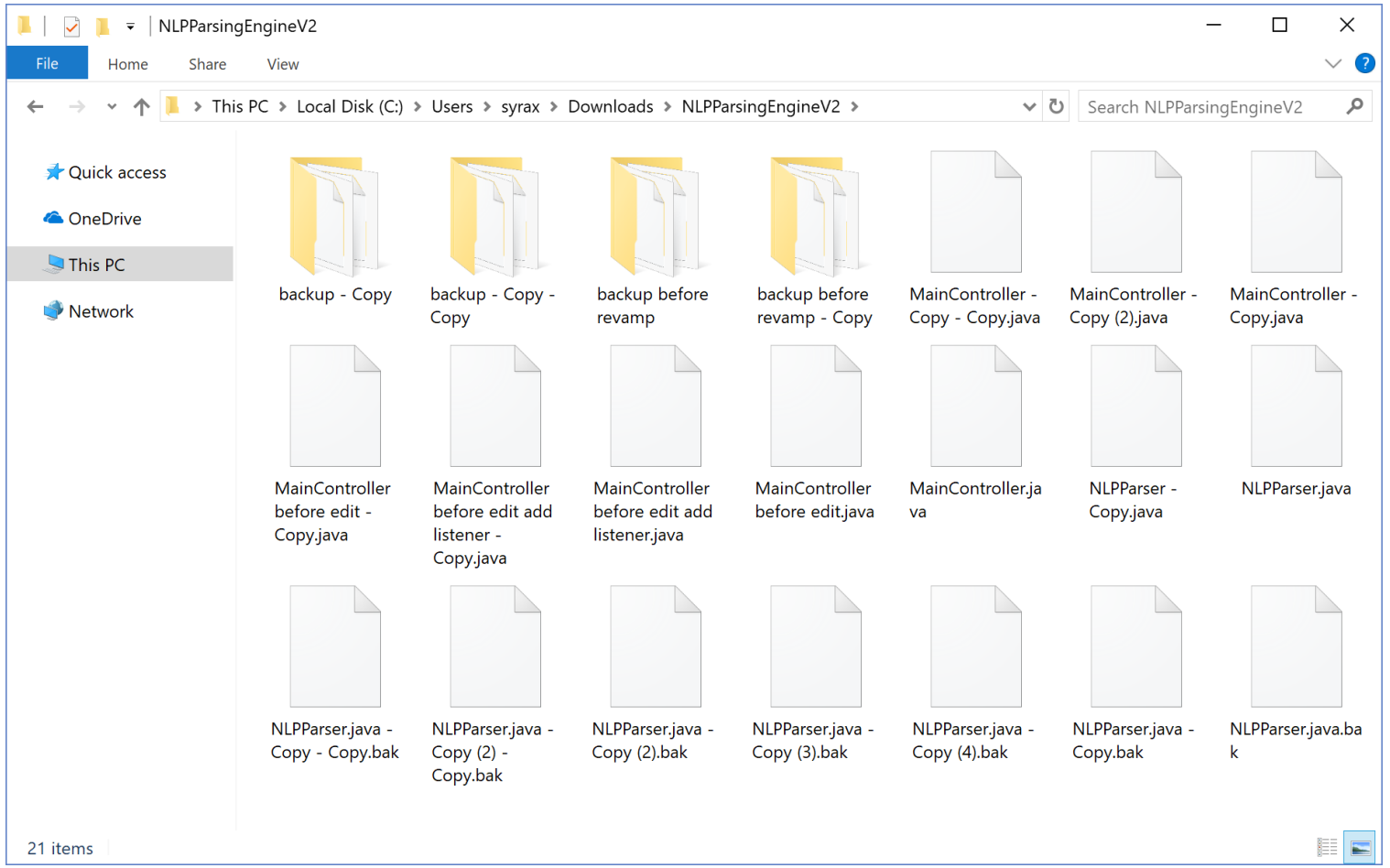
sea
connecting the dots

LC127 - Git Essentials

Kelvin Ang

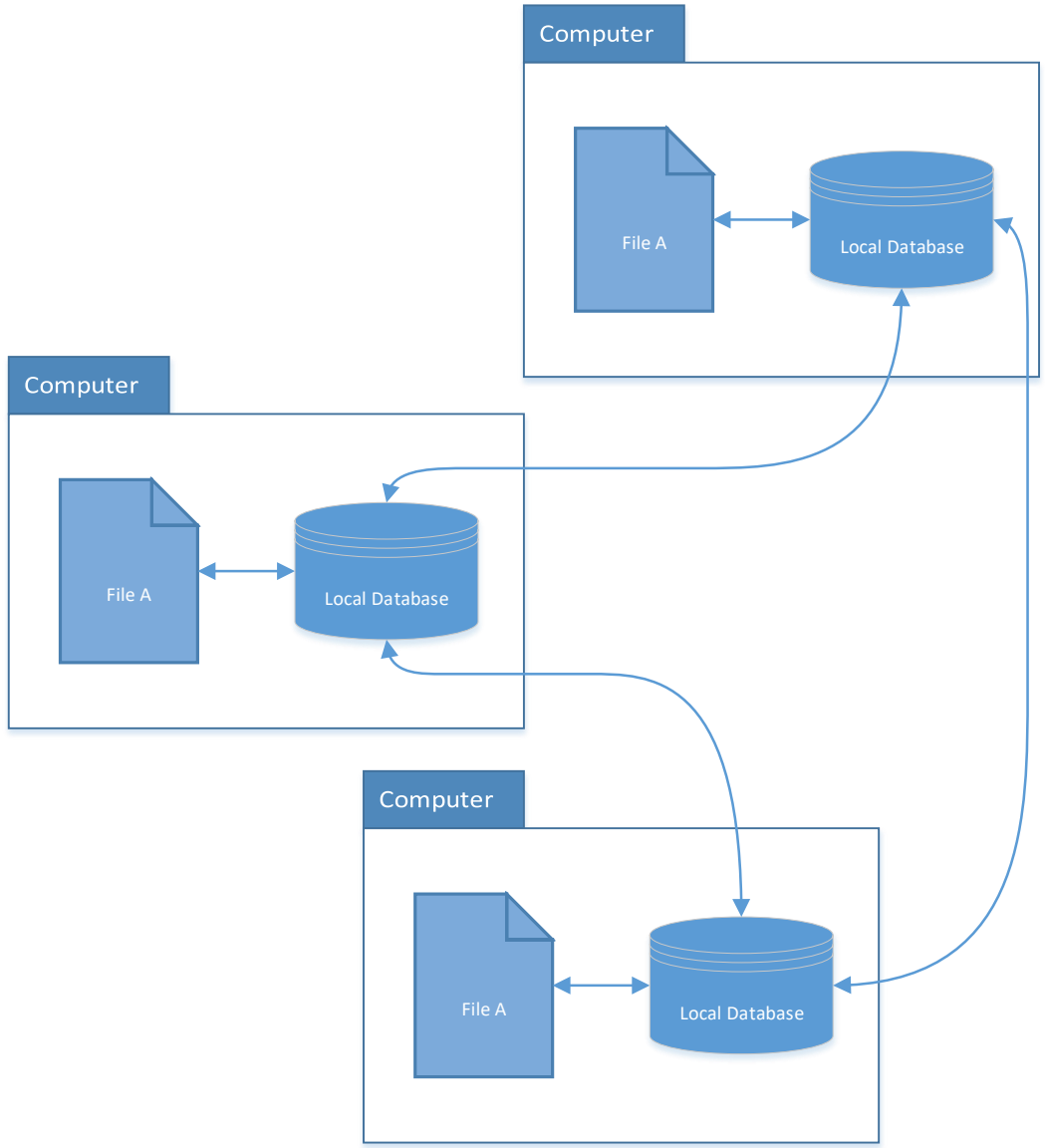
We learned what VCS solves...



We learned that Git is distributed…

- **Decentralized / Distributed**
  - Clients have identical clones of entire repositories.
  - If any server is destroyed, any copy of the repository can be used to restore it.
  - You can work on a plane or in outer space without Internet connectivity, and then push changes later.
  - All actions other than push and pull are very fast.
  - You can still apply centralized workflow, you just don't *have* to.

- Example(s):
  - Git, Mercurial, Bazaar, Darcs



We learned Git's history...

- **Andrew Tridgell** reverse-engineered BitKeeper, and copyright holder Larry McVoy withdrew its free status.
- Created in 2005 as a replacement for BitKeeper for the development of the Linux Kernel.
- Written with lessons learned from BitKeeper's inefficiencies.

- Trivia: What is Andrew Tridgell known for?

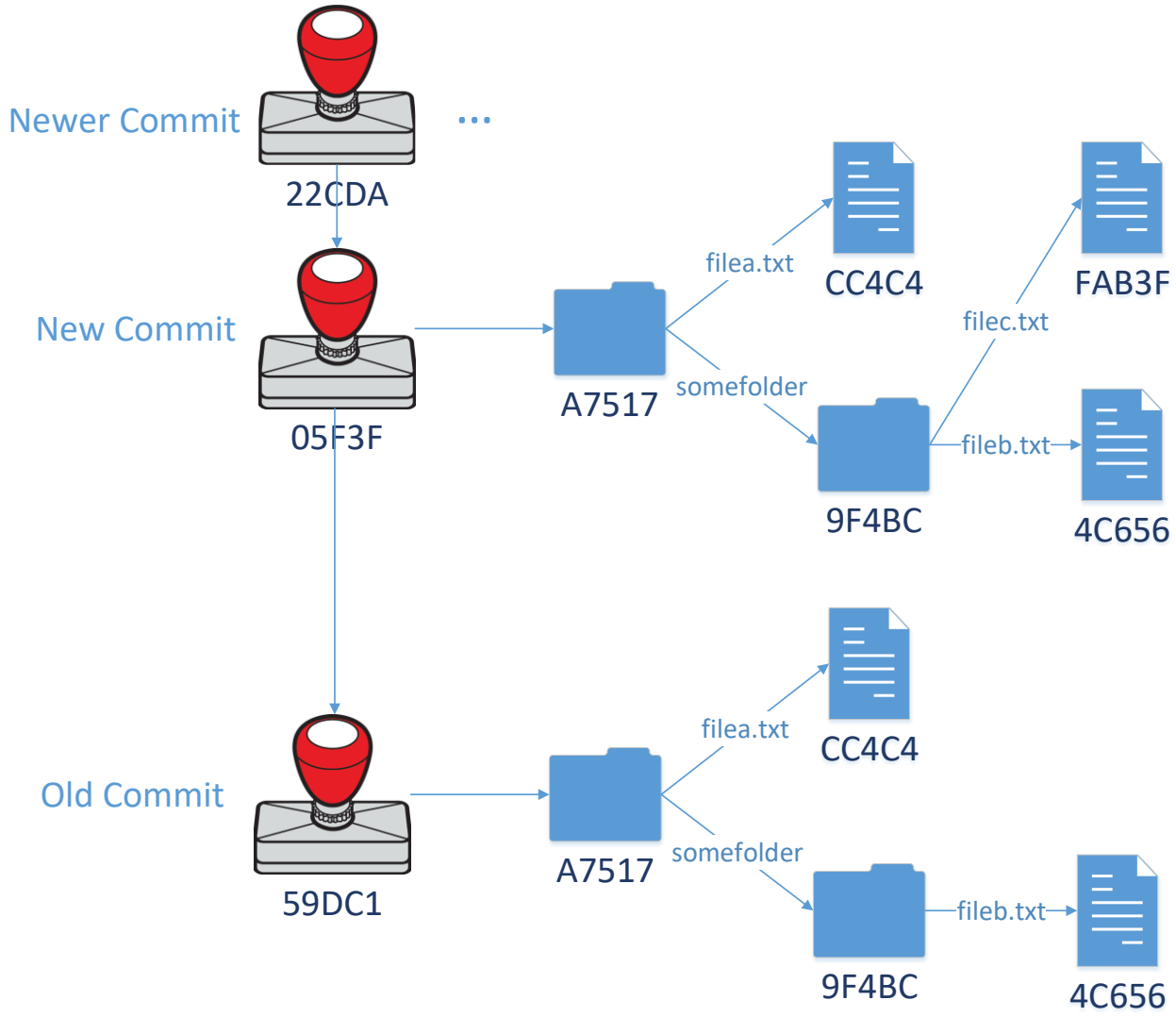
We learned Git's goals...

- Speed
- Simplicity
- Non-linear development (thousands of parallel branches)
- Fully distributed
- Efficiency for handling large projects (performance and data size)

git

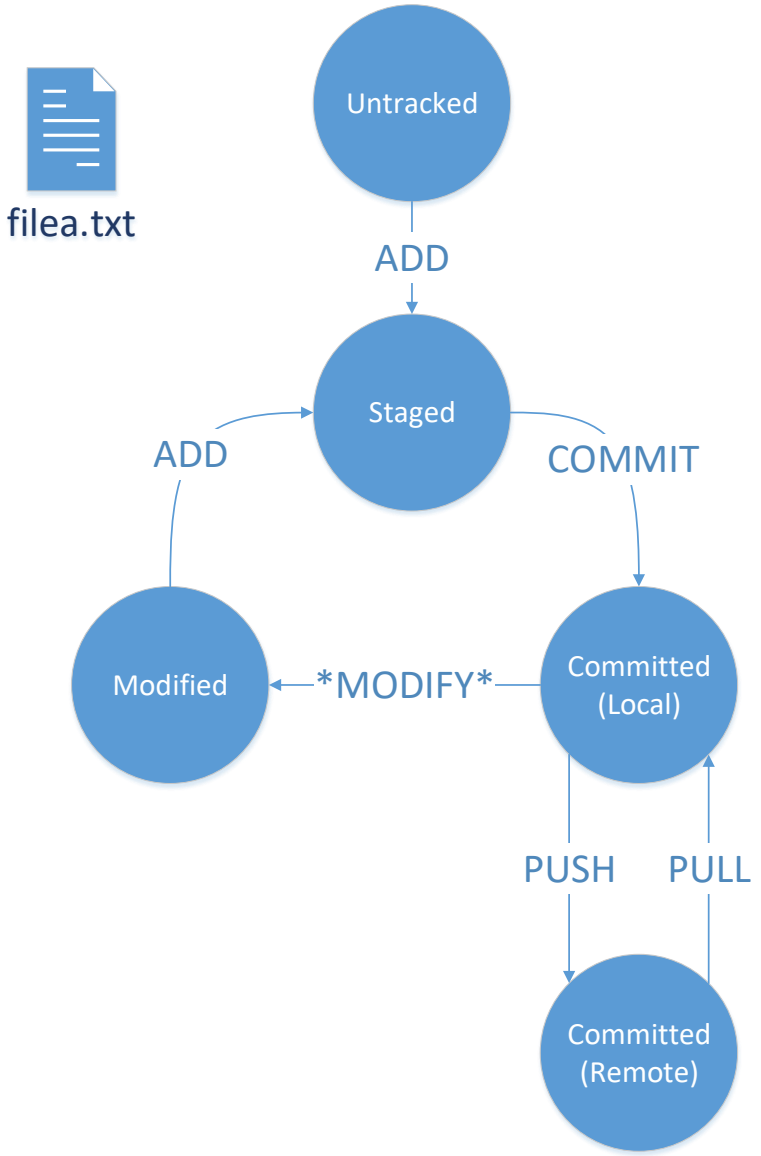
We learned Git's structure...

- Like any other objects, a commit is named by the SHA-1 hash of its content and is stored in the objects directory.
- A commit allows reconstruction of a particular version of a project simply by traversing through the entire subgraph from its root tree object.
- When checking out a commit, Git creates a folder for each tree object, and retrieves a file for each blob object.



We learned Git's file states...

- Track File: **git add**
- Stage File: **git add**
- Commit Staged Files: **git commit**
- Push changes: **git push**
- Pull changes: **git pull**



We learned to work in a local repository…

- We track and stage this README.md file using:

`git add README.md`

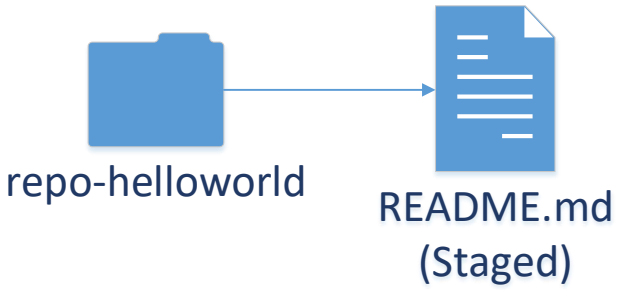
- Note that if you make any changes after staging, the file will be BOTH staged and modified (run "git status").
- If you commit now, the version you initially staged will go into the commit.
- Recall that staging actually already adds a new object, and the object is referred to by an entry in the index.

```
angk@angk-Latitude-E5570:~/repo-helloworld$ git add README.md
angk@angk-Latitude-E5570:~/repo-helloworld$ git status
On branch master

Initial commit

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

        new file:   README.md
```



We learned about .gitignore & .gitkeep…

- Suppose that you are working on a Python project. You will generate a lot of .pyc bytecode files that Python compiles the source to (Python does this to improve execution speed, instead of being purely interpreted).
- To exclude files from being added into your repository, you can specify a .gitignore file in the root of your project (alongside the .git folder).
- To exclude .pyc files, add this in your .gitignore file:

*.pyc

helloworld.py

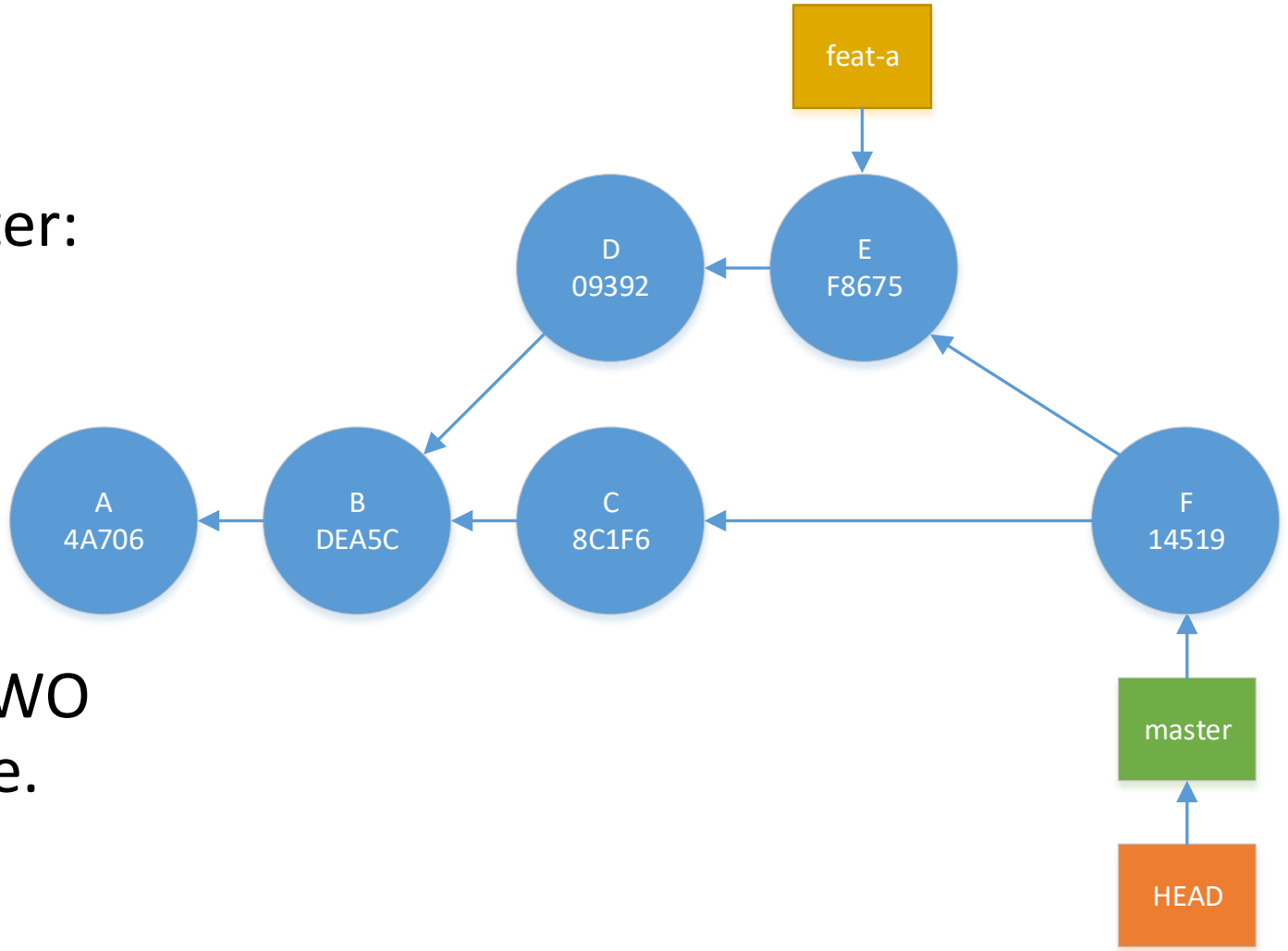
helloworld.pyc

We learned to work with branches…

- Then we attempt to merge changes from feat-a into master:

