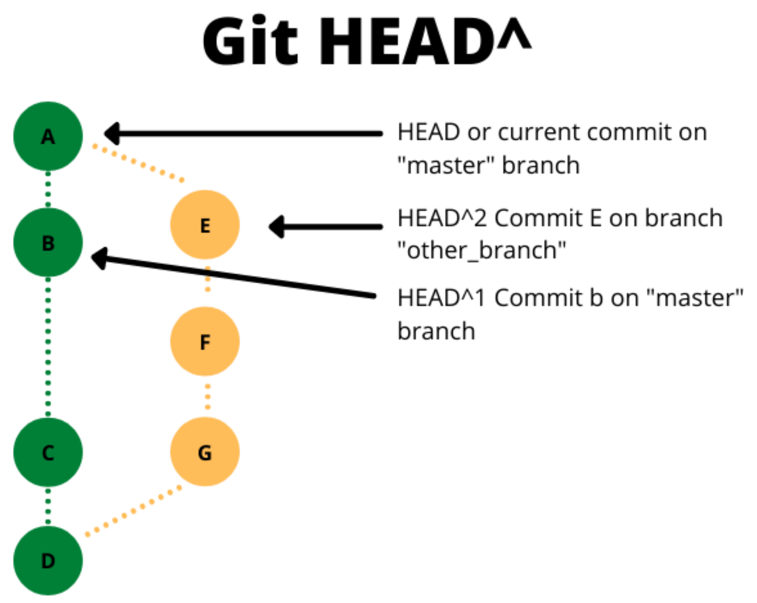
Git – Developers' Guide

Git Best Practices

Visualizing Git

https://git-school.github.io/visualizing-git/

GIT HEAD


Git HEAD^
A
B
C
D
E
F
G
HEAD or current commit on "master" branch
HEAD^2 Commit E on branch "other_branch"
HEAD^1 Commit b on "master" branch

HEAD is normally referred to "current branch". When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch.

You can see what HEAD points to by doing:

```
cat .git/HEAD
```

OR,

```
git symbolic-ref HEAD
```

OR,

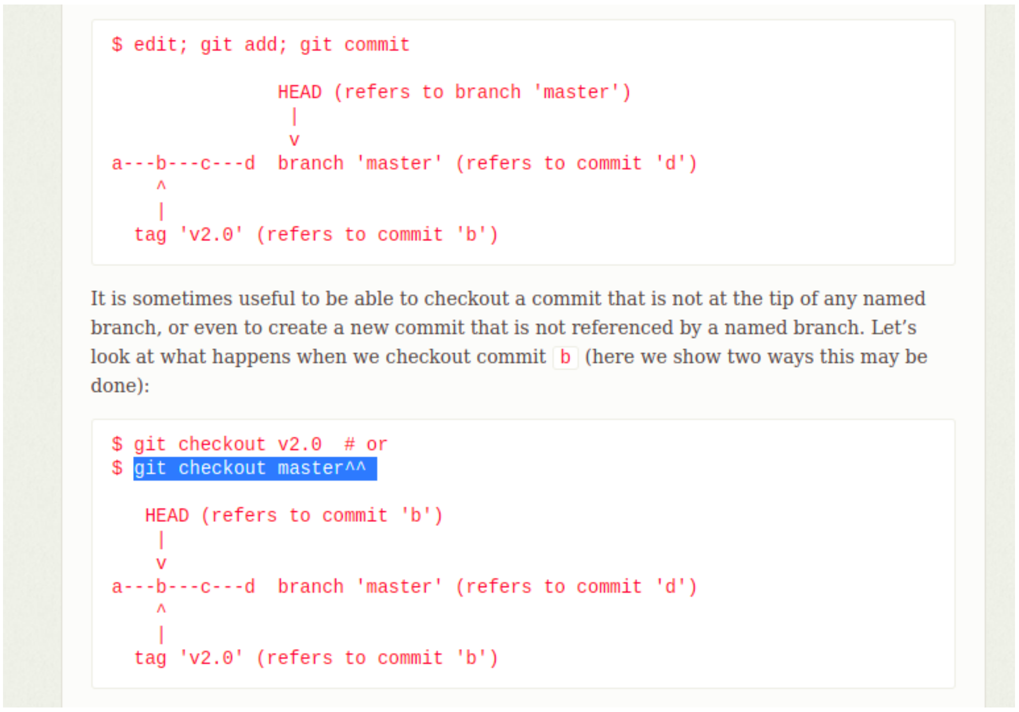
You can also see HEAD pointing to certain branch or commit when you :

```
git log
git log -oneline
```

Detached HEAD:

It is possible for HEAD to refer to a specific revision that is not associated with a branch name. This situation is called a detached HEAD.

Detached HEAD is when we detach the HEAD from the current branch and instead attach it to a certain commit. We can use commit number or <branch_name>^



```
$ edit; git add; git commit

               HEAD (refers to branch 'master')
                |
                v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

It is sometimes useful to be able to checkout a commit that is not at the tip of any named branch, or even to create a new commit that is not referenced by a named branch. Let's look at what happens when we checkout commit `b` (here we show two ways this may be done):

```
$ git checkout v2.0  # or
$ git checkout master^^

   HEAD (refers to commit 'b')
    |
    v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

