# Git Branching Strategy

Next to the main branches master and develop, our development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches

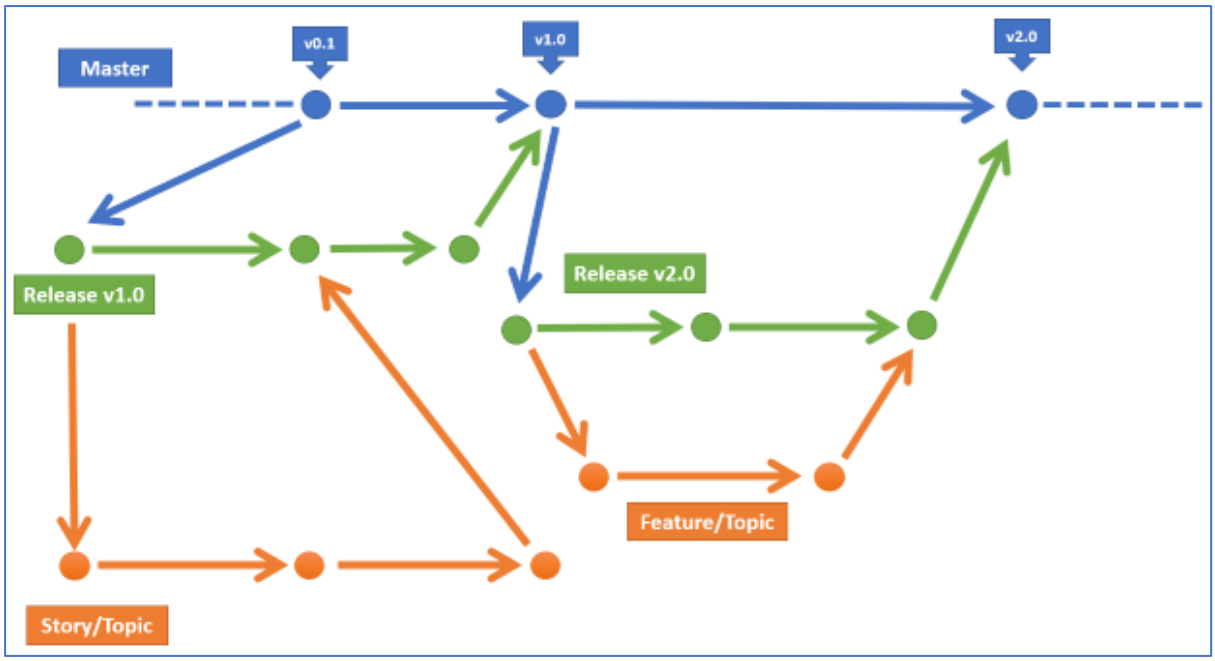
Each of these branches have a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets. We will walk through them in a minute.

Below mentioned is one of the most common and efficient Git workflows. With this one we can easily manage feature, develop and master branches all together in a particular flow. Changes can easily move from one branch to another with merge and rebase features.

# Release branch strategy

In This strategy we can work on multiple releases branched off from master branch. We can further create story branches from these release branch and then these release branches can be merged further to master branch. Master branch will receive merges from different release branches.

This also helps team to work simultaneously on various development work to be done on multiple releases.