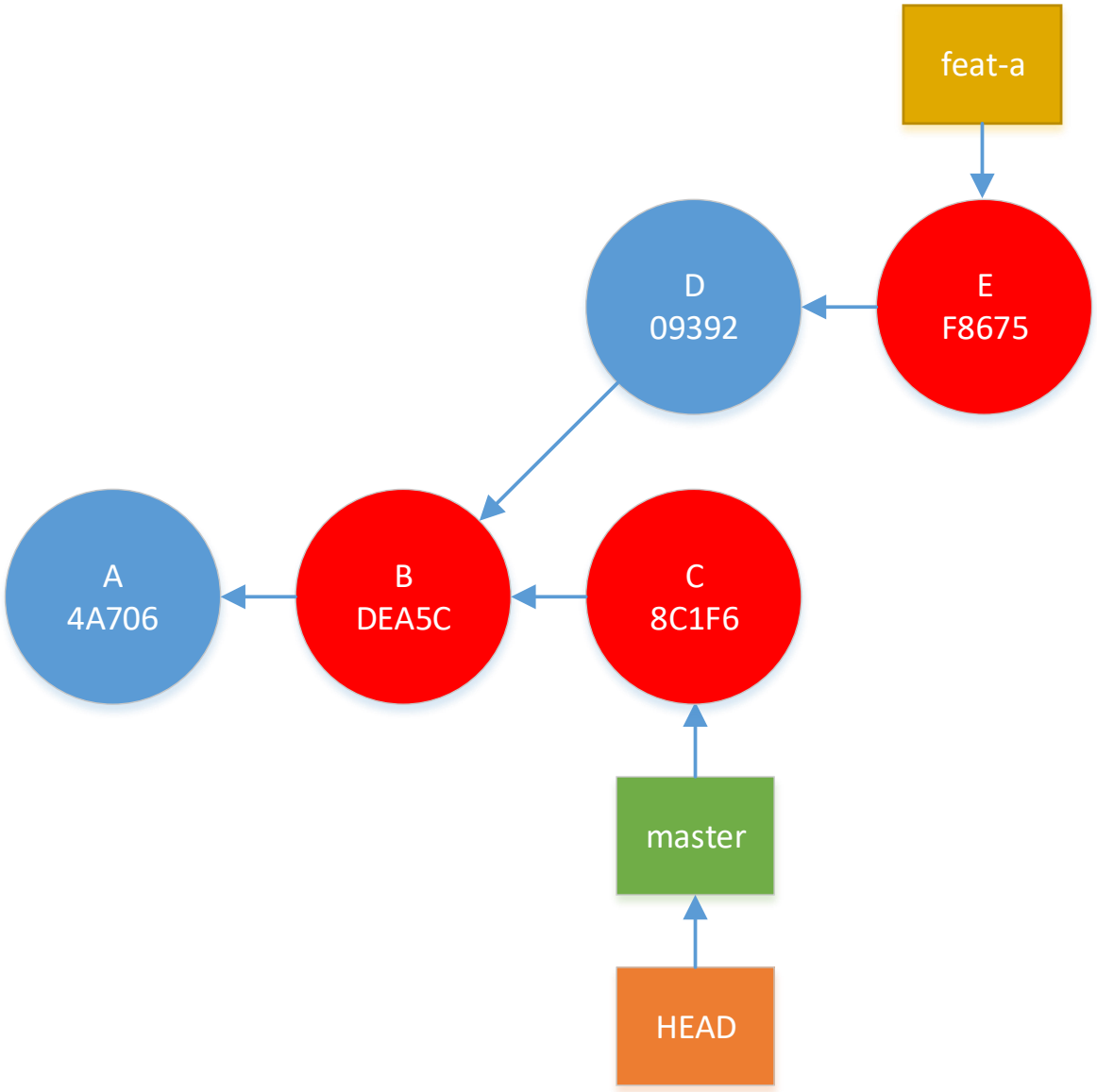
```
git merge feat-a
```

Notice that there is a special commit, commit F, which has TWO parents instead of the usual one. This is a **merge commit**.



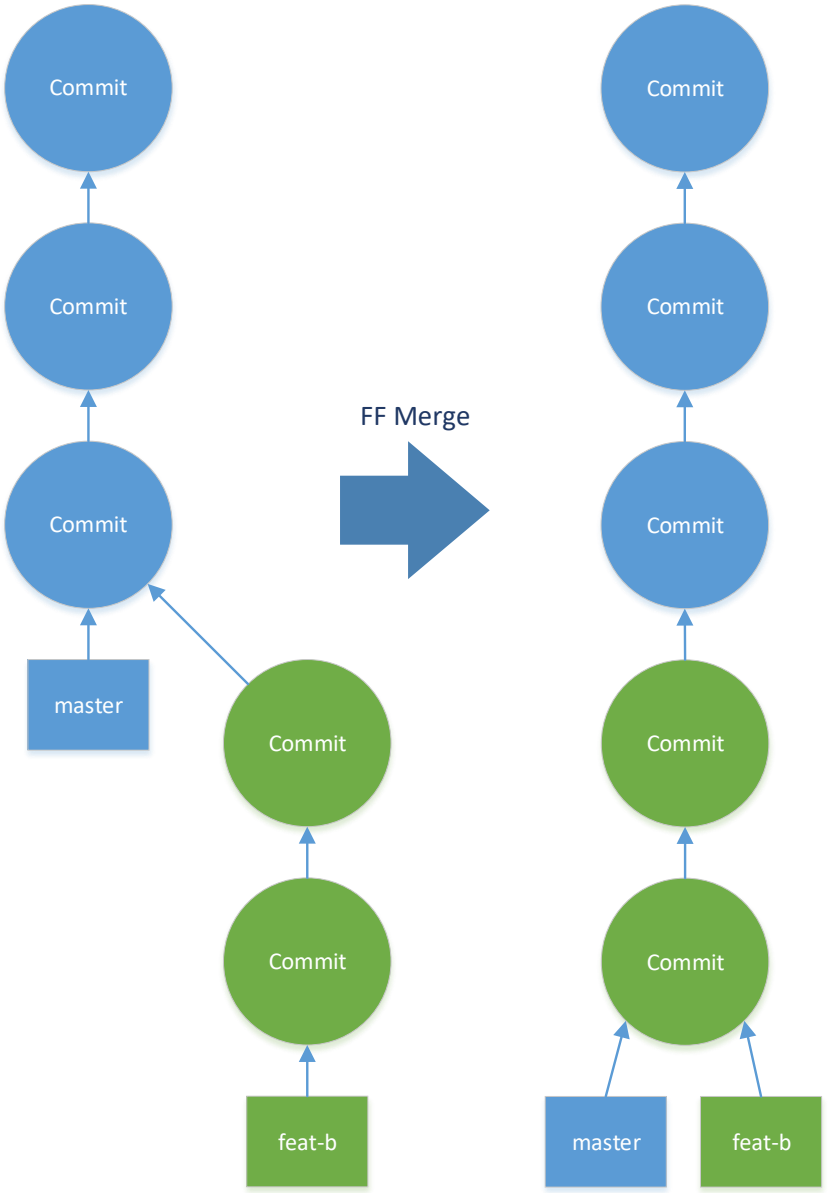
We learned about 3-way Merge…

- Recall that commits are actually SNAPSHOTS.
- In order to do merging, we only need three commits:
  - Common Ancestor: B
  - Tip of current branch: C
  - Tip of target branch: E



We learned about Fast-Forward Merge…

- Another type of merging is the fast-forward merge.
- It is used when the tip of the branch is also the common ancestor.
- **In other words, it is used when history did not really diverge.**

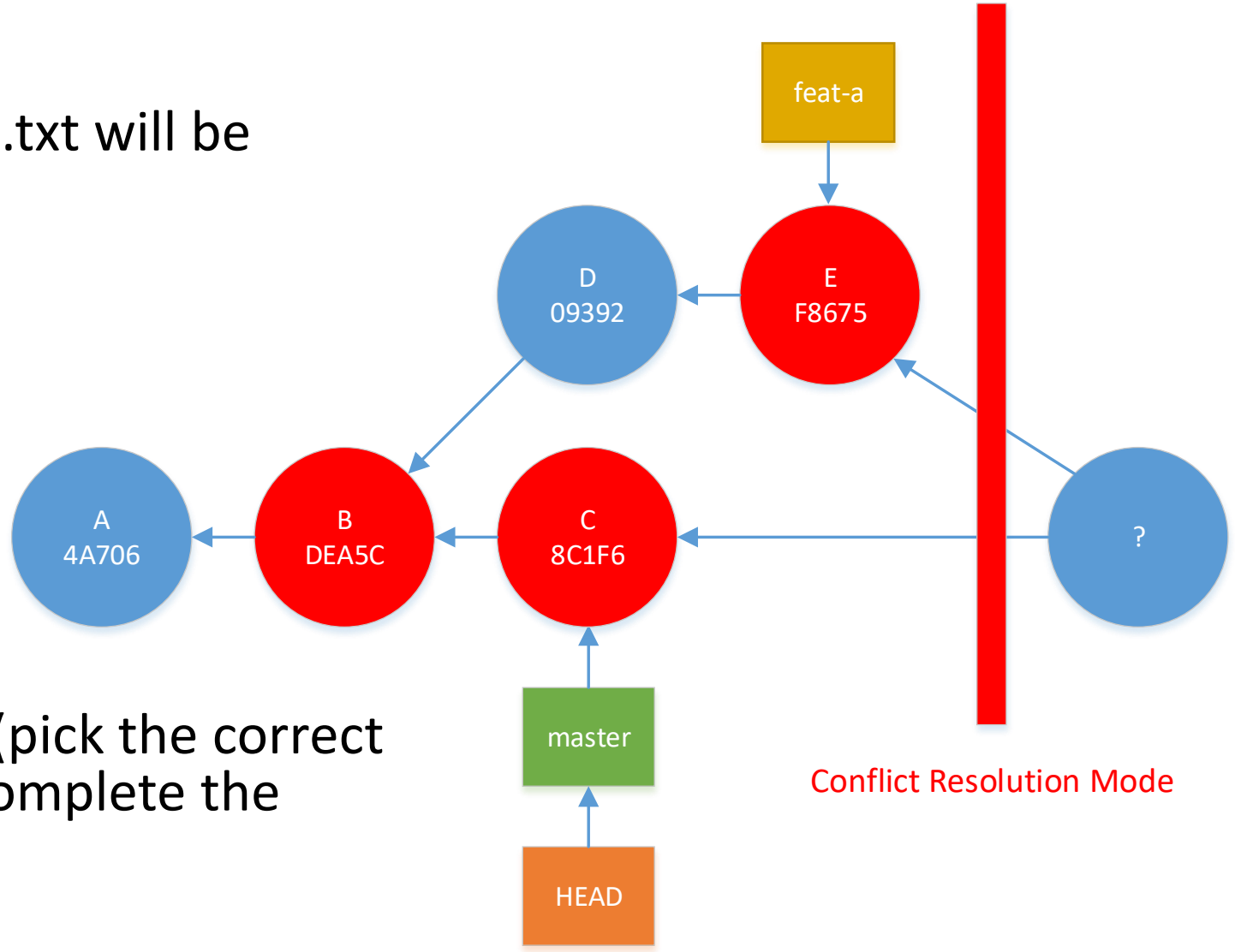


We learned about conflict resolution…

- The content of the conflicting file.txt will be marked as such:

```
line a
line b
line 3
line d
<<<<<<< HEAD
line 5
=======
line E
>>>>>>> feat-a
```

Simply modify the file accordingly (pick the correct changes), stage it and commit to complete the merge.



We learned about remote repositories…

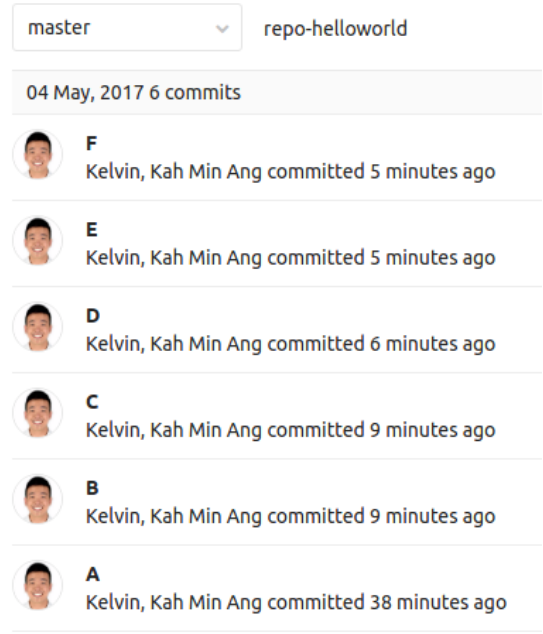
- Next, we push the master branch to the remote repository:

git push origin master

```
angk@angk-Latitude-E5570:~/repo-helloworld$ git remote add origin s
angk@angk-Latitude-E5570:~/repo-helloworld$ git push origin master
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (17/17), 1.20 KiB | 0 bytes/s, done.
Total 17 (delta 5), reused 0 (delta 0)
remote:
remote: ================================================================
remote:
remote:                We Garenians work hard, and play harder!!!
remote:
remote: ================================================================
To ssh://gitlab@git.garena.com:2222/angk/repo-helloworld.git
 * [new branch]      master -> master
```

- You are pushing the "master" branch to the remote repository named "origin".
- Now, if you browse over to the repository in GitLab, you should see your commit:

http://git.garena.com/angk/repo-helloworld/commits/master

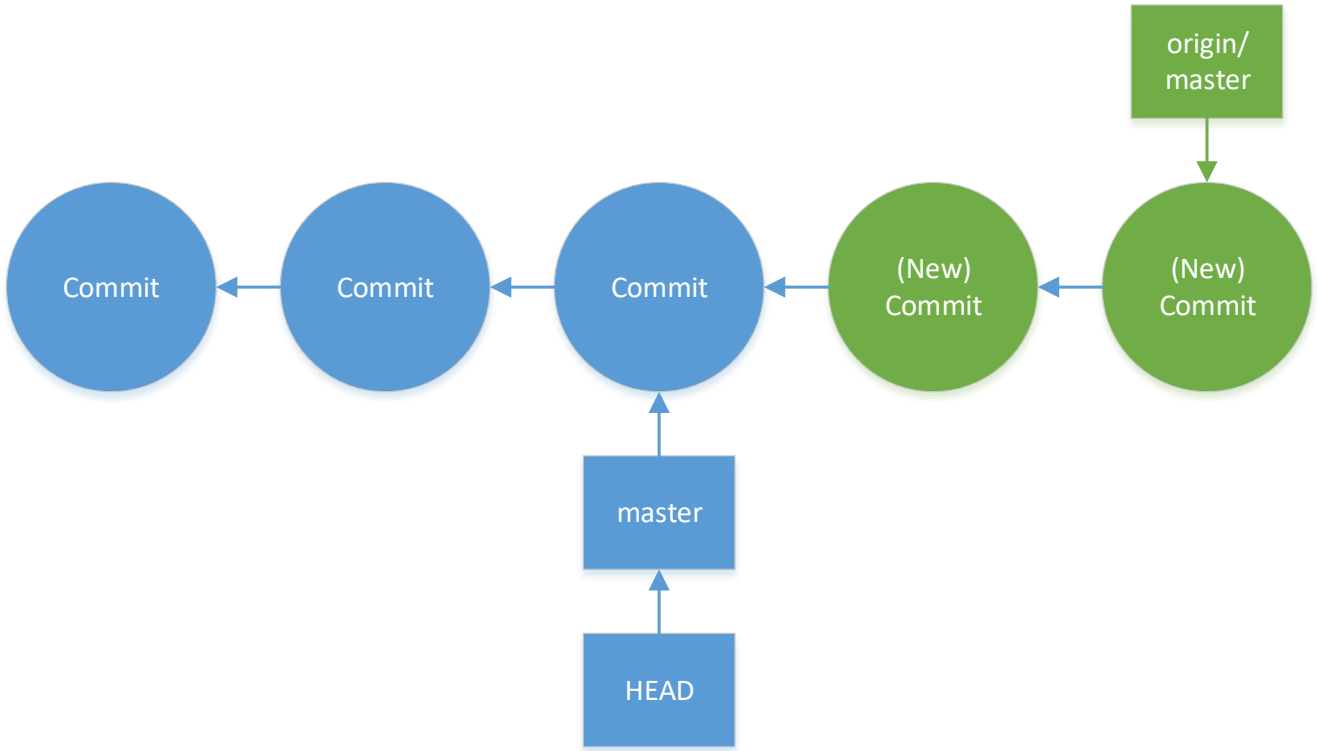


We learned remote tracking branches…

- If you wish to retrieve changes from the remote repository, you can make use of the fetch command:

git fetch origin master

- If there are changes branch in the remote repository, you will end up with something like this:

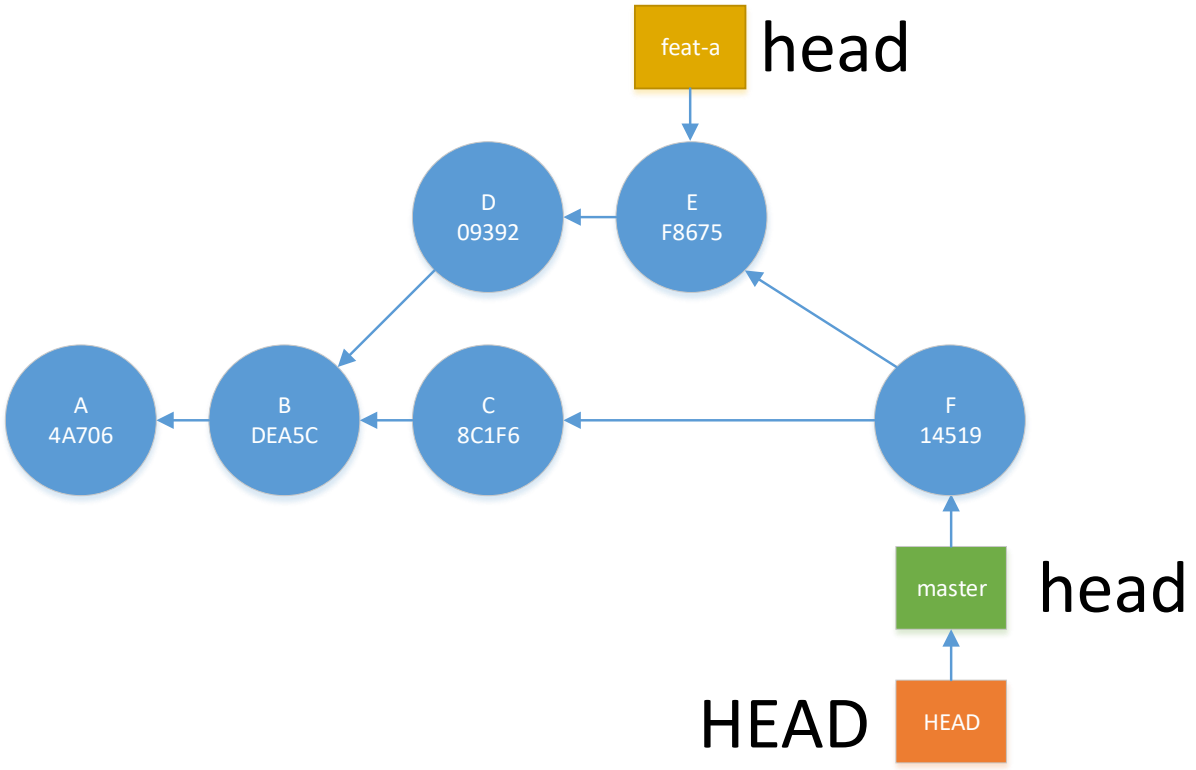


We learned about types of refs...