Here, default HEAD was pointed at the latest commit of branch 'master'. After

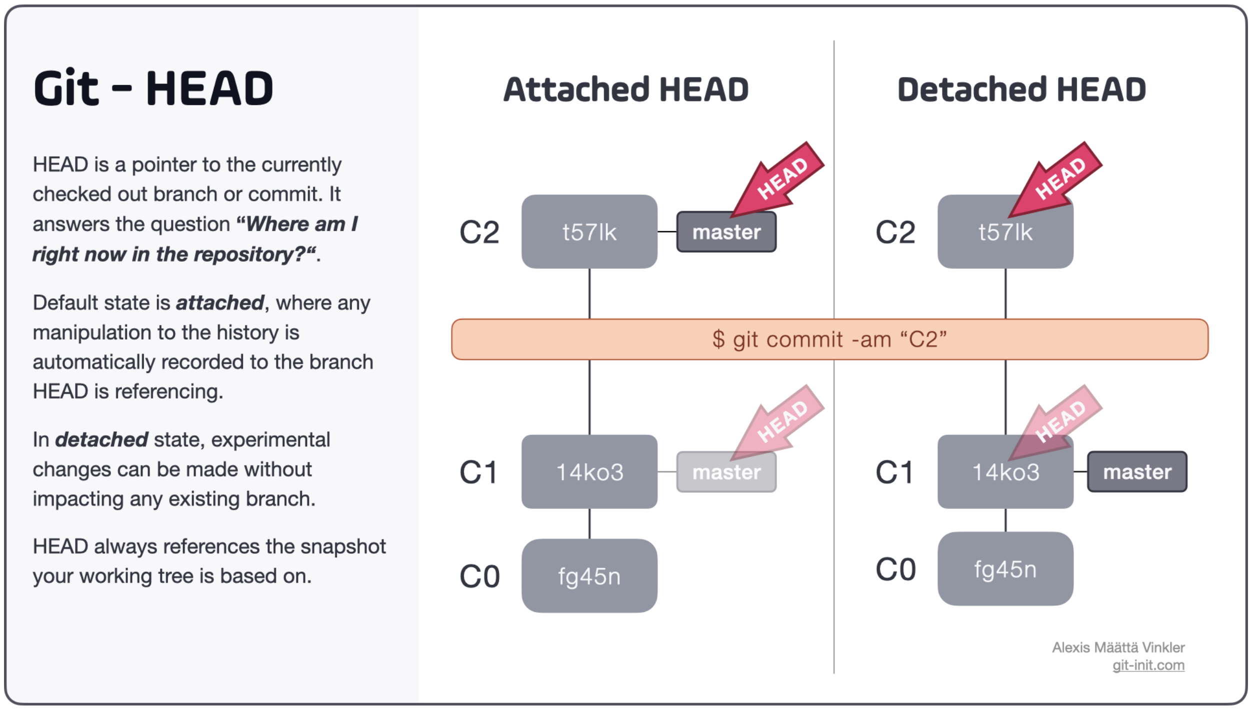
```
git checkout master^^
```

It changed its HEAD to commit –b which is called Detached HEAD.

Similarly, you can:

```
git checkout HEAD~2
```

HEAD~2 refers to commits previous from HEAD


Git - HEAD

HEAD is a pointer to the currently checked out branch or commit. It answers the question ***"Where am I right now in the repository?"***.

Default state is ***attached***, where any manipulation to the history is automatically recorded to the branch HEAD is referencing.

In ***detached*** state, experimental changes can be made without impacting any existing branch.

HEAD always references the snapshot your working tree is based on.

```
$ git commit -am "C2"
```

So, why is the head important?
* It gives the developer the ability to move back to a previous checkpoint in the project development.
* It helps keep track of commits.

GIT Commit

- **Atomic Commit**

    When you do an atomic commit, you're committing only one change. It might be across many files, but it's one single change. The smallest, most important improvement you can make in your source code. It's large enough to add value but tiny enough to be manageable. Atomic commits are easier to read, easier to track, easier to review.

- Commit early, Commit often

    It would be best if you commit your changes early and often. Early commit helps to cut the risk of conflicts between two concurrent changes. Additionally, having periodic checkpoints means that you can understand how you broke something. You can commit locally "all the time," and you need to push when everything is working. Your message will be clear and focused and easy for other developers to understand the code history.

- Do not commit dependencies

  Your Git repository is to manage your source code. It's not to store the dependencies. Also, committing your dependencies will significantly increase the size of the project repository. Instead of using Git, use a useful build/dependency tool. Examples include Maven for Java, npm for node apps, etc. Your repository stays organized. You will not be worried about updating your project dependencies

- Do not commit local configuration files

  You should not commit your local config files into the source control for many good reasons. Config files might hold secrets or personal preferences. And configuration files might change from environment to environment. Different users have different local settings. It also reduces merge conflicts in these local files, most of the time our time is wasted managing conflicts in local configuration files.

- Do not commit broken code

  This blocks other developers from working on their tasks. (with broken code). You should not commit broken code in the repository before leaving the office at the end of the day. You can consider Git's "Stash" feature instead of Git commit & push. Your code repository will stay stable. At any given time, there is no risk of halting the team's productivity because of the broken code.

- Use Git Stash with a message

  You can stash your uncommitted modifications (both staged and unstaged) for later usage. You can use them back from your working copy. Git stash is excellent for storing changes temporarily. You should use Git stash with a meaningful message.

  $ git stash save "Your meaningful stash message".

- Test Your changes before committing

  t's always good to test your changes before you commit them to your local repository. Testing is essentially an extra layer of security before you commit. It allows you to identify mistakes and bugs quicker before they go to your production server. Dev testing helps to decrease software faults and save the expense of the project in the long term.

- Write useful commit messages

  The commit message should be straightforward. And it should be meaningful. The right commit message helps other developers in many ways. They can understand the code better even without looking at it. Good commit messages act as documentation for other developers. Useful commit Messages help maintain the project in the long run.

- Review your commit (code review)

  In general, we should not commit or push our changes directly in the main branch. Dev team should check the sets of modifications before merging in the main branch. The Pull/Merge request model is a simple but successful model. Code review helps keep the company's coding style intact and finding bugs early, when it is cheap to patch them.

- Don't mix "refactoring" with a new feature

  The worst thing new developers do - refactoring and adding a new feature in the same commit. It's simply against git best practices. And, it is not suitable for many reasons. It will increase the chance of having conflicts that might be harder to resolve. Your code becomes easy to review. Team productivity will Increase.

- **Semantic Commit / Conventional Commits:**

  Semantic Commits are **commit messages with human and machine-readable meaning, which follow particular conventions**. This means, it's merely guidelines for commit messages, so that: The commit messages are semantic - because these are categorized into meaningful types, indicating the essence of the commit.

  The Conventional Commits format is as follows:

  <type>([optional scope]): <description>

  [optional body]

  [optional footer(s)]

  Type
  Only certain *types* are permitted, with the most common being:

  - fix: a commit that fixes a bug in Uno.
  - feat: a commit that adds new functionality to Uno.
  - docs: a commit that adds or improves Uno's documentation.
  - test: a commit that adds unit tests.
  - perf: a commit that improves performance, without functional changes.
  - chore: a catch-all type for any other commits. For instance, if you're implementing a single feature and it makes sense to divide the work into multiple commits, you should mark one commit as feat and the rest as chore.

  https://www.conventionalcommits.org/en/v1.0.0/

GIT Branch

The most important feature of git is its branching model. These branches are more like a new copy of your code's current state. The use of branches lets you manage the workflow more quickly and easily.

    a. A new git branch for a new feature.
    b. Another git branch for bug fixes.
    c. A new git branch for some LIVE issue, and so on.