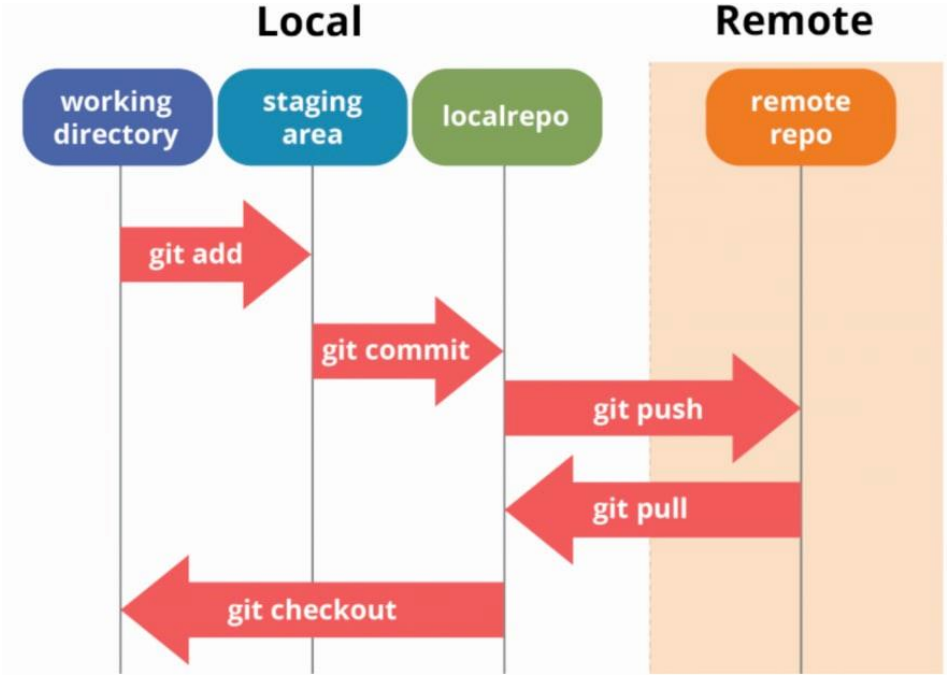


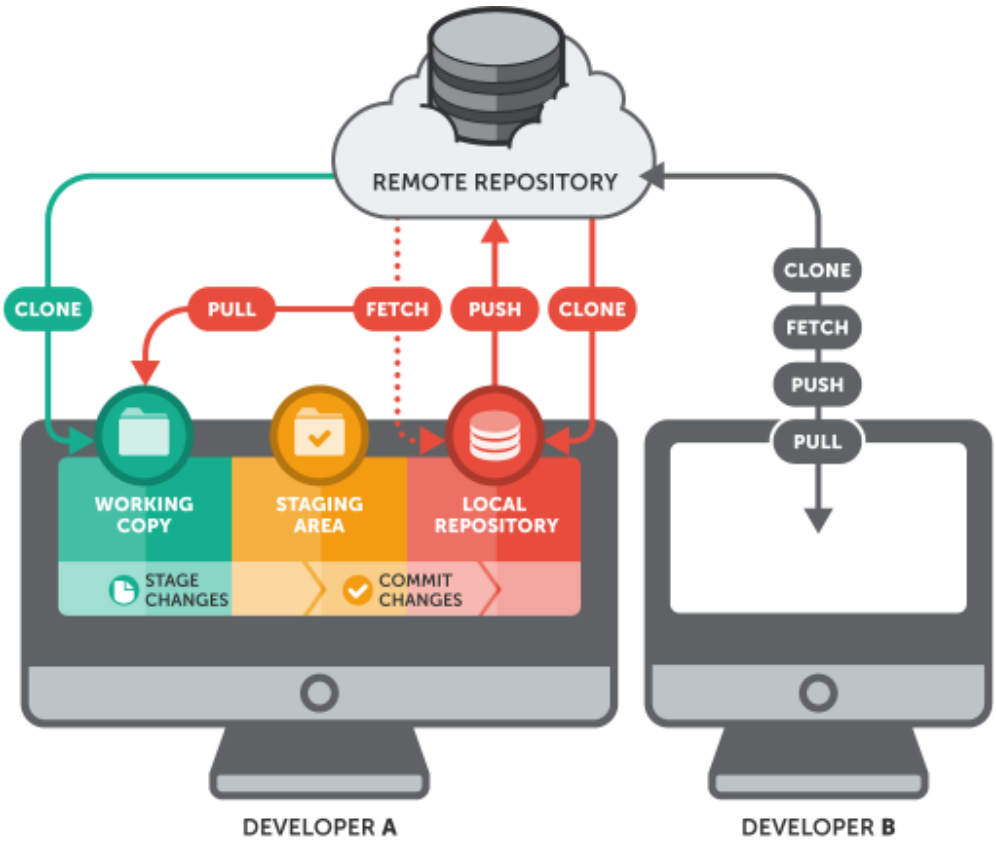

**Collaborating between Local and Remote Repository**

Git is one of distributed version control system which means we don't have to depend on network all the time. We have huge number of local repositories of a Git repository on multiple systems. We can perform all operations like commits, merge, rebases etc. without even having any connectivity with remote repository.