- There are four main types of refs in Git:
  - **head** - Actually represents the tip of a branch. Each branch has a head.
  - **HEAD** - Points to *WHERE YOU ARE*, which can refer a branch head object, or a commit hash.
  - **tag** - Same as a commit object, just that it refers to a commit instead of a root tree.
  - **remote** - A named git object describing a remote repository.

We learned about types of refs...

- A head is a file which contains the hash value referring to a commit.
- There is a head for each branch, pointing to the tip of the branch.
- A branch is simply a conceptual idea, and is physically represented by a pointer to the last commit done while in the branch.
- The default branch, the master branch, has its head stored in .git/refs/heads/master.



head = Branch Tip Pointer

HEAD = "You are here now"

We learned about submodules...

GitLab

Parent Commit	THREE
Submodule Commit	B

Dev A

Parent Commit	TWO
Submodule Commit	B
Checked-out Submodule Commit	B

Dev B

THREE	Parent Commit
B	Submodule Commit
B	Checked-out Submodule Commit

Lesson:
Always "**git status**"
before "git add ."
or "git commit -a"

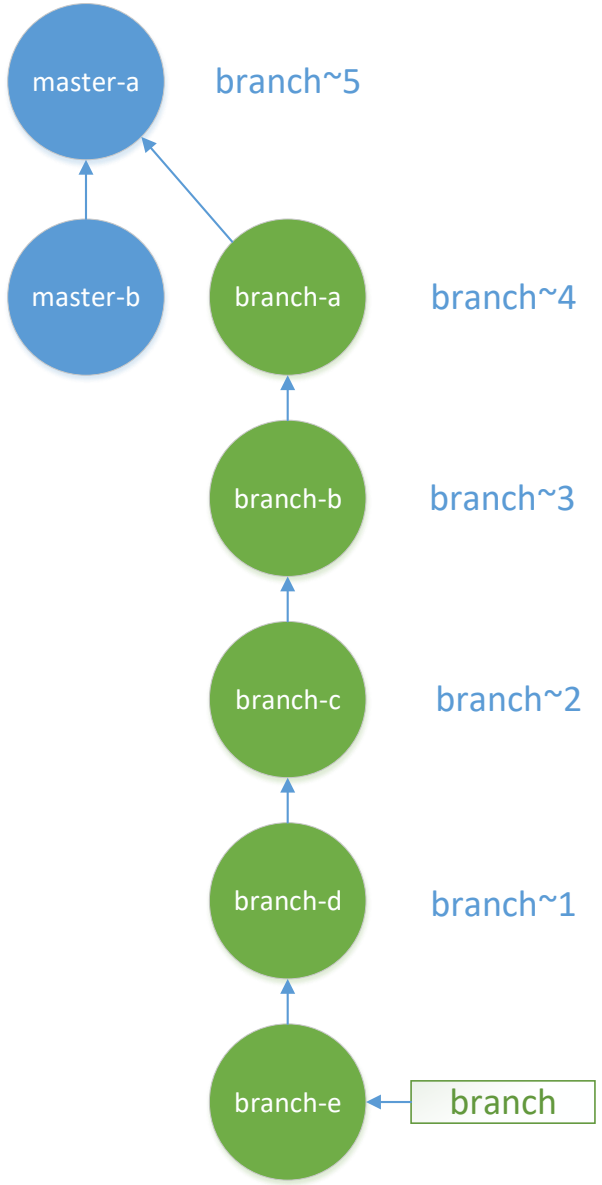
Cherry-picking

Picking cherries from other branches

Cherry-picking

- Suppose that you have a feature branch, but you realized that you only want a few of the changes you made.

- e.g. **You only want:**
  - **branch-a**
  - **branch-d**
  - **branch-e**

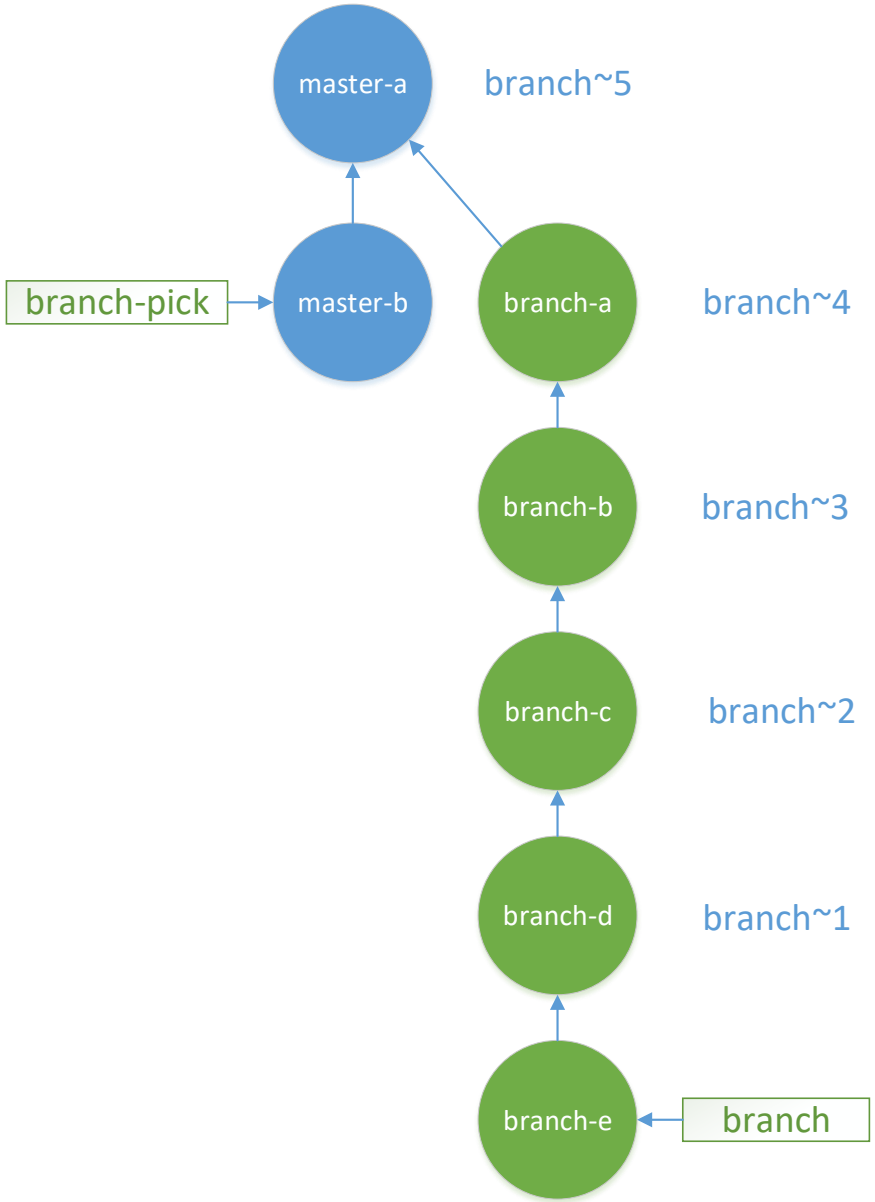


Cherry-picking

- First, you checkout a new branch from the destination branch.

git checkout master

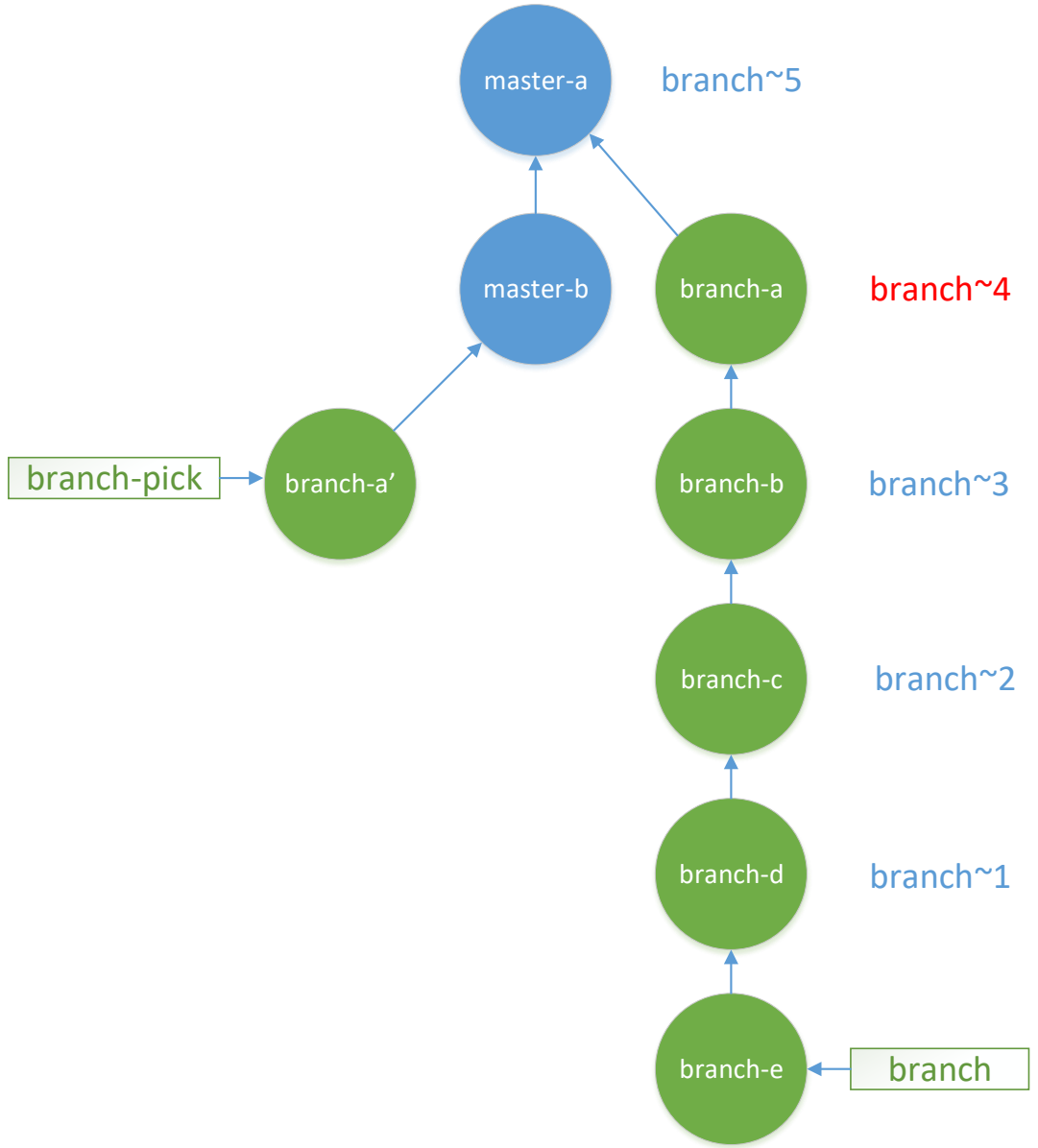
git checkout -b branch-pick



Cherry-picking

- Then you pick your desired commits from the source branch:

git cherry-pick branch~4

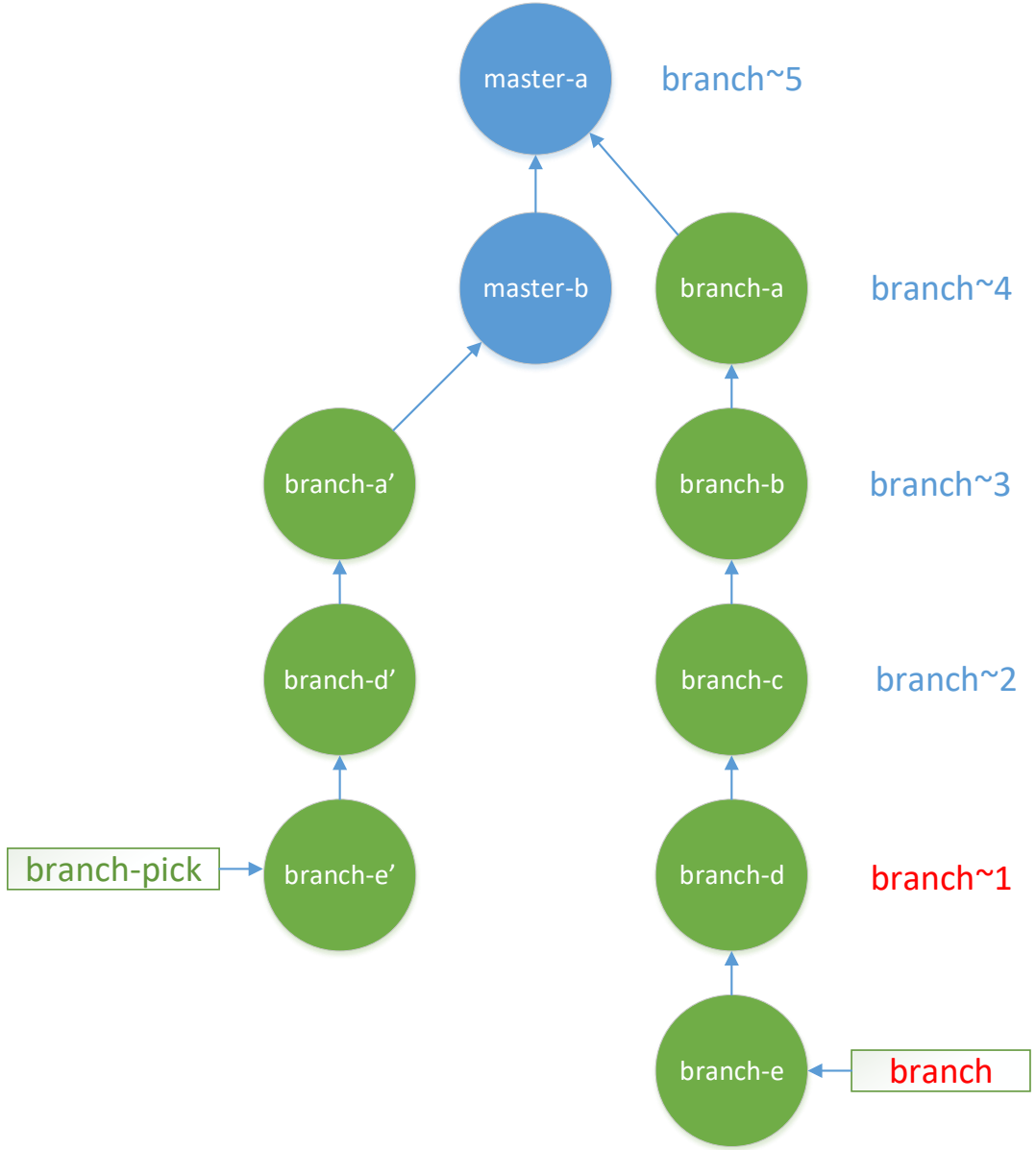


Cherry-picking

- You can also pick a range:

git cherry-pick branch~2..branch

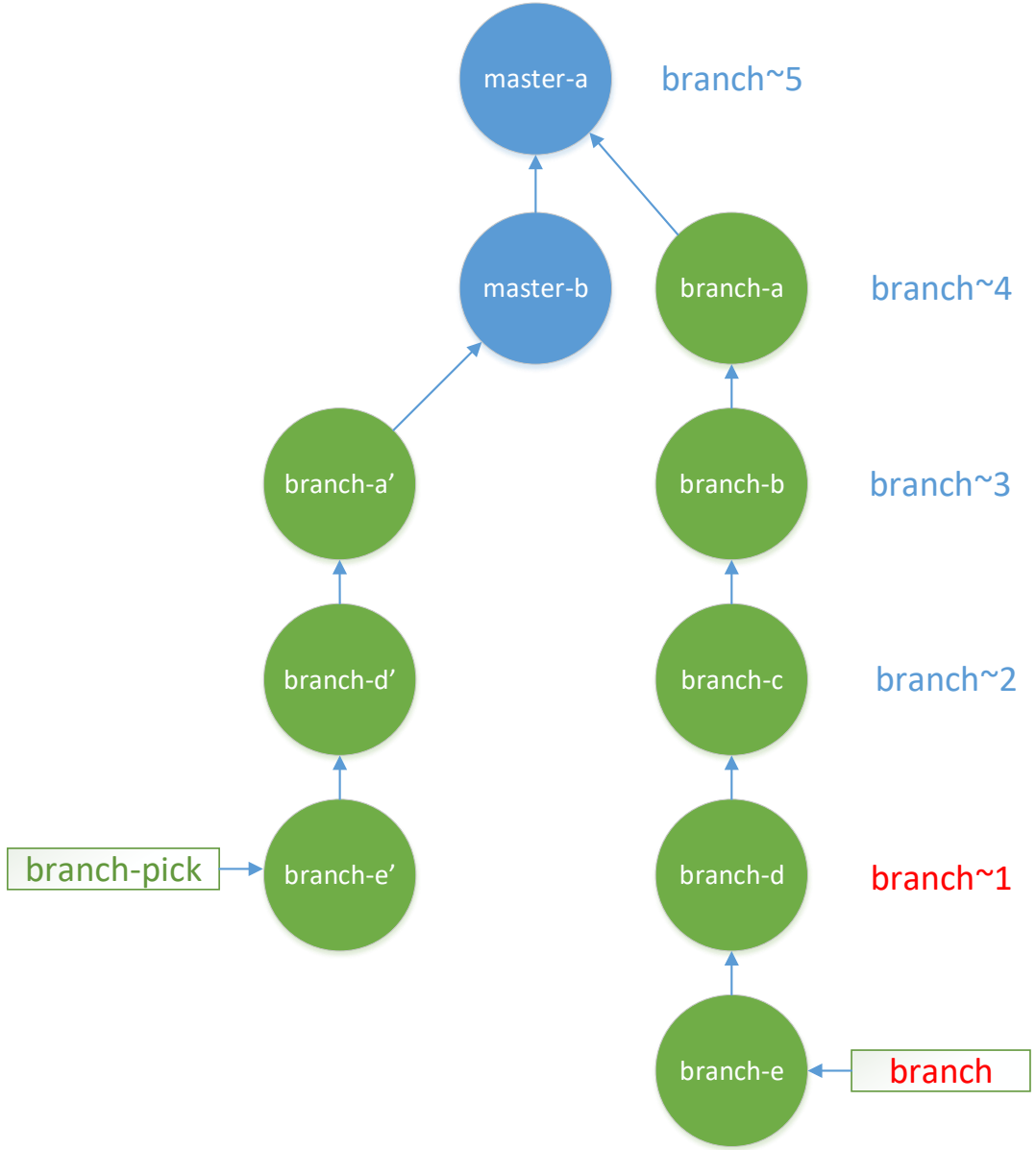
(Range excludes the first commit and includes the last commit)



Cherry-picking

- Note that the new commits are **completely different commits!**

- Cherry-pick works by **picking out the DIFFS (or patch)** introduced by a particular commit and applying it to your current commit.



Cherry-picking Cheatsheet

- To cherry-pick a commit into the current branch, use:

git cherry-pick <commit-name>

The commit name can be a branch, a commit hash, or a range.