- Branch naming convention

  It helps to keep your repository organized. The right branch name helps in avoiding confusion. If you do not use Git branch naming conventions, it leads to misunderstanding. It also complicates code maintenance.

  - Use Separators

    You can use a separator in your branch name. Many developers use slash, and others use hyphens or underscores. Which one to use depends on the needs of both you and your team. Example of branch name with a separator

    - feature-sso-implemented
    - bug_login_page_redirect
  - Use group-word in the branch name

    The idea is to use group words, so any developer can tell the branch's purpose by looking at it. These group words add more clarity. And it becomes easy to organize your branches. For example -

    - wip - the prefix wip indicates work in progress.
    - exp - experimental branch for dev purpose.
    - feature - adding a new feature.
- Delete stale branches

  Without the possibility of losing any code, you can securely remove a git branch. And You should remove old git branches that are no longer in use. For example - you completed a feature and merged your branch with master. Now you do not need your feature branch. If you do not delete old branches, you may end up with many stale-branches, and it's become hard to manage.

- Keep your branches up to date

  You should make sure you are correctly using pull and push. And you should consistently merge your base branch into your current branch as you work. It's beneficial if it's a long-outstanding branch. This practice will help you to save your time and energy. Otherwise, you will spend your time resolving an unnecessary amount of merge conflicts.

- Protect your main branch/branches

  By protecting the main branch, nobody should make a direct check in to the main branch. Or can rewrite the history of the main branch. Also, only merge requests should be allowed in master after a code review process. So, no git pushes straight to the main branch. By protecting main branch, it's your production code, ready for the world to roll out and it always stays in stable mode.

GIT RESET

The purpose of the “git reset” command is to move the current HEAD to the commit specified (in this case, the HEAD itself, one commit before HEAD and so on).

We have 3 different types for git reset. They all rewrite Git history, and they all move the HEAD back, but they deal with the changes differently:

- git reset --soft, which will keep your files, and stage all changes back automatically.
- git reset --hard, which will *completely destroy any changes and remove them from the local directory*. Only use this if you know what you’re doing.
- git reset --mixed, which is the default, and keeps all files the same but unstages the changes. This is the most flexible option, but despite the name, it doesn’t modify files.

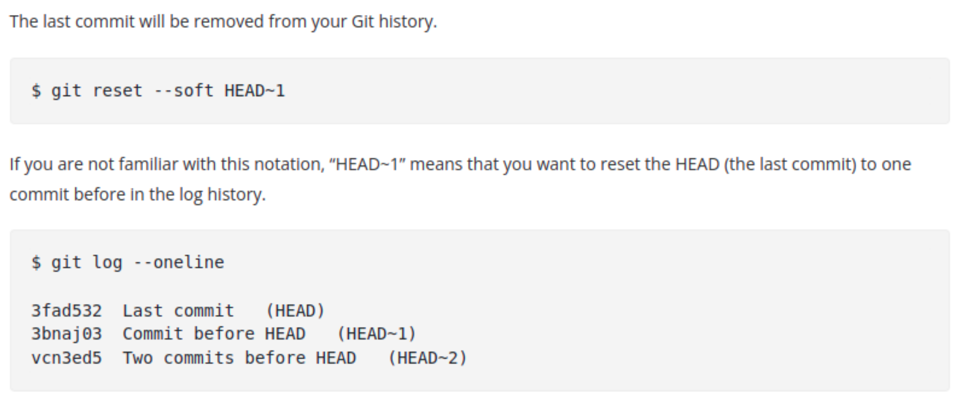
Usage:

By default, git reset uses '-- mixed’ reset

```
git reset
```

GIT SOFT RESET:

```
git reset –soft HEAD~1
```



The last commit will be removed from your Git history.

```
$ git reset --soft HEAD~1
```

If you are not familiar with this notation, “HEAD~1” means that you want to reset the HEAD (the last commit) to one commit before in the log history.

```
$ git log --oneline

3fad532  Last commit   (HEAD)
3bnaj03  Commit before HEAD   (HEAD~1)
vcn3ed5  Two commits before HEAD   (HEAD~2)
```

GIT RESET HARD:

First, it's always worth noting that git reset --hard is a potentially dangerous command, since it throws away all your uncommitted changes. For safety, you should always check that the output of git status is clean (that is, empty) before using it.

You can also reset going back to a specific commit:

Eg:

```
git reset --hard <SOME-COMMIT>
```

GIT RESET SOFT vs. MIXED vs. HARD

The difference between --mixed and --soft is whether or not your index is also modified. So, if we're on branch master with this series of commits:

- A - B - C (master)

HEAD points to C and the index matches C.

When we run

```
git reset --soft B
```

master (and thus HEAD) now points to B, but the index still has the changes from C; git status will show them as staged. So if we run git commit at this point, we'll get a new commit with the same changes as C.

Now if we perform

```
git reset --mixed B
```

(Note: --mixed is the default option). Once again, master and HEAD point to B, but this time the index is also modified to match B. If we run git commit at this point, nothing will happen since the index matches HEAD. We still have the changes in the working directory, but since they're not in the index, git status shows them as unstaged. To commit them, you would git add and then commit as usual.