Local Repository: Local repository cloned on developers' machine where they can do modifications. These changes are present locally only and developer is having full control on them. These does not need any connectivity with remote repositories.

Remote Repository: Purpose of remote repository is to publish these local changes to remote repository. We can share remote repository with multiple developers and they can easily read them. We need remote repository only whenever we want to push changes to remote repo or pulling changes to local repo.



# Basic Git Operations

**Checking status for files:** We can anytime check the status of files in Git repository using git status command.

git status

**Creating New files:** We can create new files in repo to have untracked files using below commands:

echo “First Project” > README

git status

**Tracking new files:** In order to make untracked files as tracked we can use below commands:

git add .

git status

**Stage Modified files:** We can stage modified files so that we can commit those and these changes will become permanent once done.

git add .

git status

git commit –m “First Commit”

```
root@git:~/gitrepo# git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" t
root@git:~/gitrepo# echo "First Project" > README
root@git:~/gitrepo# git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be co

        README

nothing added to commit but untracked files present (use
root@git:~/gitrepo# git add .
root@git:~/gitrepo# git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README

root@git:~/gitrepo# git commit -m "First Commit"
[master c3ebfdc] First Commit
 1 file changed, 1 deletion(-)
root@git:~/gitrepo# git status
On branch master
nothing to commit, working tree clean
root@git:~/gitrepo#
```

## Reverting to Earlier Commits Demo

**Git Log:** We can check the log history of Git repo using git log command.

git log –oneline

**Git Reset:** Once we decide the commit to which we want to change our branch pointer, we can execute below command to reset:

git reset <commit_id>

git log (Commit should not be displayed in git log now)

**Git Revert:** it helps us to revert the changes done in a specific commit. This command is used when there is a situation of rollback.

git revert <commit_id>

git log –oneline (A new commit will be displayed now which reverts that specific commit)

```
root@git:~/gitrepo# ls -lartr
total 20
-rw-r--r-- 1 root root    0 Dec 17 02:32 README
-rw-r--r-- 1 root root   11 Dec 17 02:54 CONTRIBUTION.md
-rw-r--r-- 1 root root   14 Dec 17 02:54 LICENSE.md
drwx------ 5 root root 4096 Dec 17 02:54 ..
drwxr-xr-x 3 root root 4096 Dec 17 02:54 .
drwxr-xr-x 8 root root 4096 Dec 17 02:54 .git
root@git:~/gitrepo# git log --oneline
0be1aa1 (HEAD -> master) Adding License File
9d3d772 Adding Contribution File
5a0d5ea Adding Readme file
root@git:~/gitrepo# git reset --hard 9d3d772
HEAD is now at 9d3d772 Adding Contribution File
root@git:~/gitrepo# git log --oneline
9d3d772 (HEAD -> master) Adding Contribution File
5a0d5ea Adding Readme file
root@git:~/gitrepo# git revert 9d3d772
[master 9ab3705] Revert "Adding Contribution File"
 1 file changed, 1 deletion(-)
 delete mode 100644 CONTRIBUTION.md
root@git:~/gitrepo# git log --oneline
9ab3705 (HEAD -> master) Revert "Adding Contribution File"
9d3d772 Adding Contribution File
5a0d5ea Adding Readme file
root@git:~/gitrepo# ls -lart
total 12
-rw-r--r-- 1 root root    0 Dec 17 02:32 README
drwx------ 5 root root 4096 Dec 17 02:54 ..
drwxr-xr-x 3 root root 4096 Dec 17 02:55 .
drwxr-xr-x 8 root root 4096 Dec 17 02:56 .git
root@git:~/gitrepo#
```

## Deleting Files in Git Demo

**Git rm**: Git rm command is used to delete files from git repositories. Once below command is executed the file will be deleted from local repository. In case this is done by mistake we can run git revert command to revert these changes.