- To cherry pick a single commit A, use:

git cherry-pick A

- To cherry-pick the last commit from a branch feat-a, use:

git cherry-pick feat-a

Cherry-picking Cheatsheet

- To cherry-pick the last three commits from a branch feat-a, use:

git cherry-pick feat-a~3..feat-a

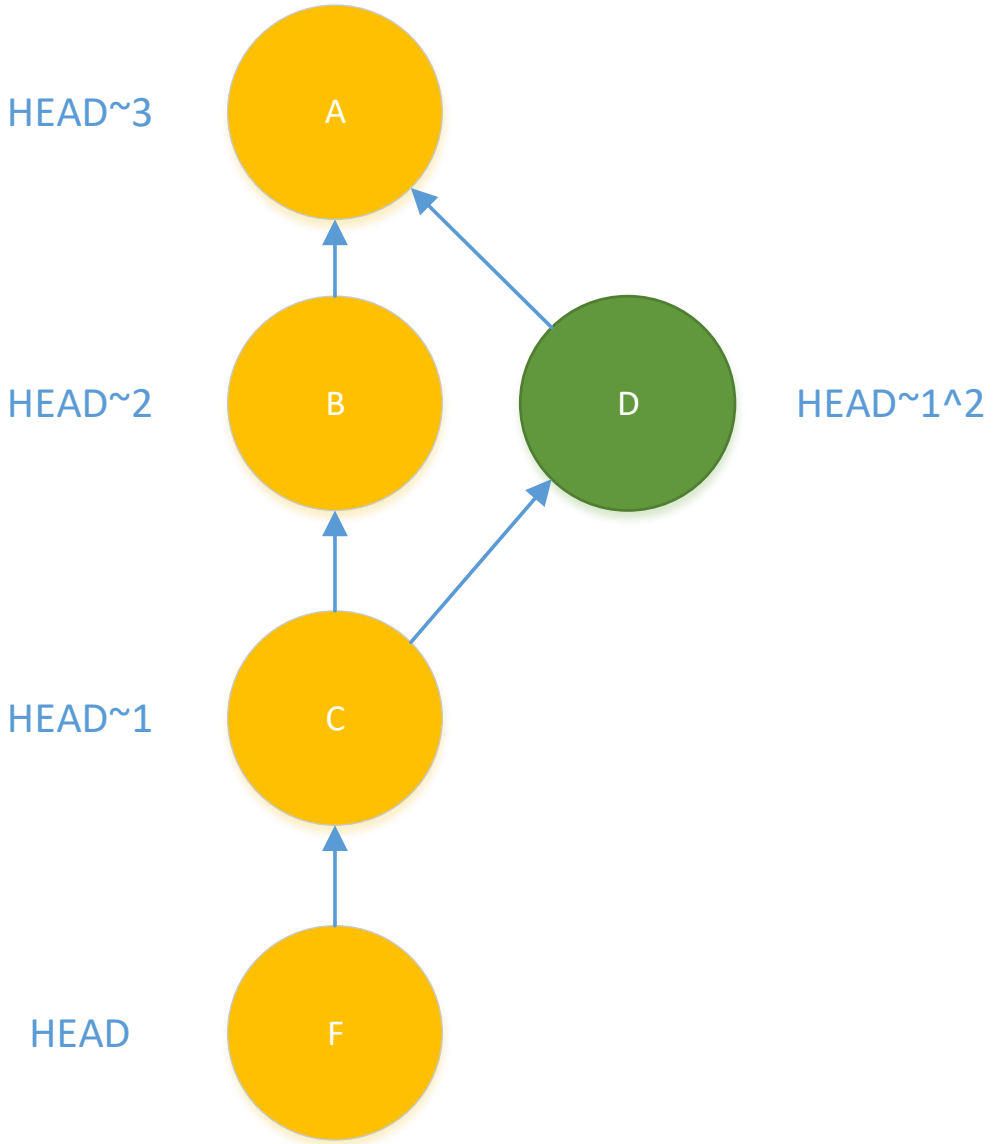
Note that in a range, the first mentioned commit must be earlier than the second.

git cherry-pick <older-commit>..<newer commit>

(The older-commit is excluded)

Cherry-picking Cheatsheet

- As you can see, ~ will always follow the main branch, while ^ allows you to switch branches.
- This applies to most refs (commits, branches, tags, etc.)



Cherry-picking Cheatsheet

- Tilde (~) vs Caret (^) when specifying relative commits…

- Tilde (~):
  - feat-a~1 means feat-a's first parent.
  - feat-a~2 means feat-a's first parent's first parent.
  - feat-a~3 means feat-a's first parent's first parent's first parent.

- Caret (^):
  - feat-a~1 means feat-a's first parent.
  - feat-a~2 means feat-a's second parent.
  - feat-a~3 means feat-a's third parent.

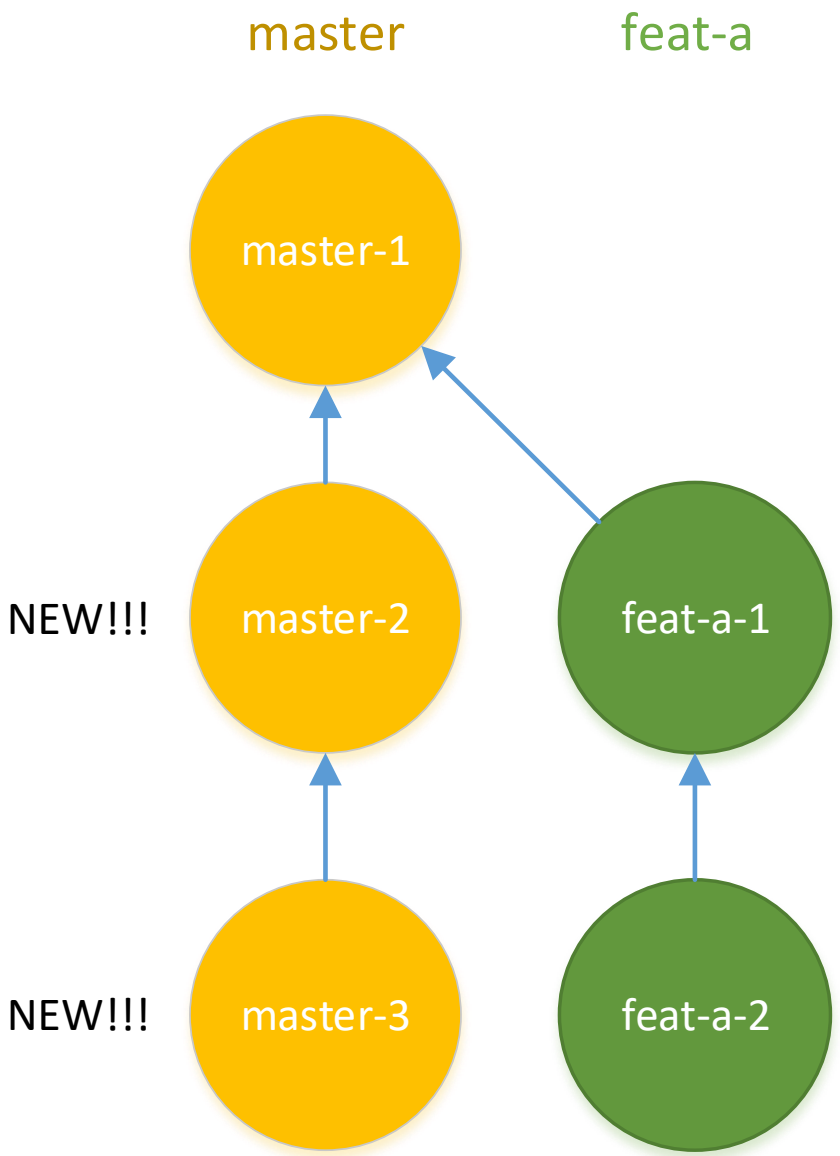
- Combination:
  - feat-a~1^2 means feat-a's first parent's second parent. This is useful when traversing merges.

Rebasing

Changing your common ancestor, or your "branching point"

Rebasing

- Suppose that during development you found that you actually needed to include changes that were introduced in the master branch.
- **Rebasing allows you to "change" the common ancestor.**



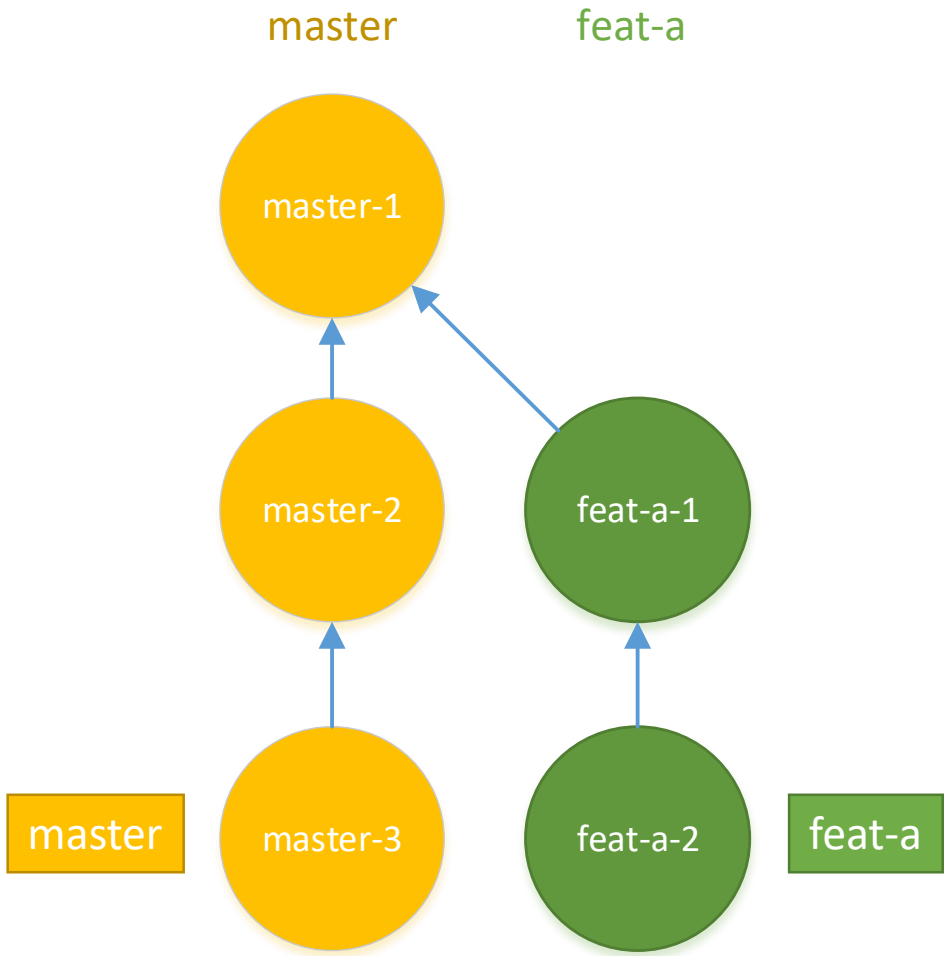
Useful Features - Rebasing

- To rebase feat-a from master-1 to master-3, use:

git checkout feat-a

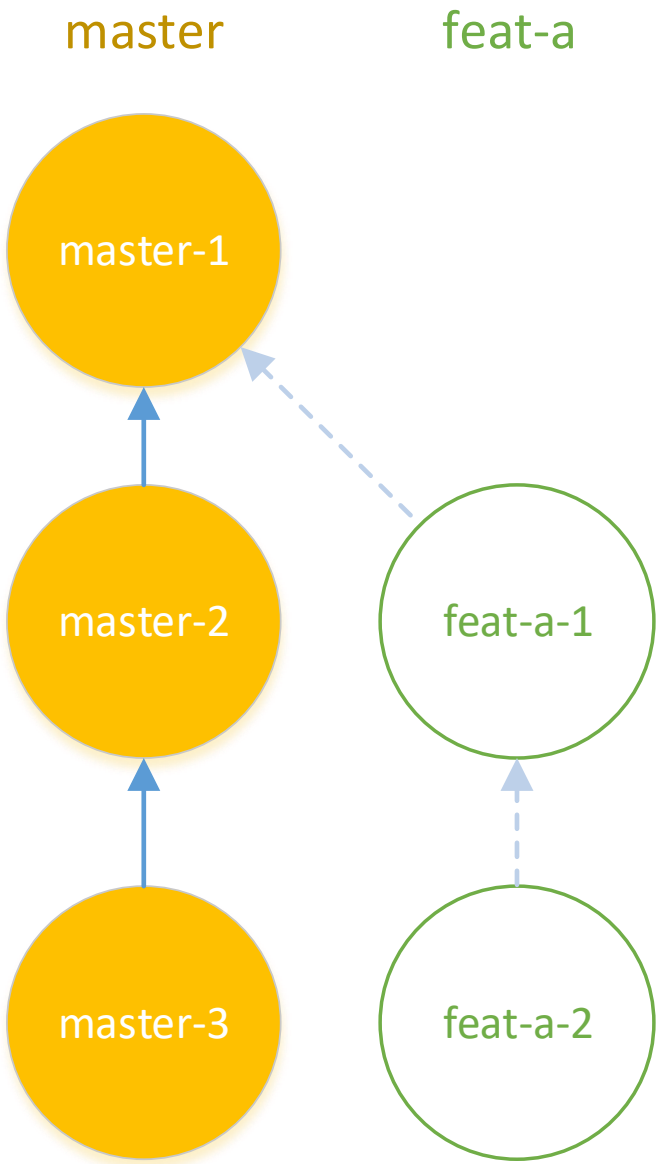
git rebase master

In other words, go to your branch, then rebase to master.



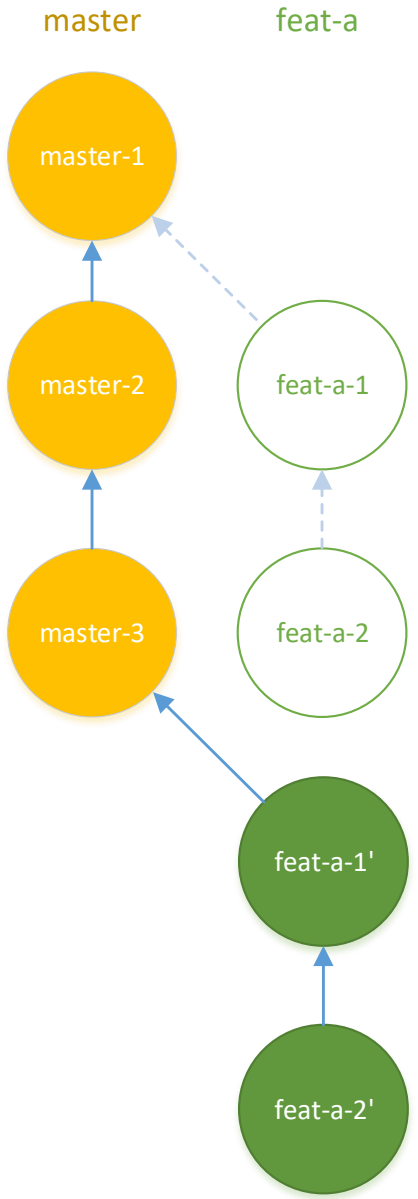
Useful Features - Rebasing

- **Your work will be rewound to the common ancestor, which is master-1.**
- The changes will then be replayed onto master-3.
- Notice that the commits are now completely new ones, but they retain the commit messages etc.



Useful Features - Rebasing

- Your work will be rewound to the common ancestor, which is master-1.
- **The changes will then be replayed onto master-3.**
- Notice that the commits are now completely new ones, but they retain the commit messages etc.



Rebasing Cheatsheet

- To rebase your current branch to another branch:

git rebase <commit/branch>

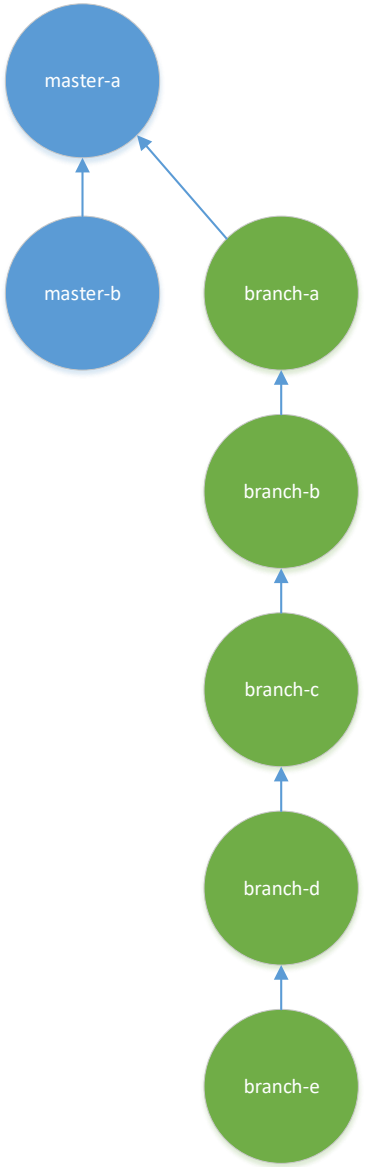
- Rebasing works similarly to cherry-picking.
- It actually "rewinds" your work back to the common ancestor.
- And then replay all your work on the specified branch.

Squashing

Squishing 10 commits into one!

Squashing

- Suppose that you made 10 commits, but you realized that all your changes could be better represented by just a single commit.
- This is where squashing comes in!

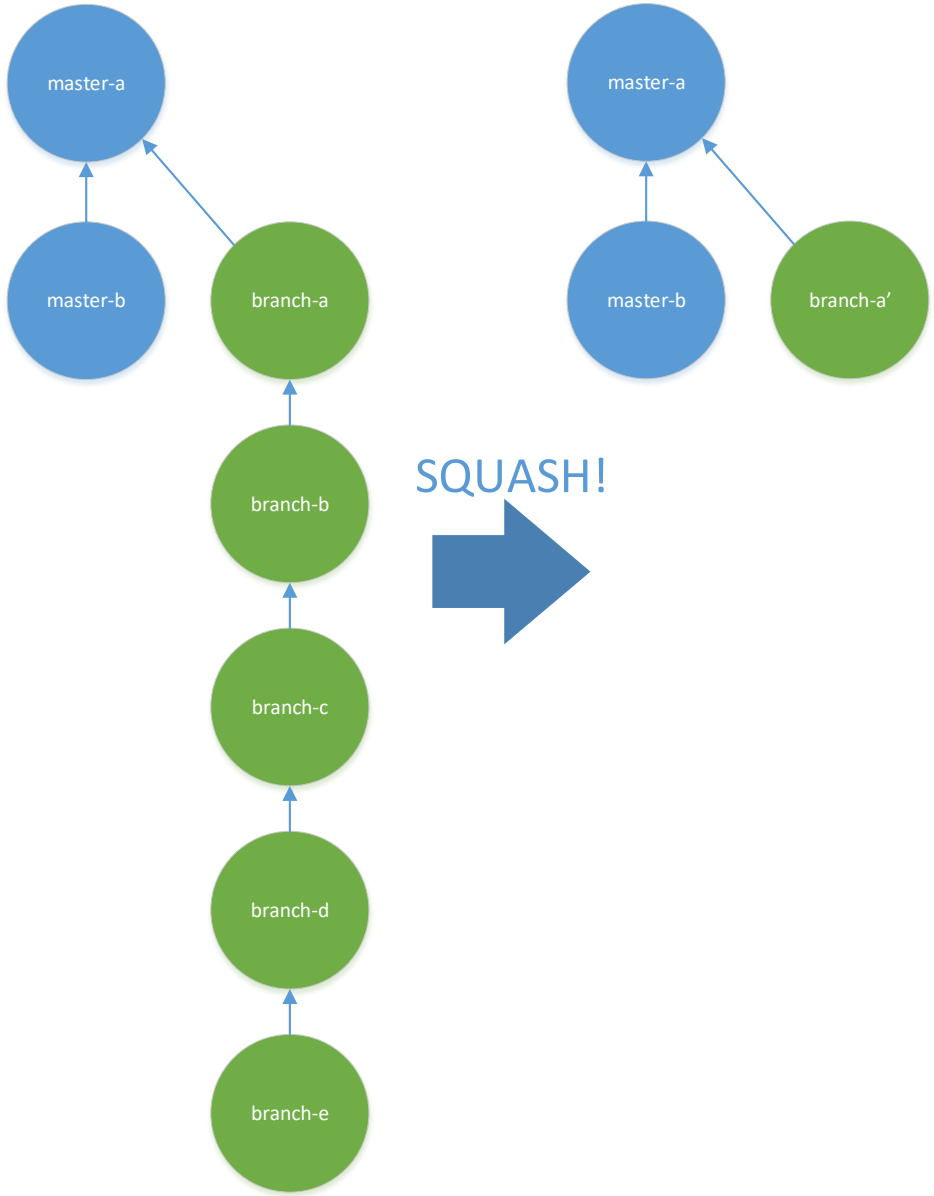


Too many unnecessary commits!

Squashing

- We want to achieve something similar to the following:

- **Warning: Please do not squash changes that you have already pushed or merged!** You are rewriting history which can result in a lot of strange issues.