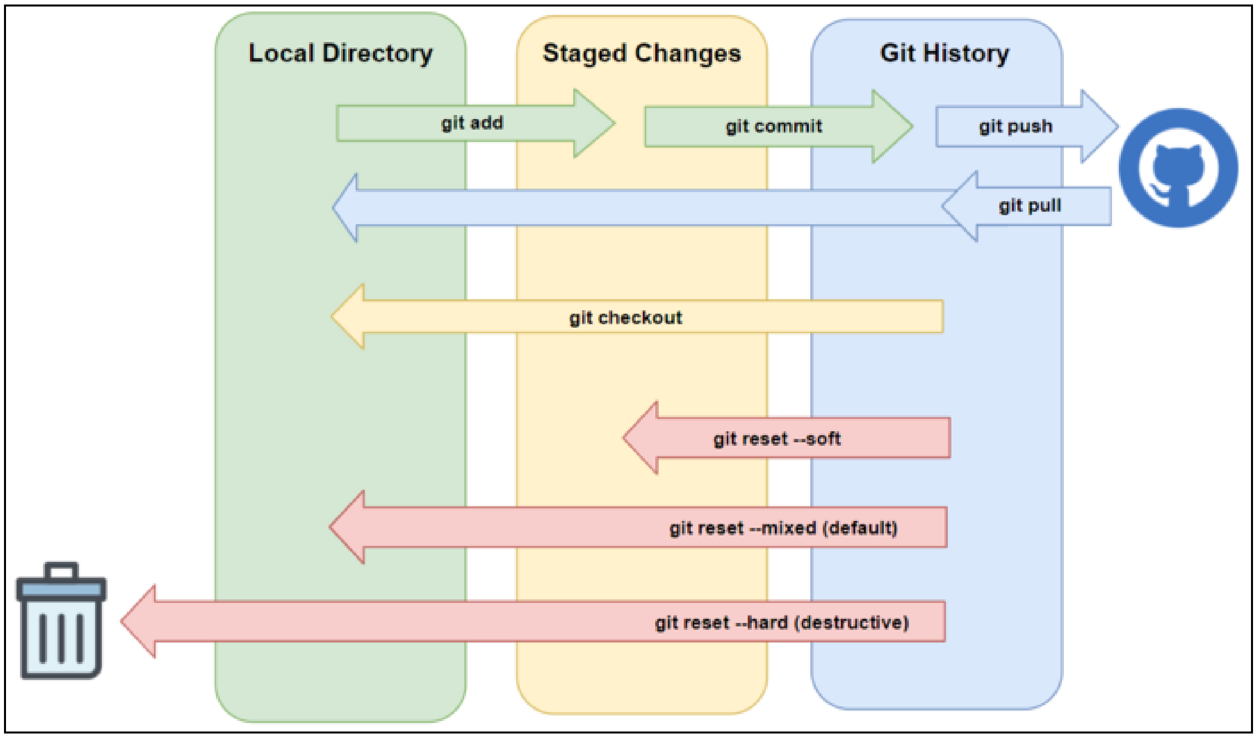
And finally, --hard is the same as --mixed (it changes your HEAD and index), except that --hard also modifies your working directory. If we're at C and run

```
git reset --hard B
```

then the changes added in C, as well as any uncommitted changes you have, will be removed, and the files in your working copy will match commit B. Since you can permanently lose changes this way, you should always run git status before doing a hard reset to make sure your working directory is clean or that you're okay with losing your uncommitted changes.

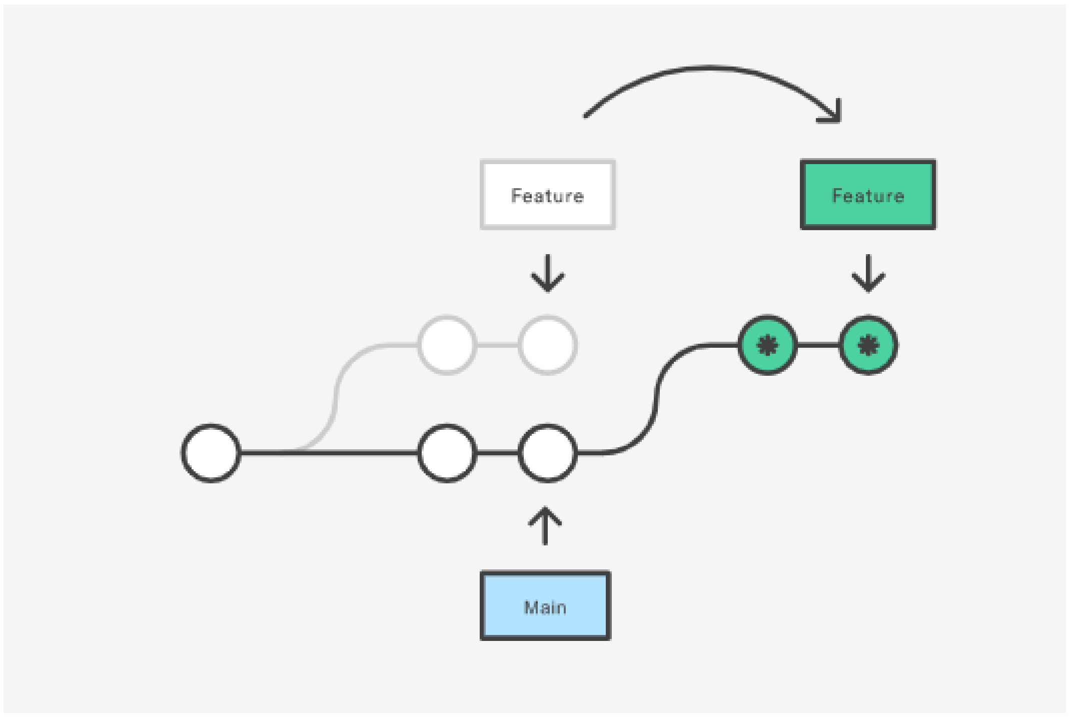
In the simplest terms:

- **--soft**: **uncommit** changes, changes are left staged (*index*).
- **--mixed** *(default)*: **uncommit + unstage** changes, changes are left in *working tree*.
- **--hard**: **uncommit + unstage + delete** changes, nothing left.



GIT Rebase

Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record**.** Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow.



USEAGE:

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the main branch has progressed since you started working on a feature branch. You want to get the latest updates to the main branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest main branch.

Rebase your working branch frequently

a. You should rebase your working branch frequently to prevent bugs, rework, and the tedious job of resolving conflicts with the upstream branch. You can do it easily by running these commands:
    - **git checkout <upstream_branch>**
    - **git pull**
    - **git checkout -**
    - **git rebase <upstream_branch>**

b. A cool trick is using **git fetch --prune**. It doesn't require you to checkout to the upstream branch to update it.

GIT Pull/Merge Request

Pull Requests are vital as they help ensure that quality code. Short pull requests allow developers to review and quickly merge code into the main branch efficiently. The main branch of your project repository becomes a production-ready branch. And the only commits on the main branch come from pull requests. Also, pull requests help others to review your changes. It improves the quality and helps to make sure only stable code is going into the master branch.

- For one concern - One Pull Request

  It is necessary for a pull request to be atomic. It does not mean a pull request is a single commit. It can either be a construct of one or several atomic commits. When you raise a pull request, make sure it's only and only for one feature. It's easy to review and much faster to get approved and team productivity and quality boost.

- Do not delay in Pull Request

  If you delay in attending the pull requests, the code quality gets reduced. There's no way back to the bad code. It is like a virus, it is. It will spread, and the standard of your code will forever be gone. Pull requests should not be unattended for a long time. Also, this delay will increase the chances of more conflicts.