git rm README

git status

git add .

git commit -m “Deleting Readme File from Git”

**Git rm --cached**: This option helps to delete file from git but file will not be permanently deleted from local repository. We can easily access file even after running this command.

git rm --cached CONTRIBUTION.md

```
root@git:~/gitrepo# ls -lart
total 16
-rw-r--r-- 1 root root    0 Dec 17 02:32 README
drwx------ 5 root root 4096 Dec 17 02:54 ..
-rw-r--r-- 1 root root   11 Dec 17 03:16 CONTRIBUTION.md
drwxr-xr-x 8 root root 4096 Dec 17 03:16 .git
drwxr-xr-x 3 root root 4096 Dec 17 03:16 .
root@git:~/gitrepo# git rm README
rm 'README'
root@git:~/gitrepo# git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    README

root@git:~/gitrepo# git add .
root@git:~/gitrepo# git commit -m "Remove README file"
[master e65717d] Remove README file
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 README
root@git:~/gitrepo# git rm --cached CONTRIBUTION.md
rm 'CONTRIBUTION.md'
root@git:~/gitrepo# git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    CONTRIBUTION.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        CONTRIBUTION.md

root@git:~/gitrepo#
```

## Deleting Files in Git Demo

**Local .gitignore:** if we create a file in our local repository with name .gitignore, Git will be able to read this file to decide which files and folders to ignore while performing git commit. Create some log and image files which is described in .gitignore file.

touch app.log

touch image.jpg

git status (This will not show those files which is mentioned in .gitignore file)

**Global .gitignore:** We can create both local and global .gitignore configuration to exclude some files from commit.

git config --global core.excludesFile ~/.gitignore

git config --global –list

```
root@git:~/gitrepo# ls -alrtt
total 16
drwx------ 5 root root 4096 Dec 17 02:54 ..
-rw-r--r-- 1 root root   11 Dec 17 03:30 CONTRIBUTION.md
-rw-r--r-- 1 root root    0 Dec 17 03:30 README
drwxr-xr-x 8 root root 4096 Dec 17 03:30 .git
drwxr-xr-x 3 root root 4096 Dec 17 03:30 .
root@git:~/gitrepo# git status
On branch master
nothing to commit, working tree clean
root@git:~/gitrepo# vi app.log
root@git:~/gitrepo# touch image.jpg
root@git:~/gitrepo# git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore

root@git:~/gitrepo# cat .gitignore
*.log
*.jpg
root@git:~/gitrepo# cp .gitignore ~/
root@git:~/gitrepo# git config --global core.excludesFile ~/.gitignore
root@git:~/gitrepo# git config --global --list
user.name=root
user.email=root@git.c.fluent-cyclist-193302.internal
core.excludesfile=/root/.gitignore
root@git:~/gitrepo#
```

## Renaming Files in Git Demo

- **Git mv:** Git move command is to rename files in Git repo without losing history of that file. Once the file is renamed, we also have to perform commit in order to save those changes permanently in repository. We can anytime browse through history of this repo to see which file is modified and renamed.

  git mv old_filename new_filename

  git status

  git commit –m “Renamed file in Git”

```
root@git:~/gitrepo# ls -alrt
total 20
-rw-r--r-- 1 root root   11 Dec 17 03:30 CONTRIBUTION.md
-rw-r--r-- 1 root root    0 Dec 17 03:30 app.log
-rw-r--r-- 1 root root    0 Dec 17 03:31 image.jpg
-rw-r--r-- 1 root root   12 Dec 17 03:31 .gitignore
drwx------ 5 root root 4096 Dec 17 03:34 ..
-rw-r--r-- 1 root root    0 Dec 17 03:44 README
drwxr-xr-x 3 root root 4096 Dec 17 03:44 .
drwxr-xr-x 8 root root 4096 Dec 17 03:44 .git
root@git:~/gitrepo# git status
On branch master
nothing to commit, working tree clean
root@git:~/gitrepo# git mv README README.md
root@git:~/gitrepo# git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README -> README.md

root@git:~/gitrepo# git commit -m "Renamed file in Git"
[master 8729fa3] Renamed file in Git
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README => README.md (100%)
root@git:~/gitrepo# ls -lart
total 20
-rw-r--r-- 1 root root   11 Dec 17 03:30 CONTRIBUTION.md
-rw-r--r-- 1 root root    0 Dec 17 03:30 app.log
-rw-r--r-- 1 root root    0 Dec 17 03:31 image.jpg
-rw-r--r-- 1 root root   12 Dec 17 03:31 .gitignore
drwx------ 5 root root 4096 Dec 17 03:34 ..
-rw-r--r-- 1 root root    0 Dec 17 03:44 README.md
drwxr-xr-x 3 root root 4096 Dec 17 03:44 .
drwxr-xr-x 8 root root 4096 Dec 17 03:44 .git
root@git:~/gitrepo#
```