Squashing

- To squash the last n commits, use:

```
git rebase -i HEAD~n
```

This is an interactive rebase which will open up your text editor. You can squash commits by replacing "pick" to "squash" and then saving and exiting your text editor.

For instance, if you wish to squash all commits into A:

Before	After
pick A pick B pick C pick D pick E	pick A **squash** B **squash** C **squash** D **squash** E

Squashing

- Alternatively, we can use soft reset:

```
git reset --soft HEAD~n
git commit
```

- This will reset your head to n commits away, keeping everything staged.

Reset Types

Type	MOVE head/HEAD	UNSTAGE	REVERT FILES
--soft	✓		
--mixed	✓	✓	
--hard	✓	✓	✓

--soft will only move the head (branch pointer) and HEAD pointer to the specified commit
--mixed will additionally revert the index, which **unstages** the files
--hard will additionally **revert** the files

Reset Types

Type	MOVE head/HEAD	UNSTAGE	REVERT FILES
--soft	✓		
--mixed	✓	✓	
--hard	✓	✓	✓

--soft will only move the head (branch pointer) and HEAD pointer to the specified commit
--mixed will additionally revert the index, which **unstages** the files
--hard will additionally **revert** the files

Q: What is the difference between **git checkout** and **git reset** then?

git checkout <commit>
git reset --soft <commit>

Squashing Cheatsheet

- To squash the last n commits, use:

```
git rebase -i HEAD~n
```

This is an interactive rebase which will open up your text editor. You can squash commits by replacing "pick" to "squash" and then saving and exiting your text editor.

For instance, if you wish to squash all commits into A:

Before	After

`pick A`
`pick B`
`pick C`
`pick D`
`pick E`
	`pick A`
	`squash` `B`
	`squash` `C`
	`squash` `D`
	`squash` `E`

Squashing Cheatsheet

- Alternatively, we can use soft reset:

```
git reset --soft HEAD~n
git commit
```

- This will reset your head to n commits away, keeping everything staged.

reflog

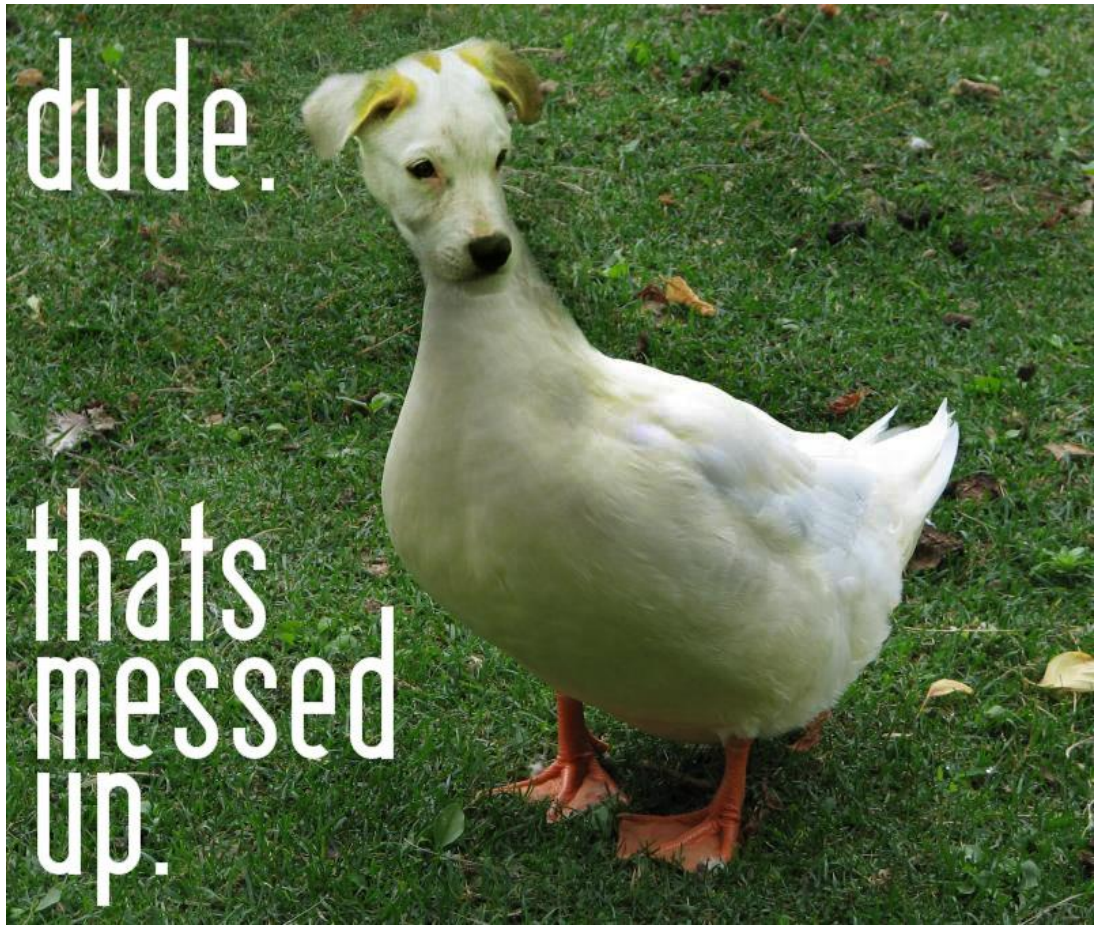
Git Reference Log.

Behind the scenes version control of your version control system.

Reference Log

- Rebasing and resets are useful…
  Until you mess something up.

- Rebase accidents
- Reset accidents
- Detached head accidents



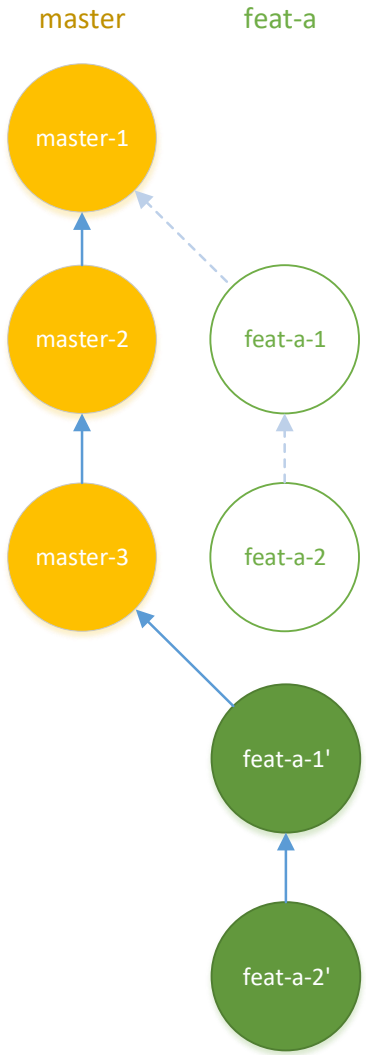
Reference Log

- Recall that rebasing creates entirely new commits…
- And reset simply moves your HEAD pointer around.

Type	MOVE head/HEAD	UNSTAGE	REVERT FILES
--soft	✓		
--mixed	✓	✓	
--hard	✓	✓	✓

Reset does not delete commits.

Rebase does not delete commits.

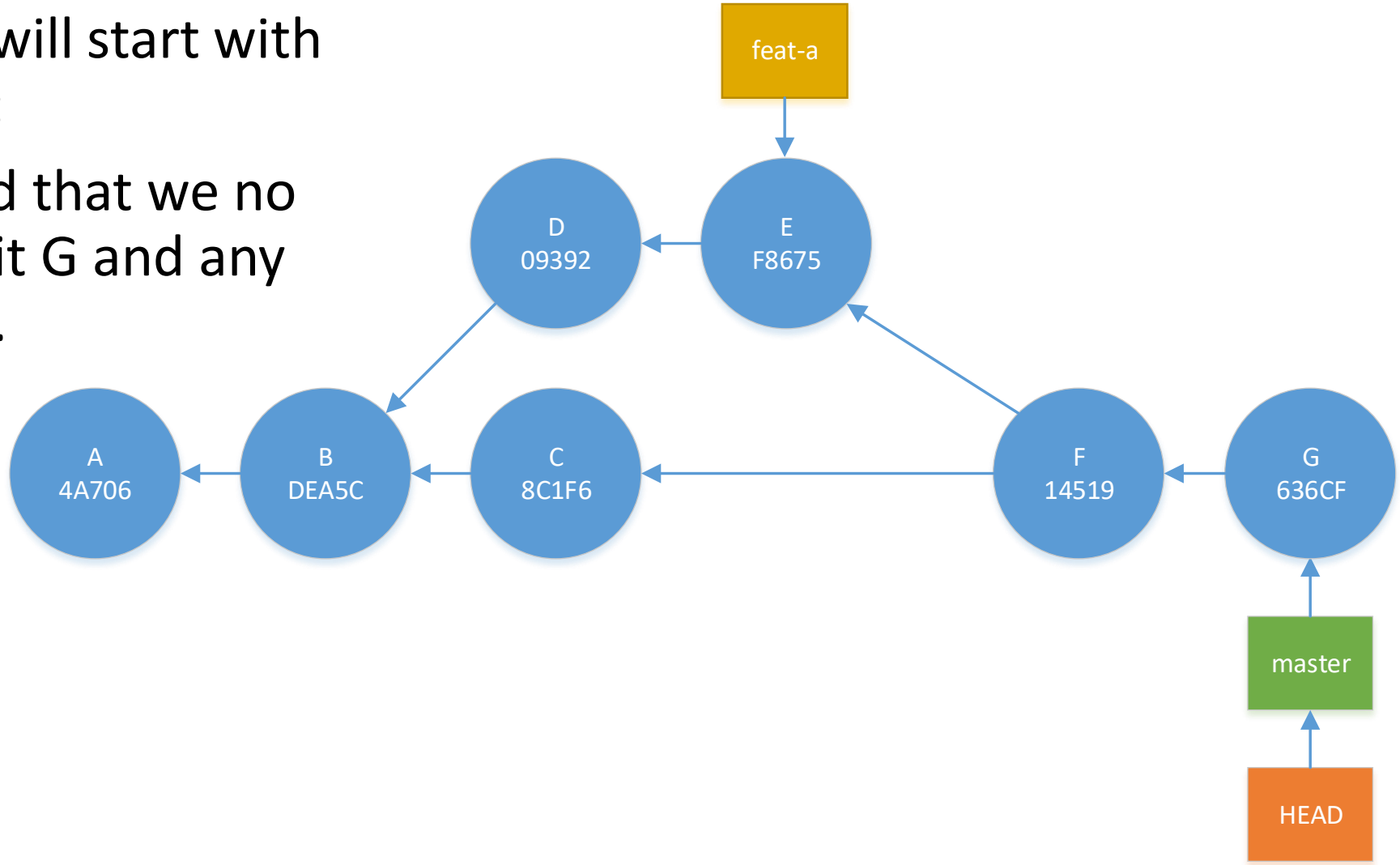


Reference Log

- Objects will stay around until they are garbage collected when **they are no longer referenced anywhere.**
- Your commits are still around somewhere.
- Where can we find it?

Reference Log

- In this section, we will start with the following state:
- Lets say we decided that we no longer want commit G and any changes made in it.

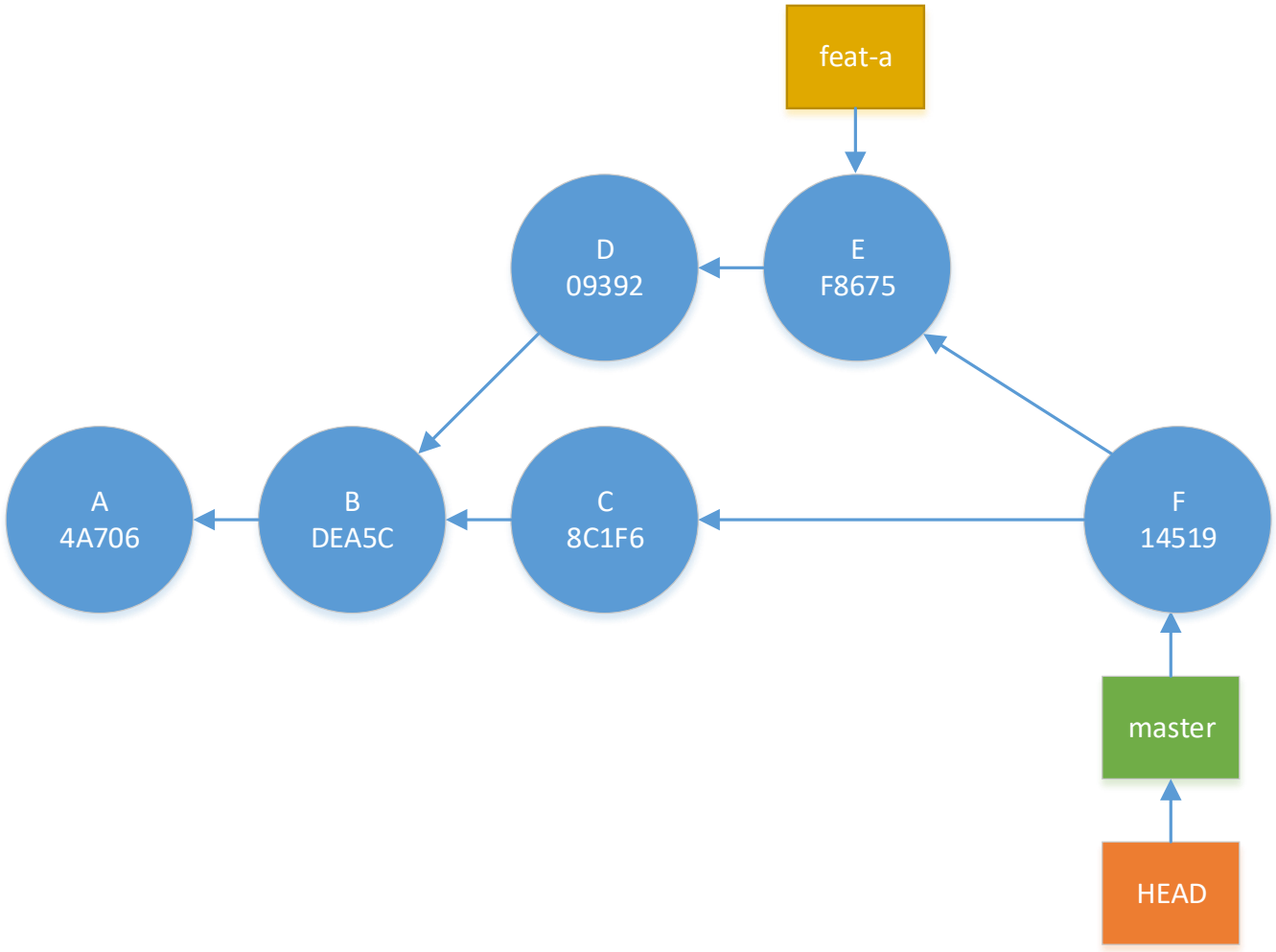


Reference Log

- Lets say we decided that we no longer want commit G and any changes made in it.
- We roll back to a previous commit:

git reset --hard HEAD^

Q: What does --hard do?



Reference Log

- If you try to look in the lost-and-found, you will find a dangling commit:

git fsck --lost-found

Dangling objects are objects that are not directly reachable by any branch or tag.

```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git fsck --lost-found
Checking object directories: 100% (256/256), done.
dangling commit 54d7b7130237ea9979b85b36a38016ead135ee60
```

Reference Log

- To see the reference log, we type:

git reflog

This shows us a history of where HEAD was at.

```
Checking object directories: 100% (256/256), done.
dangling commit 54d7b7130237ea9979b85b36a38016ead135ee60
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git reflog
8bf42a8 HEAD@{0}: reset: moving to HEAD^
54d7b71 HEAD@{1}: commit: G
8bf42a8 HEAD@{2}: merge feat-a: Merge made by the 'recursive' strategy.
71dda49 HEAD@{3}: checkout: moving from feat-a to master
ccf27b1 HEAD@{4}: commit: E
883e76a HEAD@{5}: commit: D
0c1dcd7 HEAD@{6}: checkout: moving from 0c1dcd7d16b13de0ea55062b561be7a5d0c79c3f to feat-a
0c1dcd7 HEAD@{7}: checkout: moving from master to 0c1dcd
71dda49 HEAD@{8}: commit: C
0c1dcd7 HEAD@{9}: commit: B
1971706 HEAD@{10}: commit (initial): A
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$
```

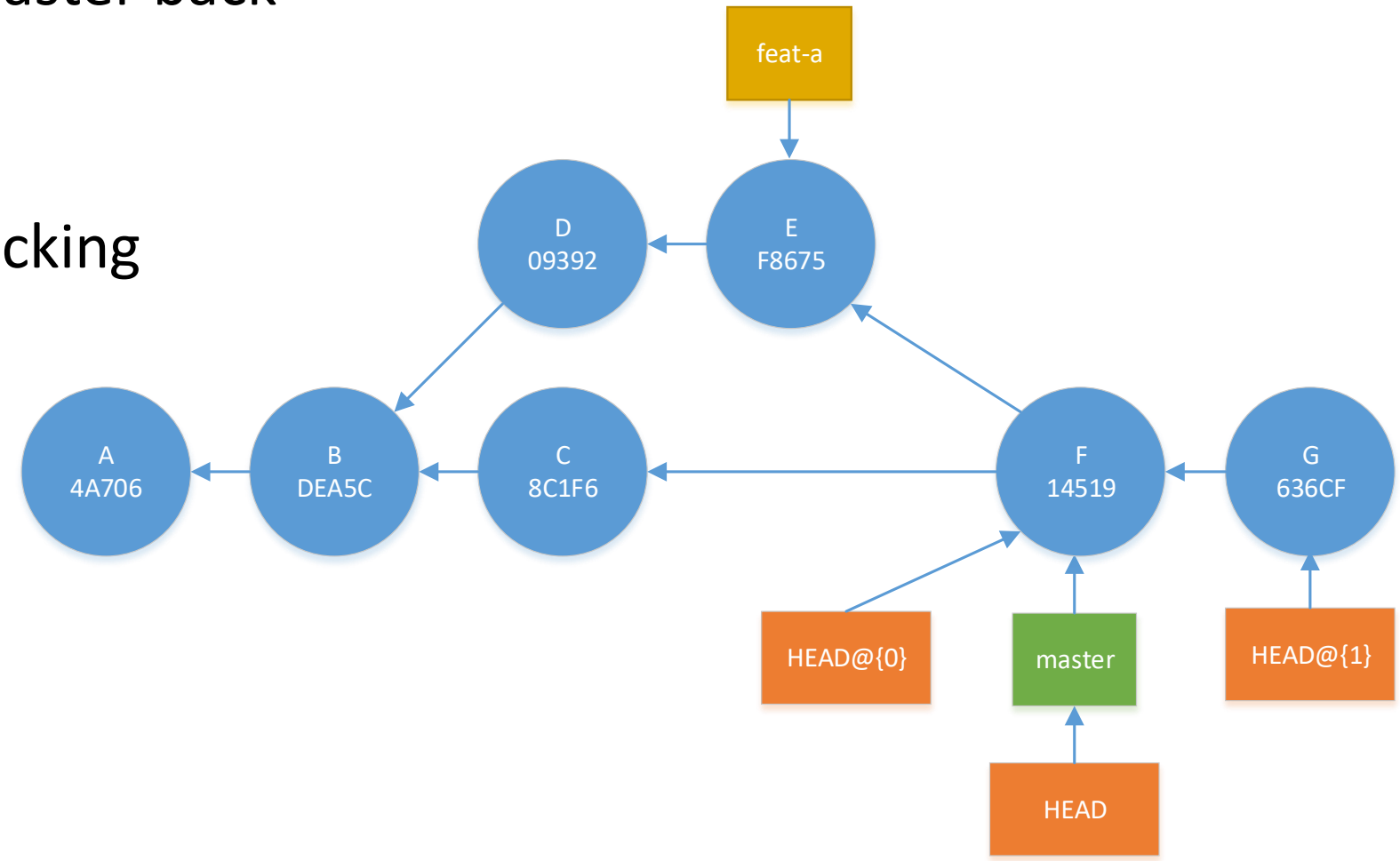
Reference Log

- HEAD@{0} means where HEAD is now.
- HEAD@{1} means where HEAD is 1 operation ago.
- HEAD@{2} means where HEAD is 2 operations ago
- …etc.

```
Checking object directories: 100% (256/256), done.
dangling commit 54d7b7130237ea9979b85b36a38016ead135ee60
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git reflog
8bf42a8 HEAD@{0}: reset: moving to HEAD^
54d7b71 HEAD@{1}: commit: G
8bf42a8 HEAD@{2}: merge feat-a: Merge made by the 'recursive' strategy.
71dda49 HEAD@{3}: checkout: moving from feat-a to master
ccf27b1 HEAD@{4}: commit: E
883e76a HEAD@{5}: commit: D
0c1dcd7 HEAD@{6}: checkout: moving from 0c1dcd7d16b13de0ea55062b561be7a5d0c79c3f to feat-a
0c1dcd7 HEAD@{7}: checkout: moving from master to 0c1dcd
71dda49 HEAD@{8}: commit: C
0c1dcd7 HEAD@{9}: commit: B
1971706 HEAD@{10}: commit (initial): A
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$
```