- Complete your work before a Pull Request

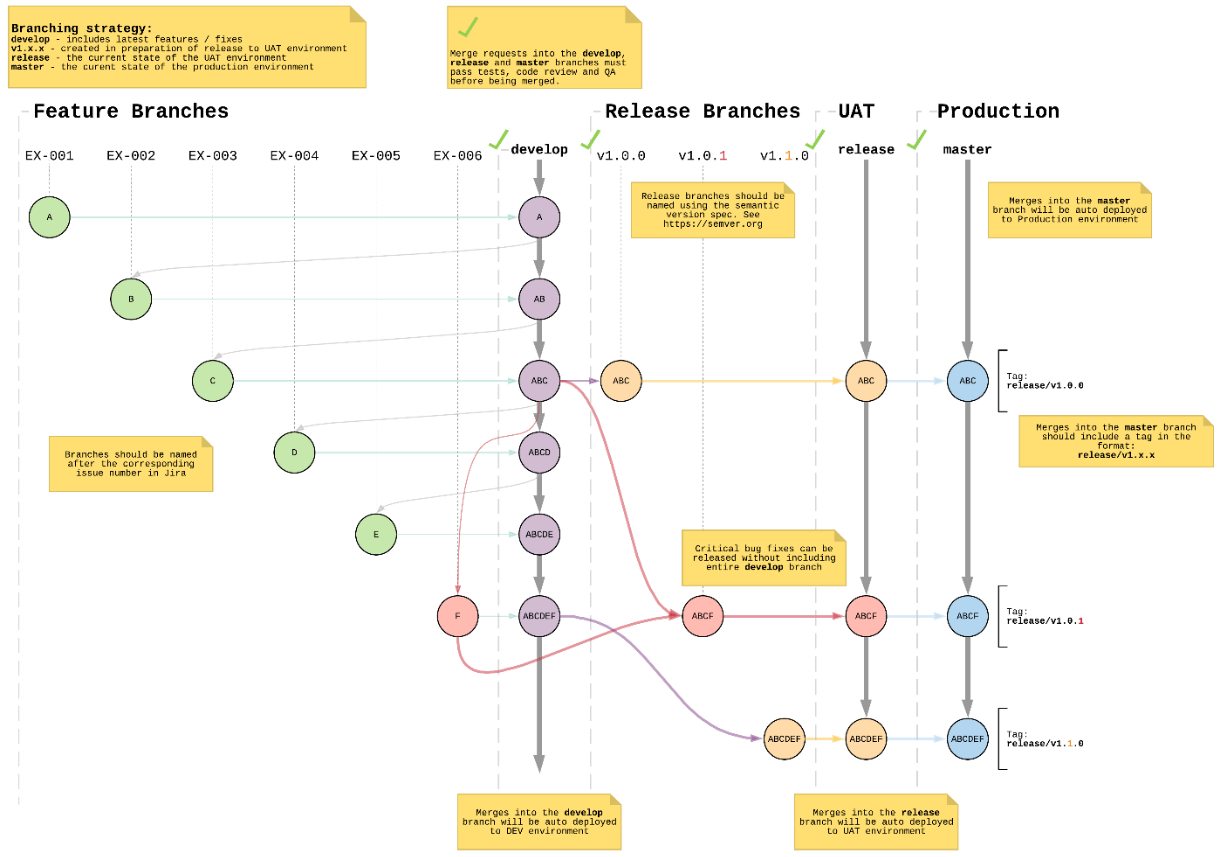
  Pull requests must be atomic in nature. You need to present the code changes in a pleasant way. So do not create a Pull request for your broken code. Complete your work before a Pull Request And in case of a large pull request, split by features. Tidy up your work and all commits before a pull request.

- Useful description with Pull Request

  This helps faster the review process and also the description gives a clear idea of what the pull request is all about. Clear and precise description with the pull request is a must.

  a. Needs to be precise on what you worked on what bug have been fixed
  b. Use standard GIT format
  c. Example for APIs:
     i. Mention API URL, its Method and Body Post with Response Example screenshot or format
  d. For UI: post a screenshot for the task completed for a fresh page
  e. For Changes and Fixes: clear detail on them

Feature Branch Workflow


Branching strategy:
develop - includes latest features / fixes
v1.x.x - created in preparation of release to UAT environment
release - the current state of the UAT environment
master - the curent state of the production environment
Merge requests into the develop, release and master branches must pass tests, code review and QA before being merged.
Feature Branches
EX-001
EX-002
EX-003
EX-004
EX-005
EX-006
develop
Release Branches
v1.0.0
v1.0.1
v1.1.0
UAT
release
Production
master
Release branches should be named using the semantic version spec. See https://semver.org
Merges into the master branch will be auto deployed to Production environment
A
B
AB
C
ABC
Tag: release/v1.0.0
Merges into the master branch should include a tag in the format: release/v1.x.x
Branches should be named after the corresponding issue number in Jira
D
ABCD
E
ABCDE
Critical bug fixes can be released without including entire develop branch
F
ABCDEF
ABCF
Tag: release/v1.0.1
Tag: release/v1.1.0
Merges into the develop branch will be auto deployed to DEV environment
Merges into the release branch will be auto deployed to UAT environment

Article: https://nvie.com/posts/a-successful-git-branching-model/

Naming Convention:

Branches:

1. Main Branches: Development, UAT (Staging), Production.
2. Feature Branches: Named after the corresponding issue number in PM tool.
   a. Examples: DC-FT1
3. Release Branch: Named after the version number