## Git Branching Demo

Git branching helps us to manage the branches in Git repositories. We can create, switch and delete these branches according to our requirements.

**List Branch:** We can list branches in an existing repository using command:

git branch –l

**Create Branch:** We can create branch in Git to support parallel and independent programming. Use below command to create branch.

git branch <branch_name>

**Delete Branch:** Any time we can delete obsolete branches from our repository so that our repo can be light weight.

git branch –D <branch_name>

```
root@git:~/gitrepo# git branch -l
* master
root@git:~/gitrepo# git branch develop
root@git:~/gitrepo# git checkout develop
Switched to branch 'develop'
root@git:~/gitrepo# git branch -l
* develop
  master
root@git:~/gitrepo# git branch -D develop
error: Cannot delete branch 'develop' checked out at
root@git:~/gitrepo# git checkout master
Switched to branch 'master'
root@git:~/gitrepo# git branch -D develop
Deleted branch develop (was 8729fa3).
root@git:~/gitrepo# git branch -l
* master
root@git:~/gitrepo#
```

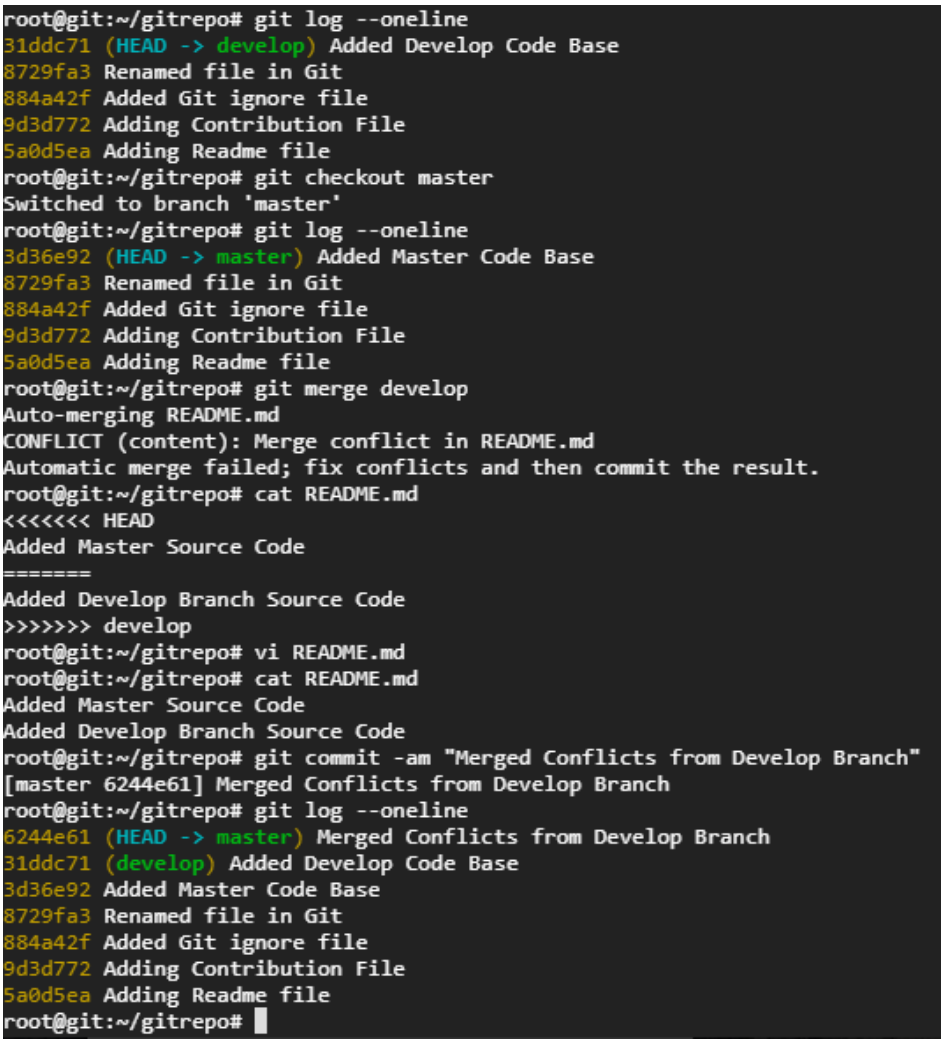
**Switch Branch:** We can switch between branches so that we can work on different streams of source code. It helps us to move from one branch to another.