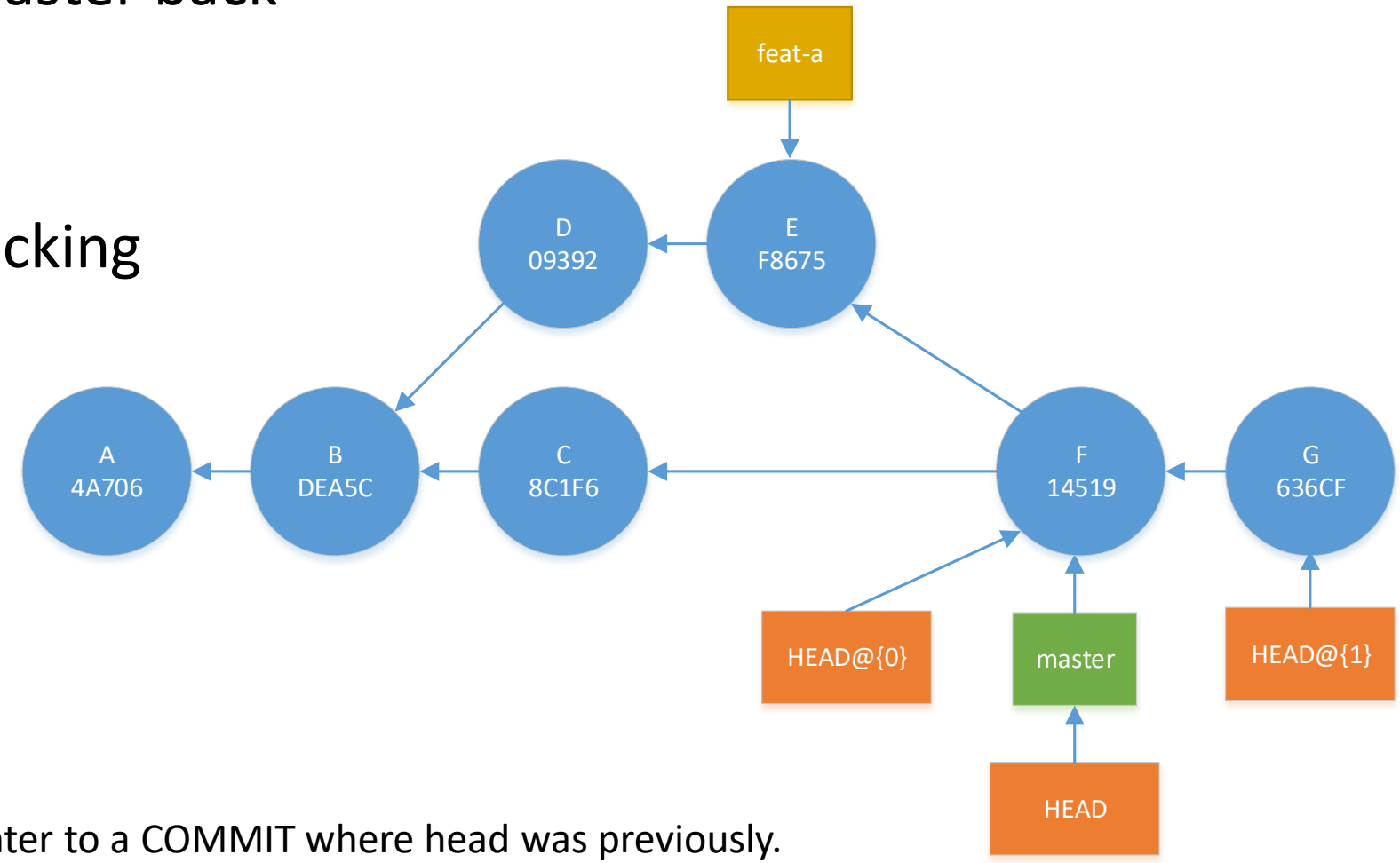
Reference Log

- Q: How do we move master back to HEAD@{1}?
- Clue: Recall remote tracking branches…



Reference Log

- Q: How do we move master back to HEAD@{1}?
- Clue: Recall remote tracking branches…

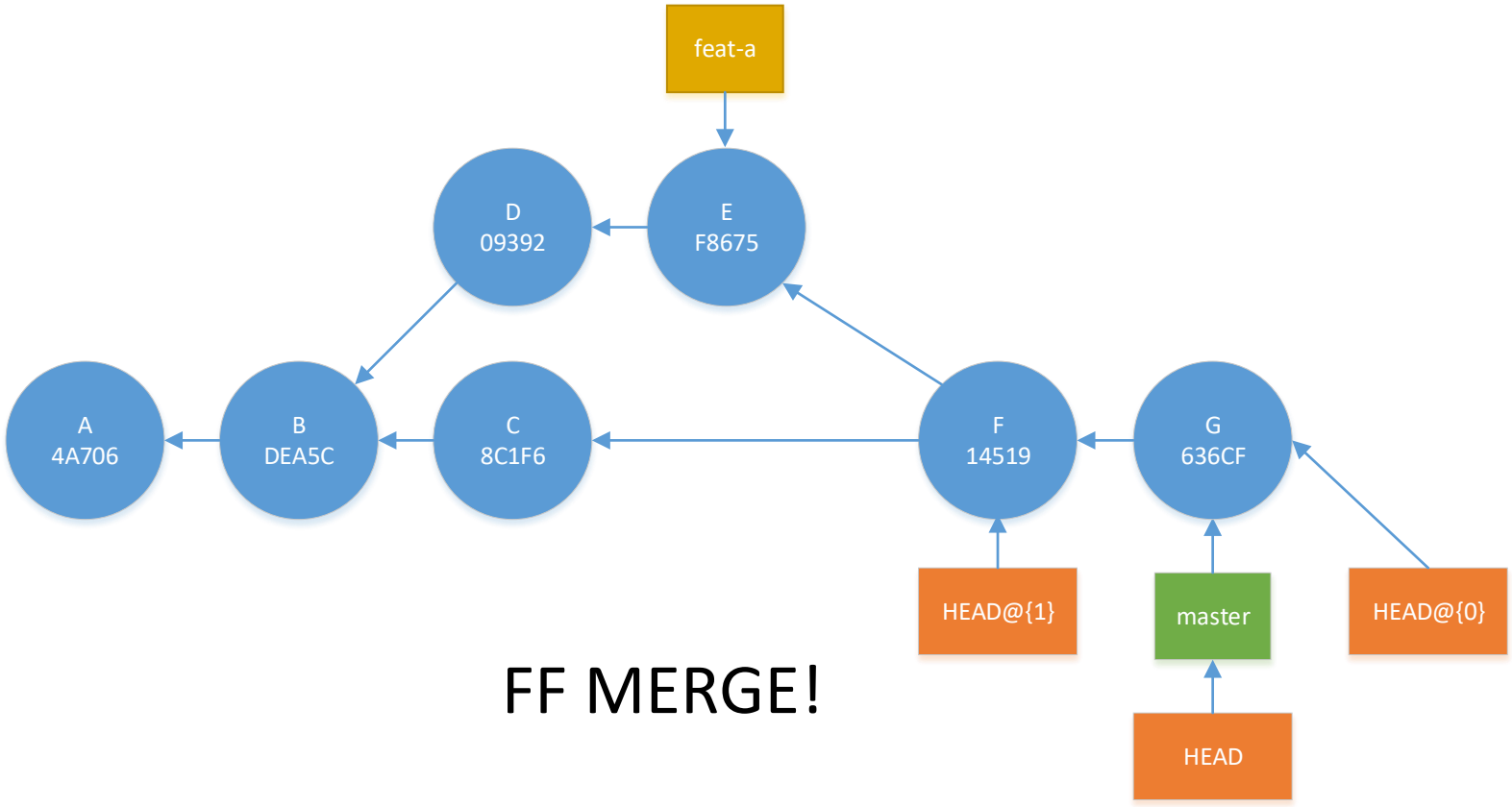


Note: HEAD@{0} is not HEAD. It is just a pointer to a COMMIT where head was previously.

Reference Log

- We simply do a fast forward merge:

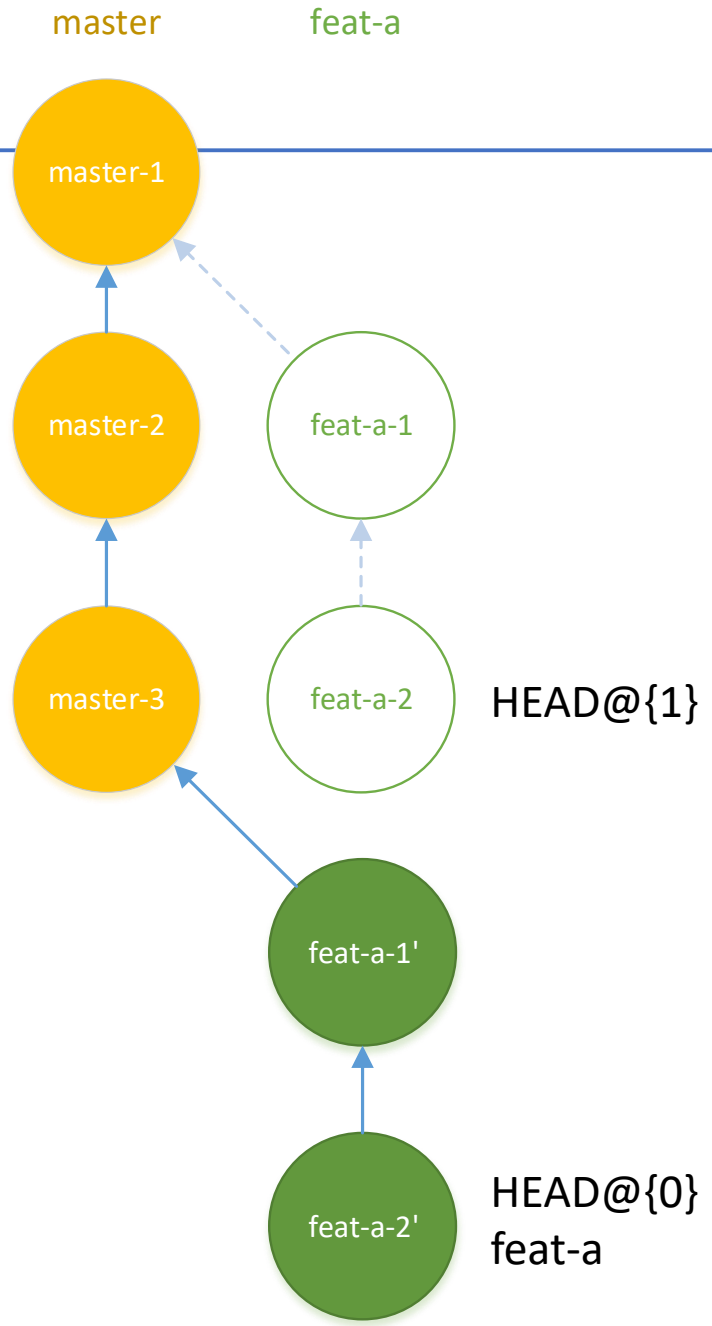
git merge HEAD@{1}



FF MERGE!

Reference Log

- Let's discuss… How do we undo something like this?

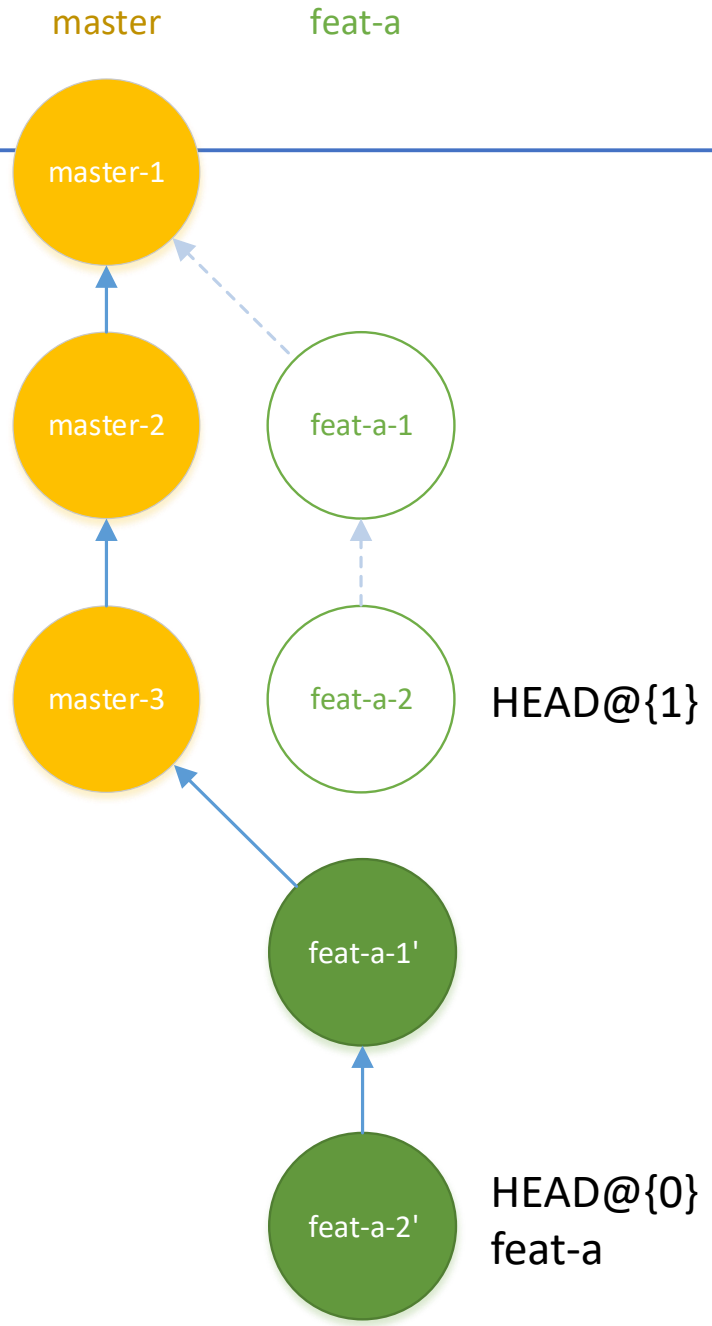


Reference Log

- Let's discuss… How do we undo something like this?

- Instead of switching around and using FF merging, you can use:

git reset --hard HEAD@{1}



Reference Log Cheatsheets

- Find dangling objects:

git fsck --lost-and-found

- Clear all dangling objects:

git reflog expire --expire-unreachable=now –all

git gc --prune=now

Reflog still references the objects, so they won't be garbage collected. The first command expires these reflog references so GC can clear them in the second command.

Reference Log Cheatsheets

- Check the reference log:

git reflog

- To recover a previous state of your branch:

git reset <common ancestor>

git merge <reference from reflog>

sea
connecting the dots

Altering History

And some related hazards ☹

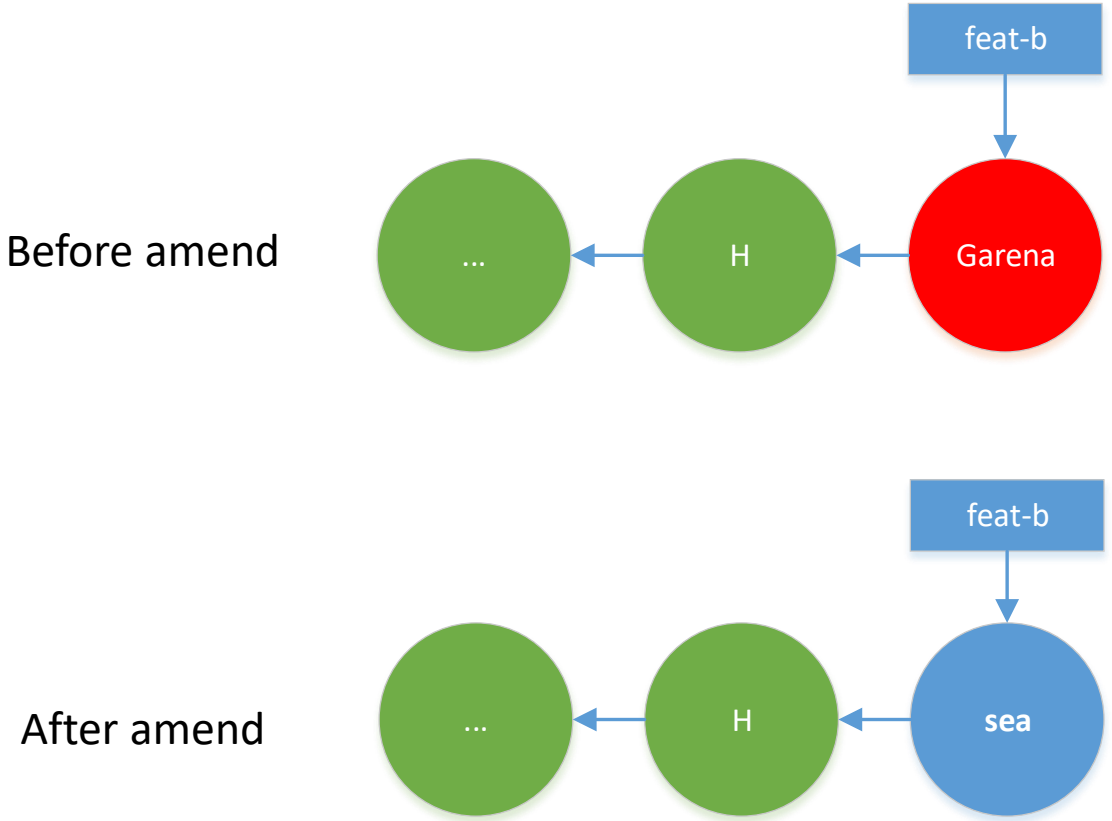
Altering History

- Sometimes you can make changes that alters the history of a branch:
  - Renaming a commit
  - Rebasing
  - Squashing

Altering History

- Suppose that we have a commit named "Garena"
- A shortcut to rename the previous commit is:

git commit --amend




```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git commit --amend
[feat-b 663d60d] sea
 Date: Tue Jun 6 12:47:04 2017 +0800
 1 file changed, 1 insertion(+)
 create mode 100644 testfile7
```

Altering History

- Notice that renaming the commits completely changes the commit hash (its "description" has changed, so its hash is recalculated).