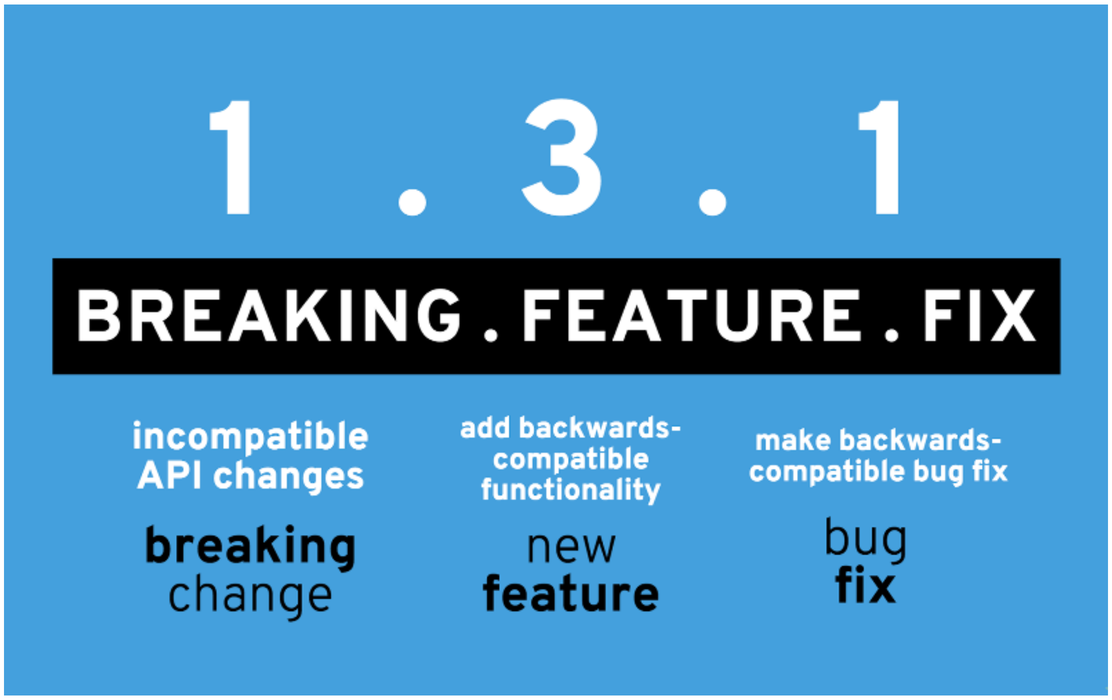
Comments:

1. Comments can be of any type, whether to find the bug in the code , or some ways for optimizing a code, or discuss on the feature as a whole, etc.

Development Workflow

Development Flow

1. Developer or a development team will create a new feature branch for every feature assigned. Feature branch should be named after the feature code in PM tool.
2. After feature completion, merge request should be sent to development branch. A senior member of the development team should review the code and accept merge request.
3. After a proper integration, and regression test, senior QA should merge the branch to UAT/Staging.
4. Again, after tests are performed by client and end users, staging branch should be merged with production.
5. For each recent release, a release branch should be kept with version number following semantics displayed below



6. For every immediate bug fix: Pull from release branch, make fixes, test and deploy

Merge Request Title:

Main task title that you have worked on the branch

Merge Request Description:

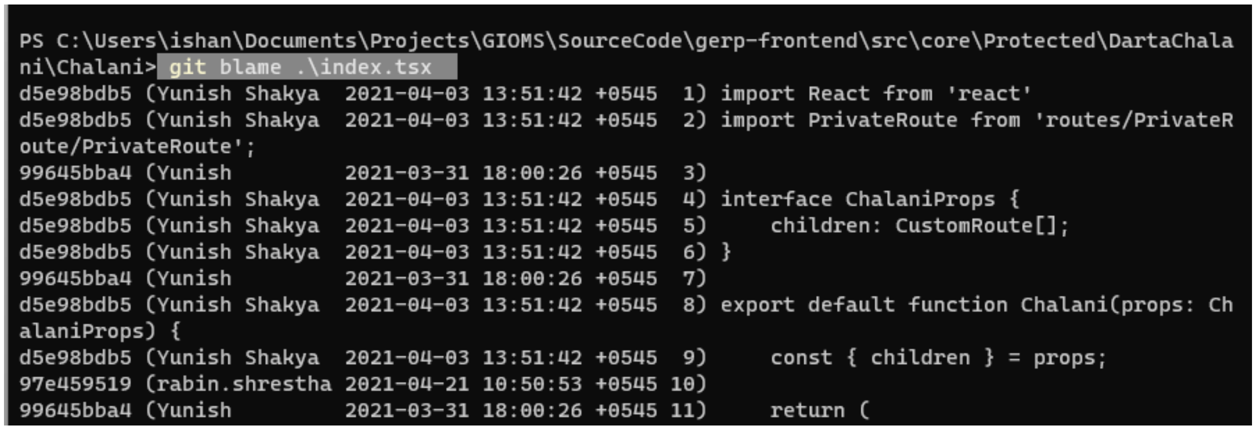
1. Needs to be precise on what you worked on what bug have been fixed
2. Use standard GIT format
3. Example for APIs:

- Mention API URL, its Method and Body Post with Response Example screenshot or format

4. For UI: post a screenshot for the task completed for a fresh page
5. For Changes and Fixes: clear detail on them

Notes

1. Don't git push straight to master. Branch it out!
    a. Create new branch: **git branch -b <branch_name>**
    b. Swith to another branch: **git checkout <branch_name>**
    c. A cool trick is to use **git checkout -** to switch back to the previous branch.
2. Run **git blame <file_name>** to list line by line:
    a. The last commit that changed it
    b. The author of the commit
    c. The timestamp of the commit
    d. Other git log commands
        i. git log
        ii. git log –oneline
        iii. Group commits by user - git shortlog



```
PS C:\Users\ishan\Documents\Projects\GIOMS\SourceCode\gerp-frontend\src\core\Protected\DartaChala
ni\Chalani> git blame .\index.tsx
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  1) import React from 'react'
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  2) import PrivateRoute from 'routes/PrivateR
oute/PrivateRoute';
99645bba4 (Yunish         2021-03-31 18:00:26 +0545  3)
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  4) interface ChalaniProps {
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  5)     children: CustomRoute[];
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  6) }
99645bba4 (Yunish         2021-03-31 18:00:26 +0545  7)
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  8) export default function Chalani(props: Ch
alaniProps) {
d5e98bdb5 (Yunish Shakya  2021-04-03 13:51:42 +0545  9)     const { children } = props;
97e459519 (rabin.shrestha 2021-04-21 10:50:53 +0545 10)
99645bba4 (Yunish         2021-03-31 18:00:26 +0545 11)     return (
```

3. Write descriptive and meaningful commit messages.
    a. https://chris.beams.io/posts/git-commit/
    b. Commits should look very similar to a task list
4. Commit only related work
    a. Apply Single Responsibility principle for commits, not only to the code. You should commit the least number of lines that make sense together.
    b. This practice helps when digging into the codebase, like when using **git log** or **git blame**.
5. Rebase your working branch frequently
    a. You should rebase your working branch frequently to prevent bugs, rework, and the tedious job of resolving conflicts with the upstream branch. You can do it easily by running these commands:

        i. **git checkout <upstream_branch>**
        ii. **git pull**
        iii. **git checkout -**
        iv. **git rebase <upstream_branch>**

b. A cool trick is using **git fetch --prune**. It doesn't require you to checkout to the upstream branch to update it.

