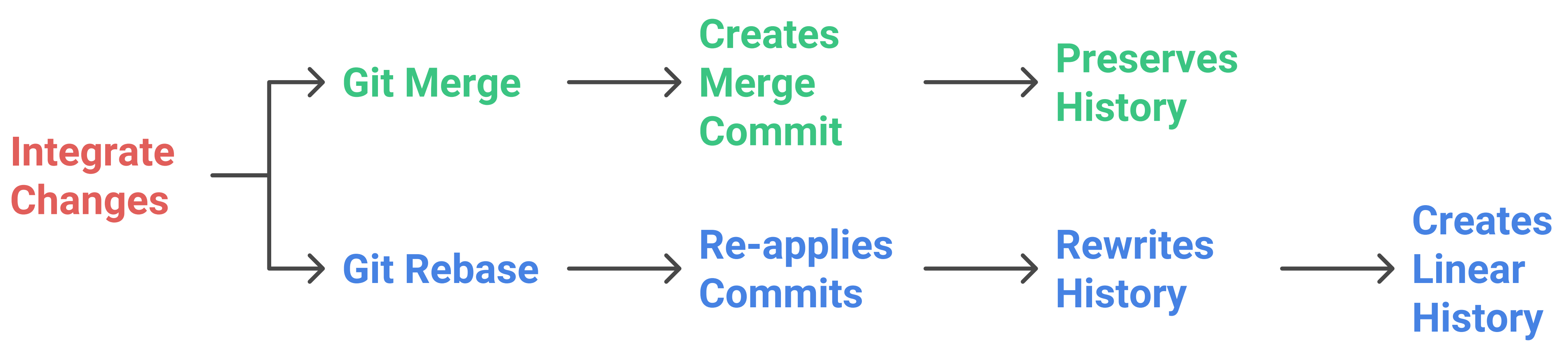
- **Complexity**: Beginners often find rebasing more complex than merging. Incorrect use may probably cause lost commits. It can also create complicated conflicts.

Best Practices for Using Rebase

1. Perform rebasing only on local or private branches.
2. Do not rebase branches already pushed to a shared repository unless it is needed.
3. Run git rebase -i (interactive rebase) to tidy up messy commit histories before
3. merging.
4. Always communicate with your team when using rebase in collaborative projects.

Git Merge vs Git Rebase:

1. **Git Merge**: Combines two branches. It creates a merge commit. This process keeps the history of both branches intact.
2. **Git Rebase**: Re-applies the commits of one branch on top of another. It rewrites history. It creates a clean and linear commit history.



Conclusion

Git rebase helps developers keep a clean and clear project history. This tool provides many benefits. However, using it with caution in team settings is very important. Properly understanding when and how to use rebase is crucial. It improves your version control workflow.

> Visit *subbutechops.com* to explore the exciting world of technology and data. Get ready to discover a lot more.

Subbu Tech Ops

LEARN. APPLY. MASTER