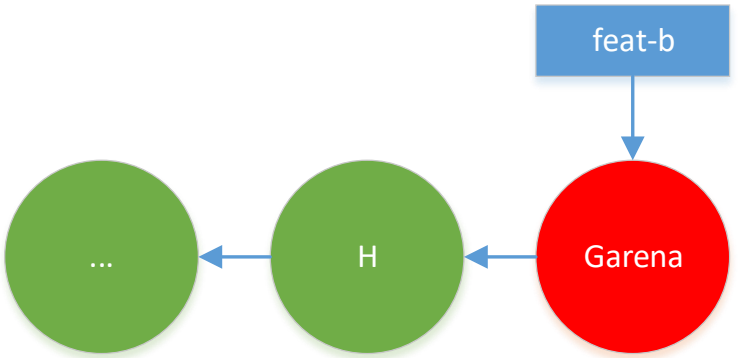
```
commit 363cb26c88b150acda4f6120ae64fa6e882cccc3
Author: Ang Kah Min, Kelvin <angk@garena.com>
Date:   Tue Jun 6 12:47:04 2017 +0800

    Garena
```

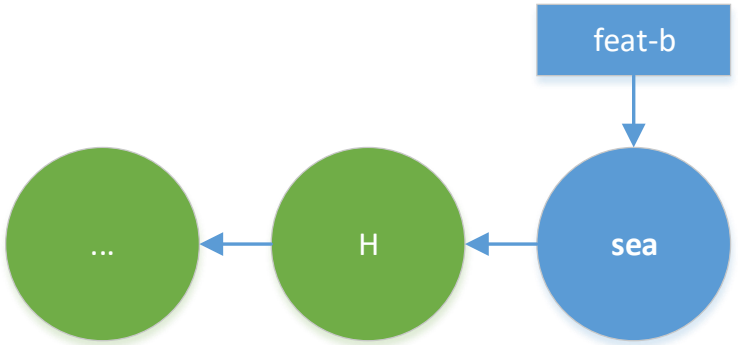
```
commit 663d60d77f9261dd0a2aa3e2023ab5262ac0dc83
Author: Ang Kah Min, Kelvin <angk@garena.com>
Date:   Tue Jun 6 12:47:04 2017 +0800

    sea
```

Before amend

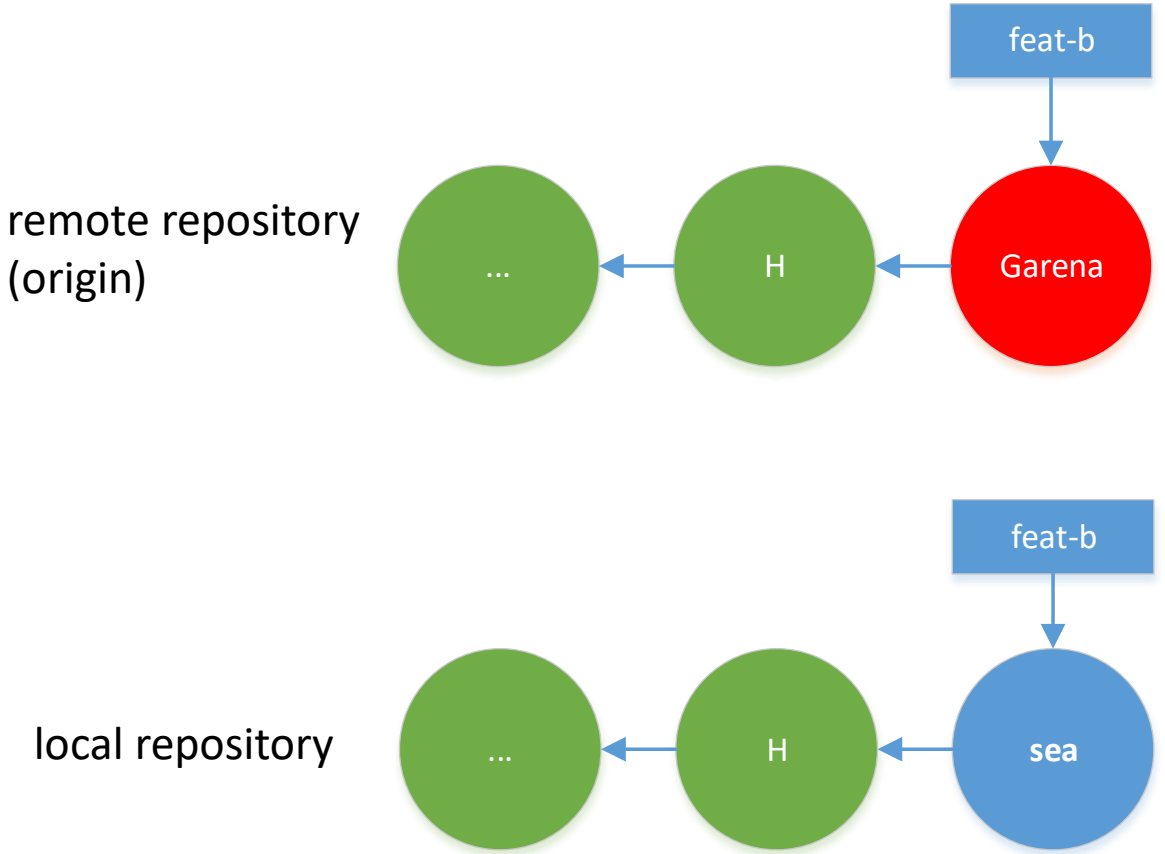


After amend



Altering History

- If you've pushed the old commit to the repository previously, you'll encounter an issue as the tip of the remote branch is not part of your history anymore.




```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git push origin feat-b
To ssh://gitlab@git.garena.com:2222/angk/repo-helloworld.git
 ! [rejected]        feat-b -> feat-b (non-fast-forward)
error: failed to push some refs to 'ssh://gitlab@git.garena.com:2222/angk/repo-h
elloworld.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

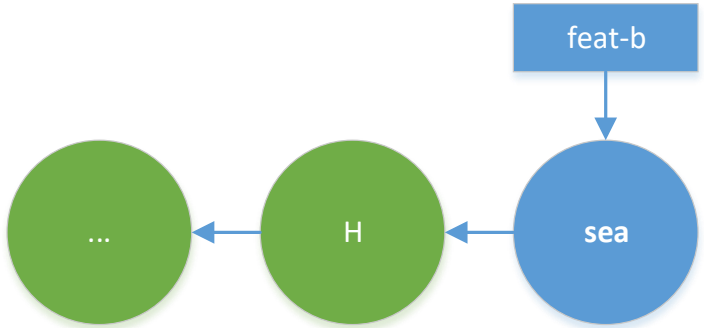
Altering History

- To resolve this, you use:

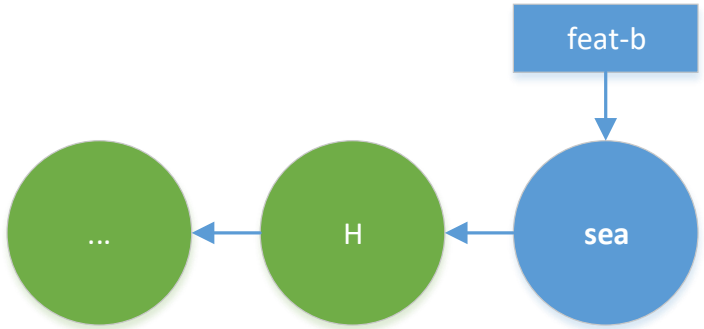
git push origin feat-b --force

The **--force** parameter is to forcefully overwrite whatever is on the remote branch.

remote repository (origin)



local repository



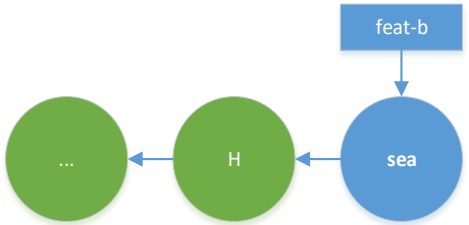
Altering History

- To resolve this, you use:

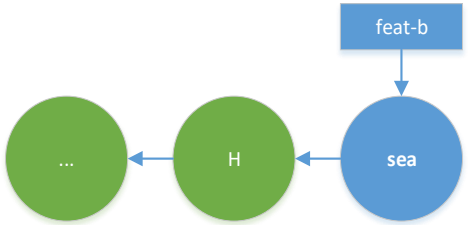
git push origin feat-b --force

The **--force** parameter is to forcefully overwrite whatever is on the remote branch.

remote repository (origin)



local repository

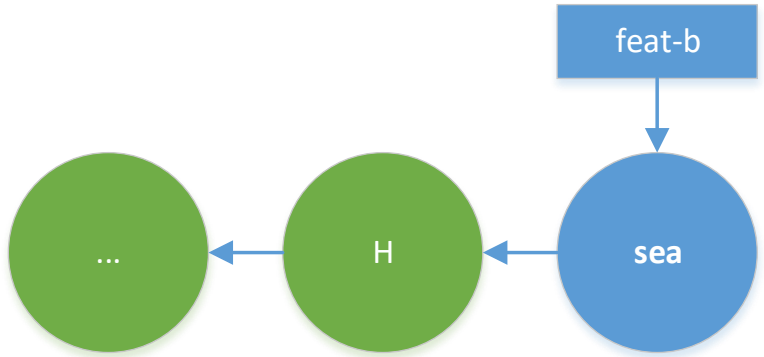



```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git push origin feat-b --f
orce
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 280 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote:
remote: ========================================================================
remote:
remote:                We Garenians work hard, and play harder!!!
remote:
remote: ========================================================================
remote:
remote: To create a merge request for feat-b, visit:
remote:   https://git.garena.com/angk/repo-helloworld/merge_requests/new?merge_r
equest%5Bsource_branch%5D=feat-b
remote:
To ssh://gitlab@git.garena.com:2222/angk/repo-helloworld.git
 + 363cb26...663d60d feat-b -> feat-b (forced update)
```

Altering History

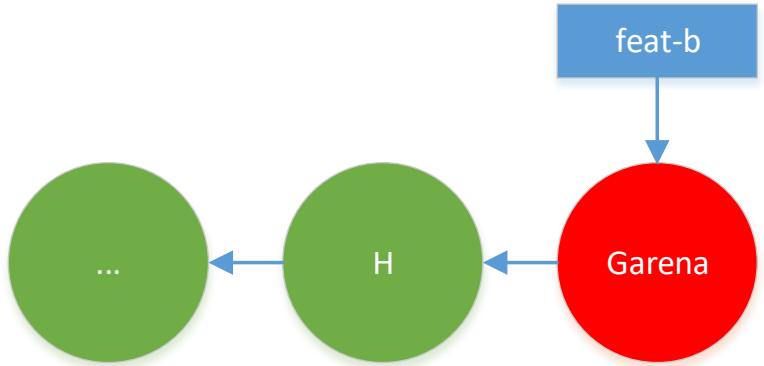
- This is generally fine if you are the only one using your branch. However…

Remote Repository (origin)



- If someone else with the old commit tries to do a **git pull**… There will be some weird issues

Your friend's local repository



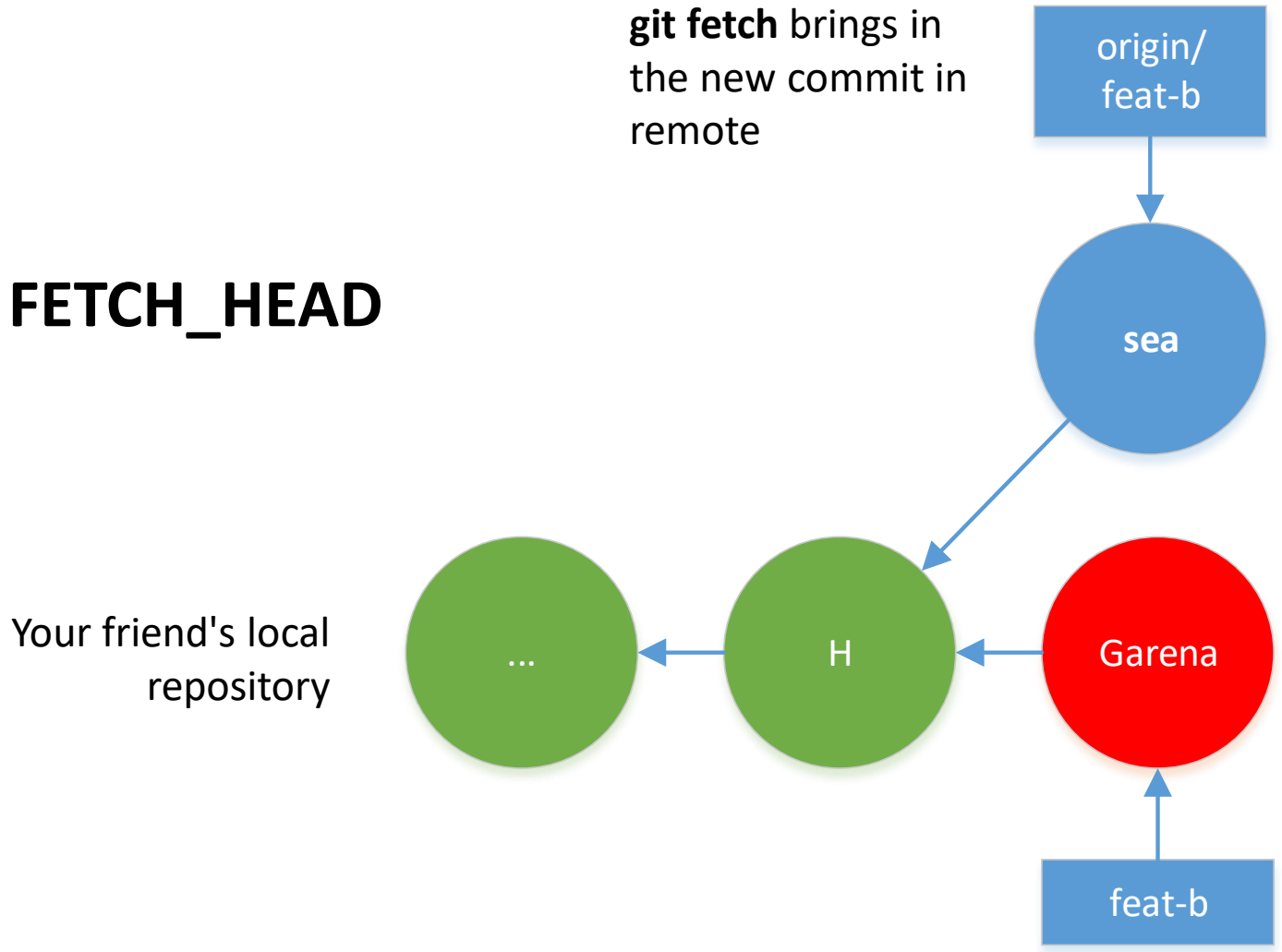
Altering History

Recall that:

git pull

is equivalent to

git fetch && git merge FETCH_HEAD



Altering History

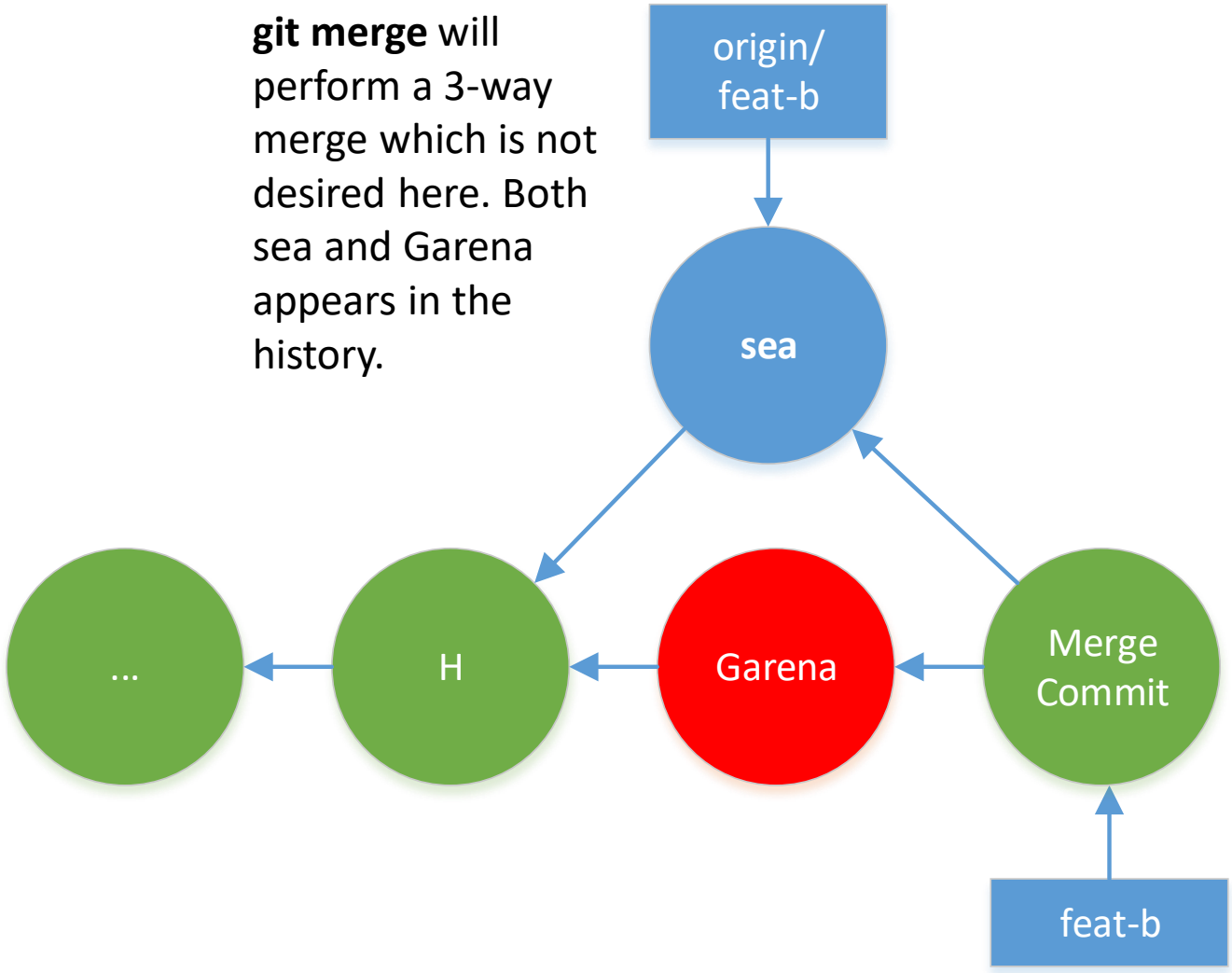
Recall that:

git pull

is equivalent to

git fetch && git merge FETCH_HEAD

git merge will perform a 3-way merge which is not desired here. Both sea and Garena appears in the history.



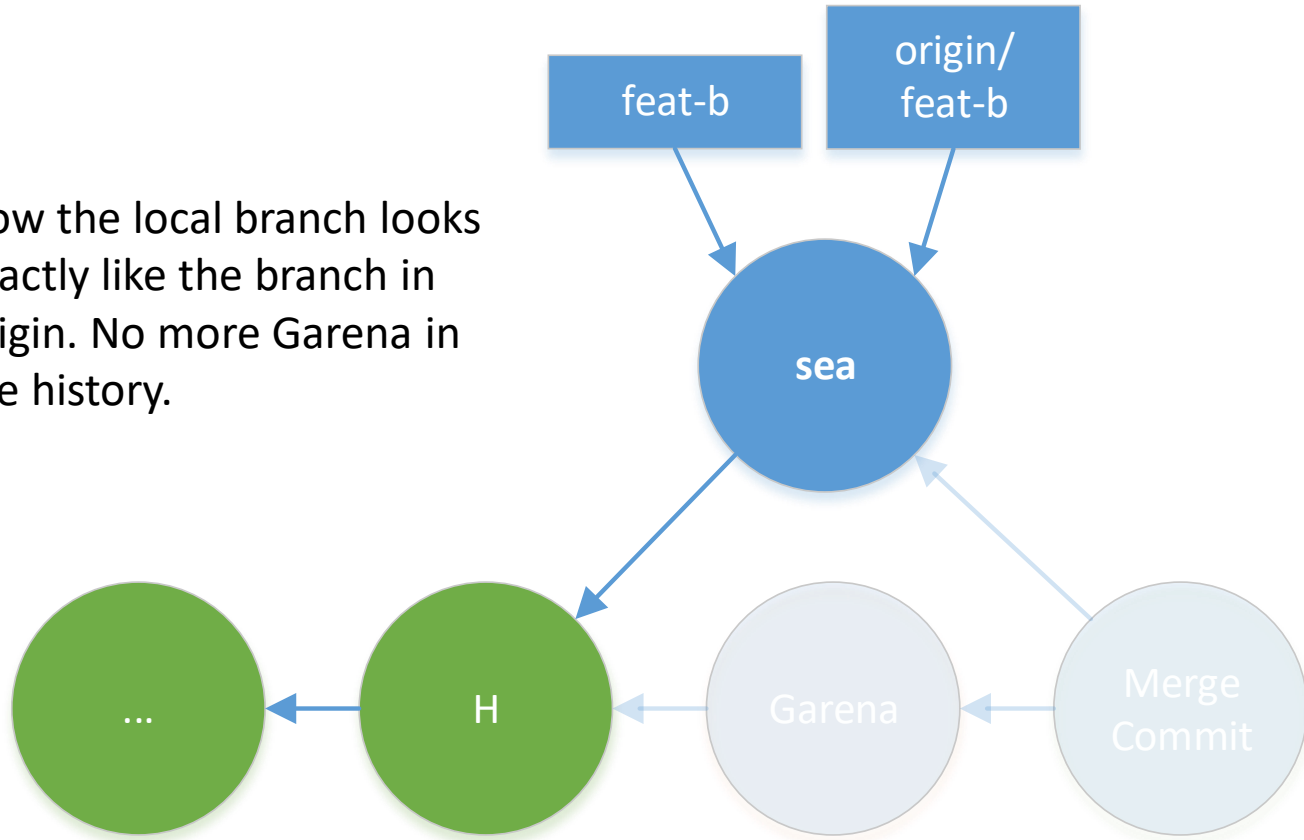
How do we fix this?

Altering History

To change your branch to how it looks like on origin:

git reset --hard origin/feat-b

Now the local branch looks exactly like the branch in origin. No more Garena in the history.

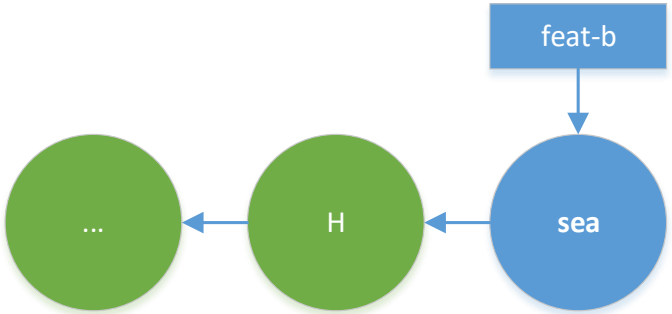


What happens to the unreferenced commits?

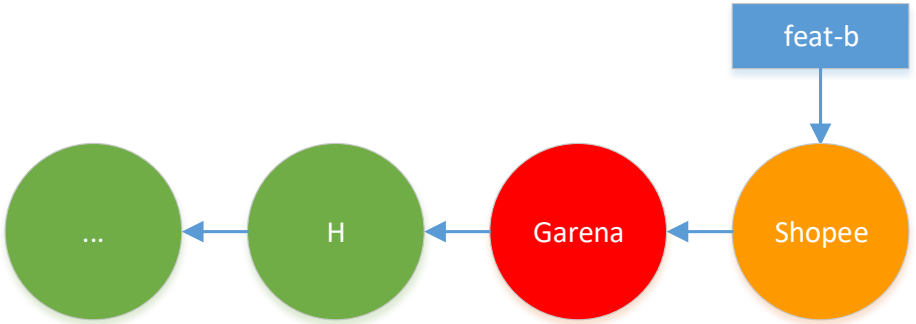
Altering History

- What if someone else has made some developments from your previous commit?

Remote Repository (origin)



Your friend's local repository