GIT Bisect

- Used for finding out the bug commit
- usage:

git bisect [help|start|bad|good|new|old|terms|skip|next|reset|visualize|view|replay|log|run]

https://git-scm.com/docs/git-bisect

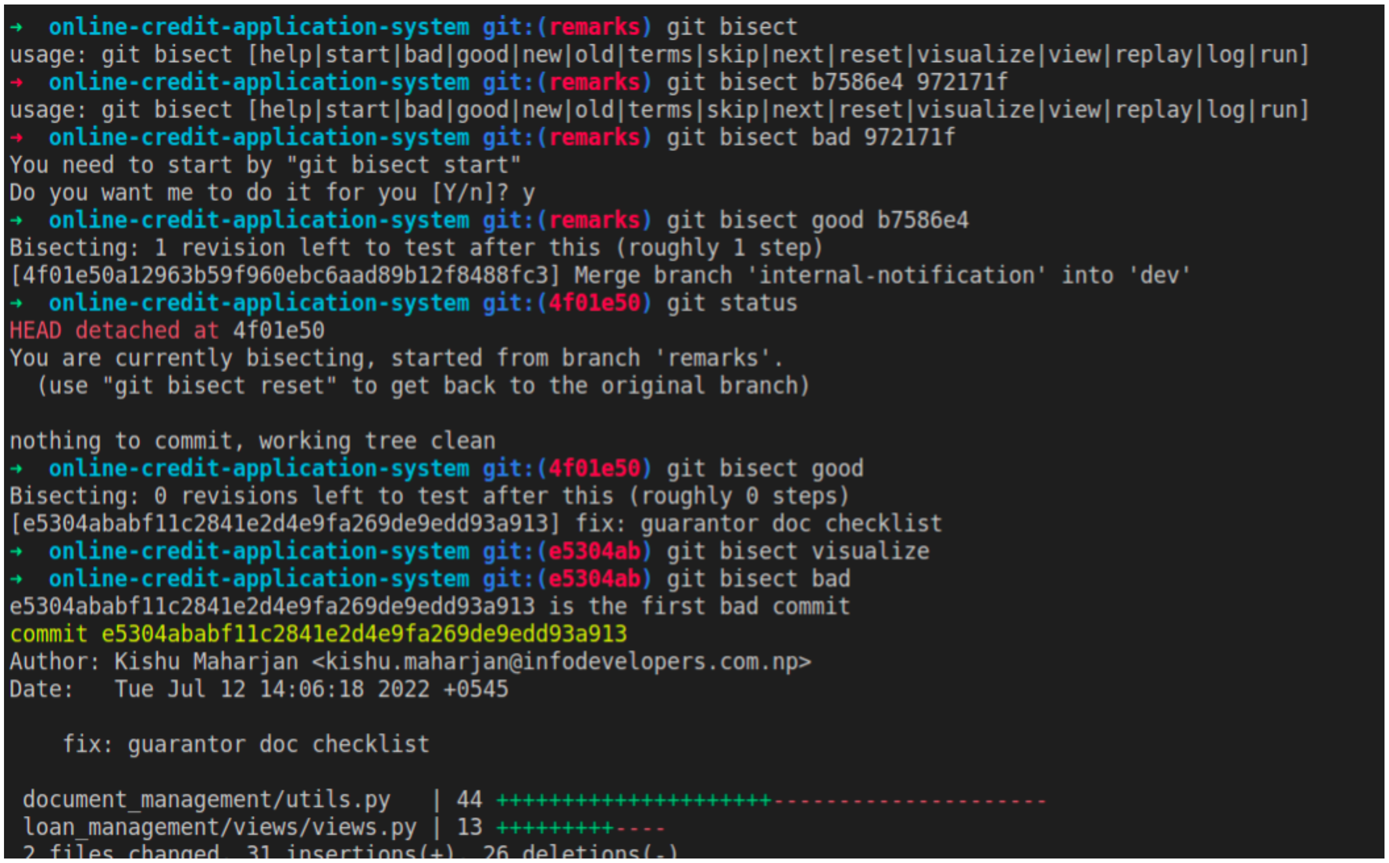
```
git bisect start
$ git bisect bad                 # Current version is bad
$ git bisect good v2.6.13-rc2
```

```
git bisect start to start bisecting

git bisect bad <commit /branch>

git bisect good <commit /branch>
```

```
Using commit for bisect
```



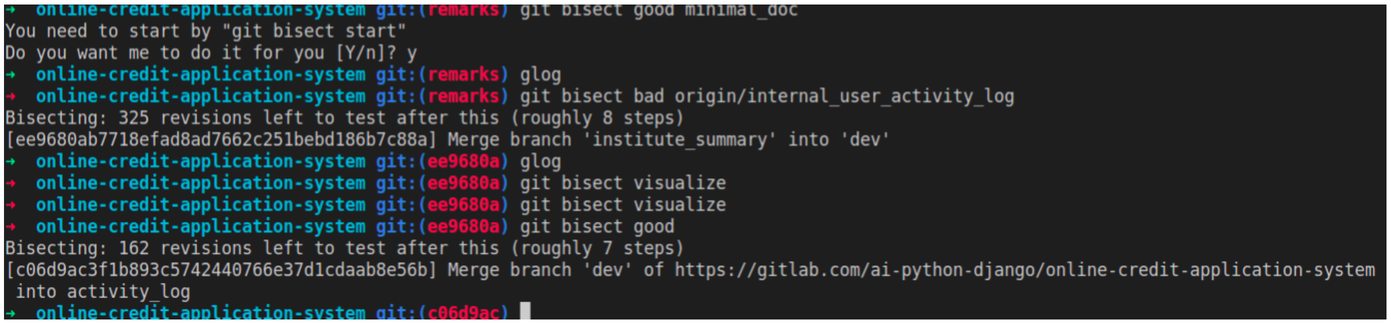
```
→  online-credit-application-system git:(remarks) git bisect
usage: git bisect [help|start|bad|good|new|old|terms|skip|next|reset|visualize|view|replay|log|run]
→  online-credit-application-system git:(remarks) git bisect b7586e4 972171f
usage: git bisect [help|start|bad|good|new|old|terms|skip|next|reset|visualize|view|replay|log|run]
→  online-credit-application-system git:(remarks) git bisect bad 972171f
You need to start by "git bisect start"
Do you want me to do it for you [Y/n]? y
→  online-credit-application-system git:(remarks) git bisect good b7586e4
Bisecting: 1 revision left to test after this (roughly 1 step)
[4f01e50a12963b59f960ebc6aad89b12f8488fc3] Merge branch 'internal-notification' into 'dev'
→  online-credit-application-system git:(4f01e50) git status
HEAD detached at 4f01e50
You are currently bisecting, started from branch 'remarks'.
  (use "git bisect reset" to get back to the original branch)

nothing to commit, working tree clean
→  online-credit-application-system git:(4f01e50) git bisect good
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[e5304ababf11c2841e2d4e9fa269de9edd93a913] fix: guarantor doc checklist
→  online-credit-application-system git:(e5304ab) git bisect visualize
→  online-credit-application-system git:(e5304ab) git bisect bad
e5304ababf11c2841e2d4e9fa269de9edd93a913 is the first bad commit
commit e5304ababf11c2841e2d4e9fa269de9edd93a913
Author: Kishu Maharjan <kishu.maharjan@infodevelopers.com.np>
Date:   Tue Jul 12 14:06:18 2022 +0545

    fix: guarantor doc checklist

 document_management/utils.py    | 44 +++++++++++++++++++++---------------------------
 loan_management/views/views.py | 13 +++++++++----
 2 files changed, 31 insertions(+), 26 deletions(-)
```