git checkout <branch_name>

## Resolving Merge Conflicts in Git

Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand-new change, which may incorporate changes from both branches.

Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge. In this example, both changes are incorporated into the final merge:



## Git Branching

Git branching helps us to manage the branches in Git repositories. We can create, switch and delete these branches according to our requirements.

**List Branch:** We can list branches in an existing repository using command:

```
git branch –l
```

**Create Branch:** We can create branch in Git to support parallel and independent programming. Use below command to create branch.

```
git branch <branch_name>
```

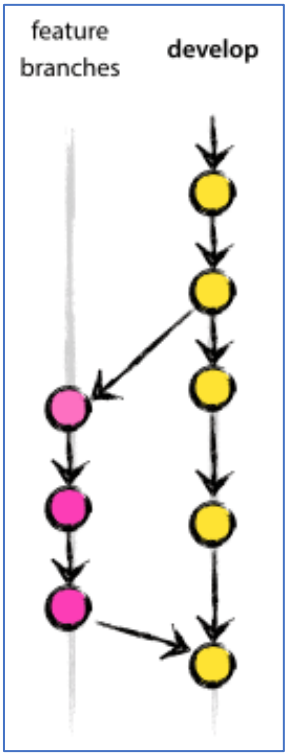
**Delete Branch:** Any time we can delete obsolete branches from our repository so that our repo can be light weight.

```
git branch –D <branch_name>
```

**Switch Branch:** We can switch between branches so that we can work on different streams of source code. It helps us to move from one branch to another.

```
git checkout <branch_name>
```

## Creating a feature/story branch




Feature branches (or sometimes called story branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point.

When starting work on a new story, branch off from the develop branch.

**$ git checkout -b story1 develop**

**Switched to a new branch "story1"**

Now Developer will be working on this individual story branches and performs

Check-ins to these respective branches as per below flow:

**$ git checkout develop**

**Switched to branch 'develop'**

**$ git merge --no-ff story1**

**Updating ea1b82a..05e9557**

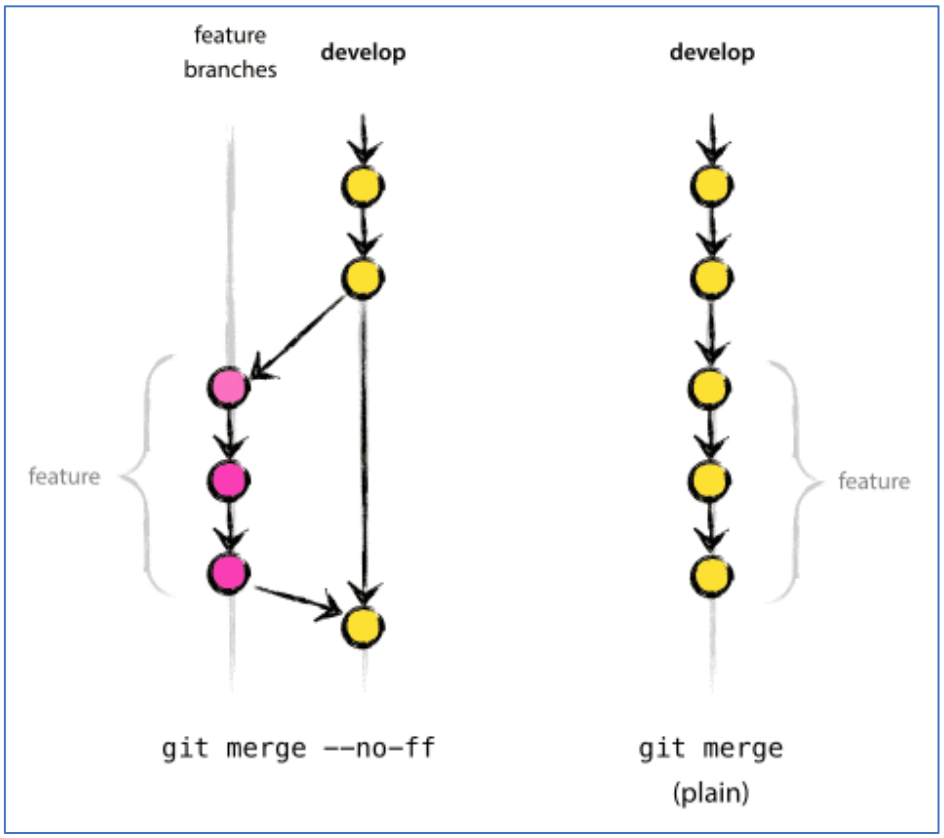
**(Summary of changes)**

**$ git branch -d story1**

**Deleted branch story1 (was 05e9557).**

**$ git push origin develop**

The **--no-ff** flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:



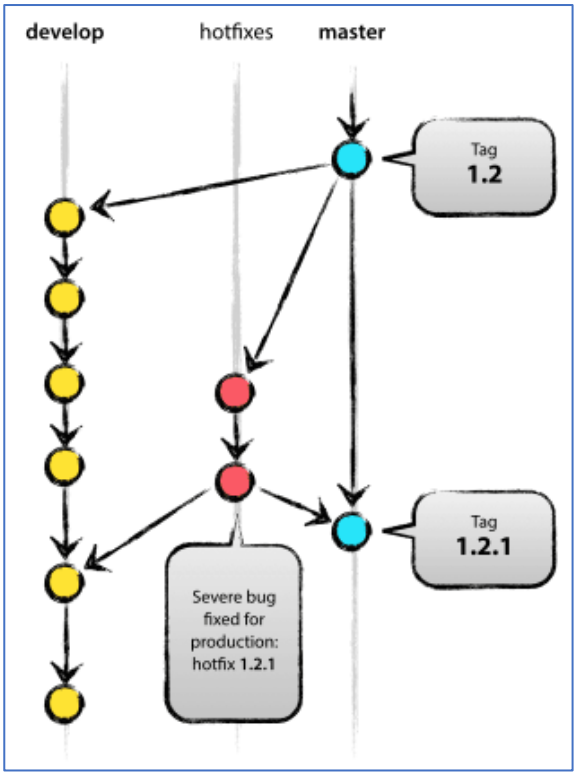
## Creating a Hotfix branch

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version.

When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

**$ git checkout -b hotfix-1.2.1 master**

**Switched to a new branch "hotfix-1.2.1"**




**$ git commit -m "Fixed severe production problem"**

**[hotfix-1.2.1 abbe5d6] Fixed severe production problem**

**5 files changed, 32 insertions (+), 17 deletions (-)**

**NOTE: When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well.**

**$ git checkout master**

**Switched to branch 'master'**

**$ git merge --no-ff hotfix-1.2.1**

**Merge made by recursive.**

**(Summary of changes)**

**$ git tag -a 1.2.1**

Next, include the bugfix in develop, too:

**$ git checkout develop**

**Switched to branch 'develop'**

**$ git merge --no-ff hotfix-1.2.1**

**Merge made by recursive.**

**(Summary of changes)**