Using branch for bisect:



```
→  online-credit-application-system git:(remarks) git bisect good minimal_doc
You need to start by "git bisect start"
Do you want me to do it for you [Y/n]? y
→  online-credit-application-system git:(remarks) glog
→  online-credit-application-system git:(remarks) git bisect bad origin/internal_user_activity_log
Bisecting: 325 revisions left to test after this (roughly 8 steps)
[ee9680ab7718efad8ad7662c251bebd186b7c88a] Merge branch 'institute_summary' into 'dev'
→  online-credit-application-system git:(ee9680a) glog
→  online-credit-application-system git:(ee9680a) git bisect visualize
→  online-credit-application-system git:(ee9680a) git bisect visualize
→  online-credit-application-system git:(ee9680a) git bisect good
Bisecting: 162 revisions left to test after this (roughly 7 steps)
[c06d9ac3f1b893c5742440766e37d1cdaab8e56b] Merge branch 'dev' of https://gitlab.com/ai-python-django/online-credit-application-system
 into activity_log
→  online-credit-application-system git:(c06d9ac)
```

Git bisect reset: to end bisecting

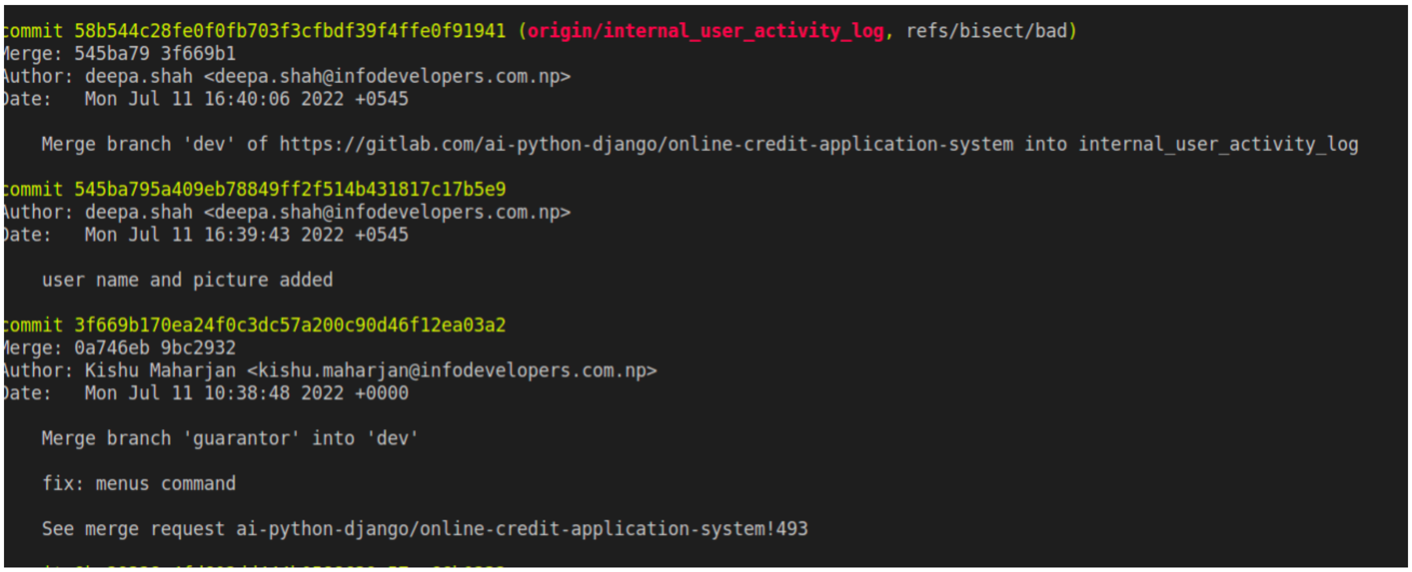
```
git bisect reset
```

If you want to reset to diff commit:

`git bisect reset <commit>`

`git bisect visualize`

Visualize the commit: shows git log in the particular commit



```
ommit 58b544c28fe0f0fb703f3cfbdf39f4ffe0f91941 (origin/internal_user_activity_log, refs/bisect/bad)
Merge: 545ba79 3f669b1
uthor: deepa.shah <deepa.shah@infodevelopers.com.np>
ate:   Mon Jul 11 16:40:06 2022 +0545

    Merge branch 'dev' of https://gitlab.com/ai-python-django/online-credit-application-system into internal_user_activity_log

ommit 545ba795a409eb78849ff2f514b431817c17b5e9
uthor: deepa.shah <deepa.shah@infodevelopers.com.np>
ate:   Mon Jul 11 16:39:43 2022 +0545

    user name and picture added

ommit 3f669b170ea24f0c3dc57a200c90d46f12ea03a2
erge: 0a746eb 9bc2932
uthor: Kishu Maharjan <kishu.maharjan@infodevelopers.com.np>
ate:   Mon Jul 11 10:38:48 2022 +0000

    Merge branch 'guarantor' into 'dev'

    fix: menus command

    See merge request ai-python-django/online-credit-application-system!493
```

REFERENCES

1. https://www.becomebetterprogrammer.com/git-head/
2. https://www.conventionalcommits.org/en/v1.0.0/
3. https://git-scm.com/docs/git-bisect