```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld2$ git status
On branch feat-b
Your branch and 'origin/feat-b' have diverged,
and have 3 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
nothing to commit, working directory clean
```

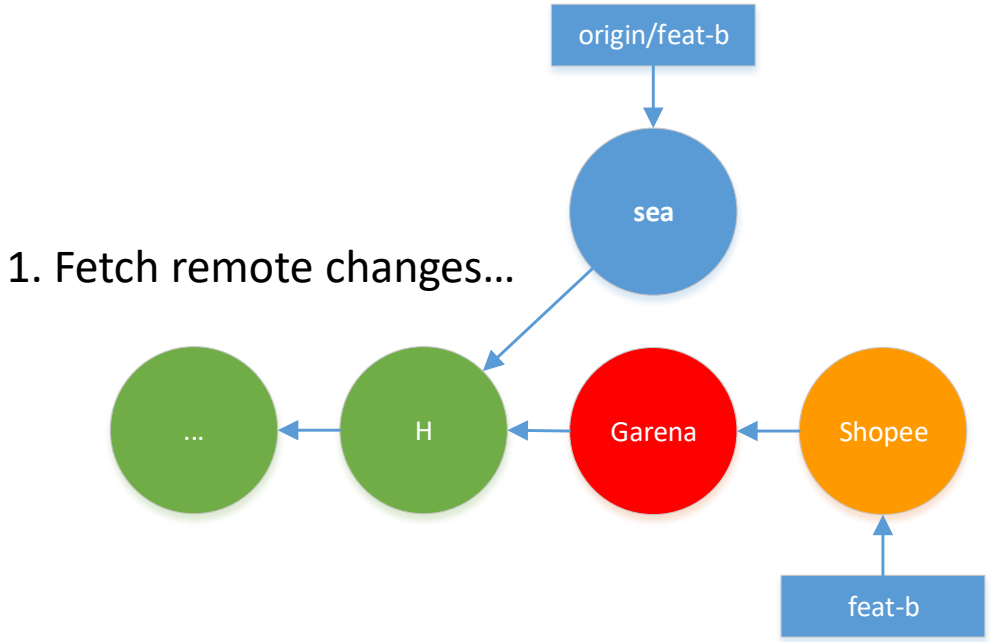
(It says 3 changes because I had a "Mall" commit after "Shopee" in reality)

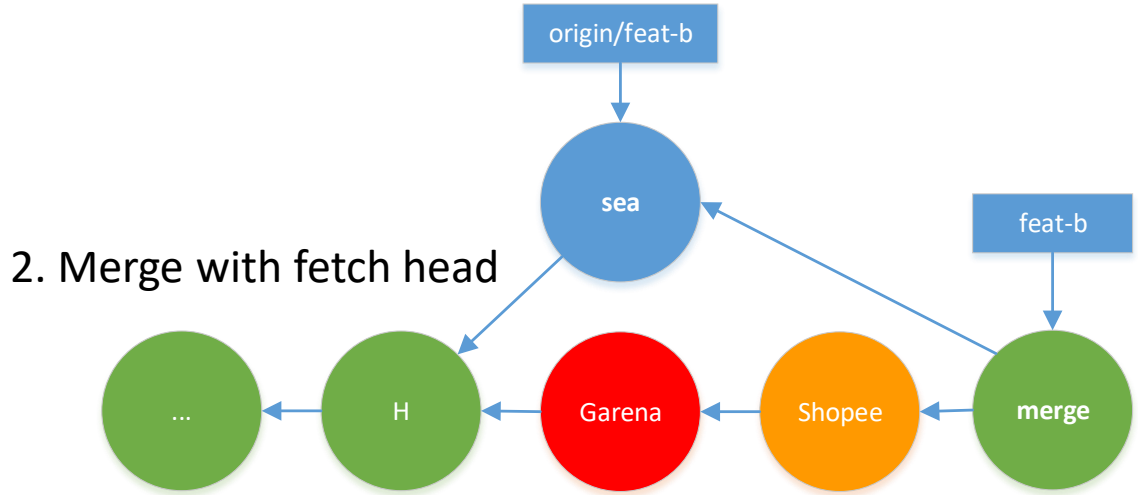
Altering History

Possibility #1

- If you try to "git pull" in feat-b branch, git will attempt to merge the two different histories using a 3-way merge.

- You will end up with both "sea" and "Garena" commits in the history.



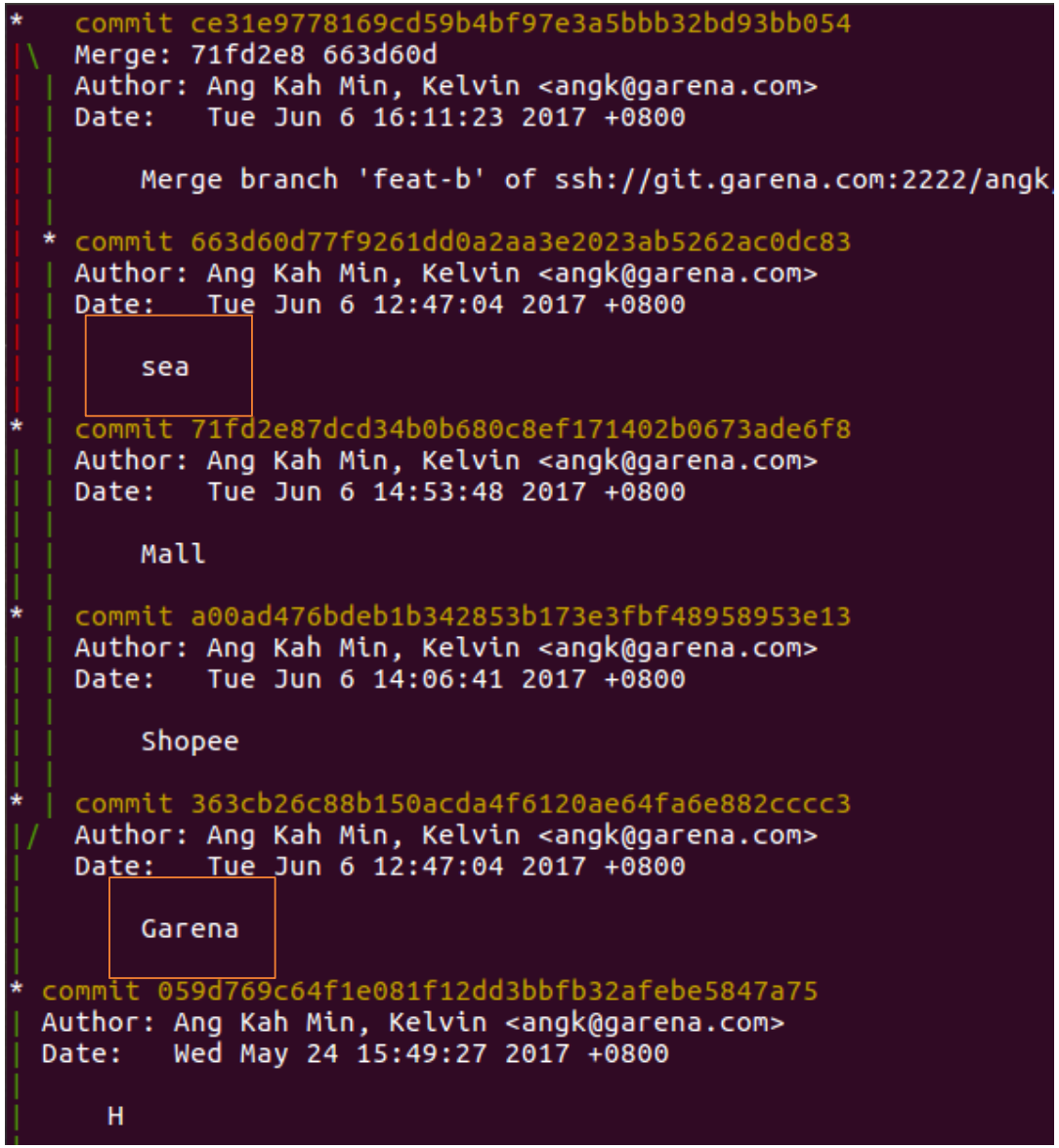


Altering History

Possibility #1

- If you try to "git pull" in feat-b branch, git will attempt to merge the two different histories using a 3-way merge.

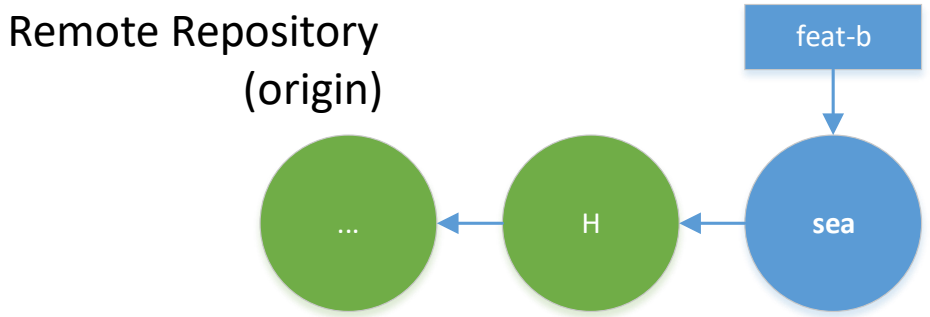
- You will end up with both "sea" and "Garena" commits in the history.

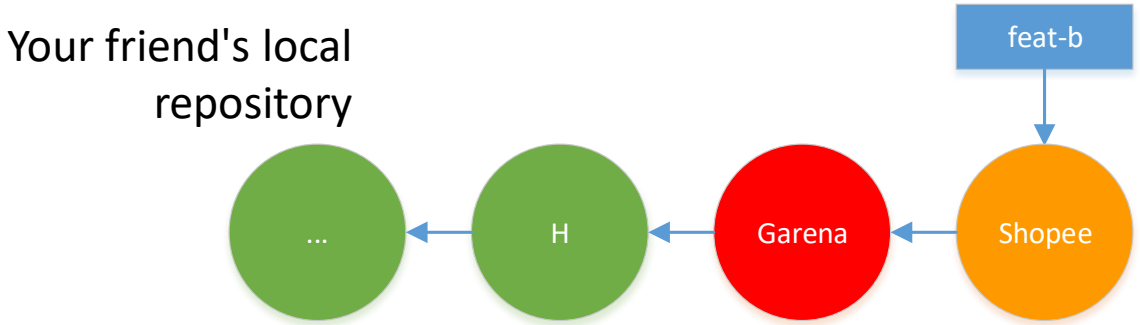


Altering History

Possibility #2

- If the user tries to push the branch, his push will be rejected as the tip of the remote branch is not an ancestor of the local branch.






```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld$ git push origin feat-b
To ssh://gitlab@git.garena.com:2222/angk/repo-helloworld.git
 ! [rejected]        feat-b -> feat-b (non-fast-forward)
error: failed to push some refs to 'ssh://gitlab@git.garena.com:2222/angk/repo-h
elloworld.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Altering History

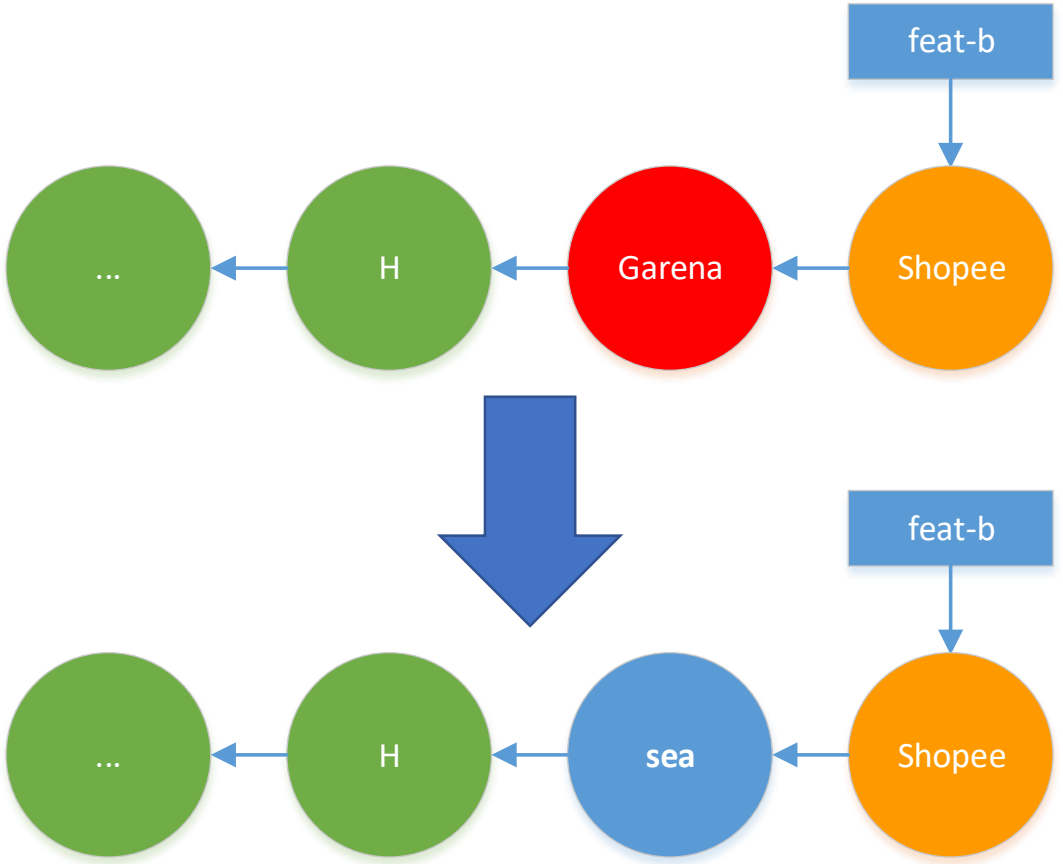
Solution

- You can use rebasing to fix this issue. This is not the only way to fix it.
- The commands to fix this using rebase is:

git fetch origin feat-b

git rebase -i origin/feat-b

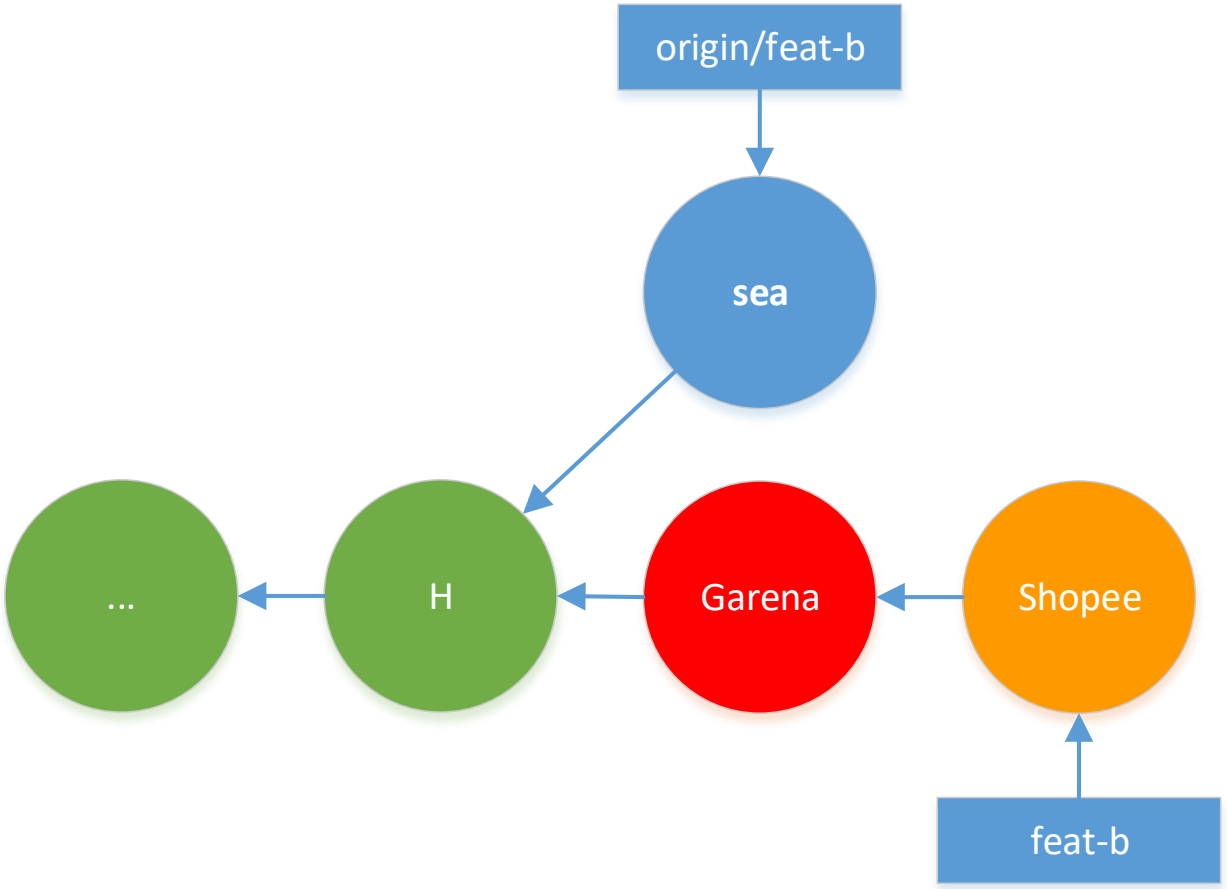
(step-by-step graphics in next slide)



Altering History

git fetch origin feat-b

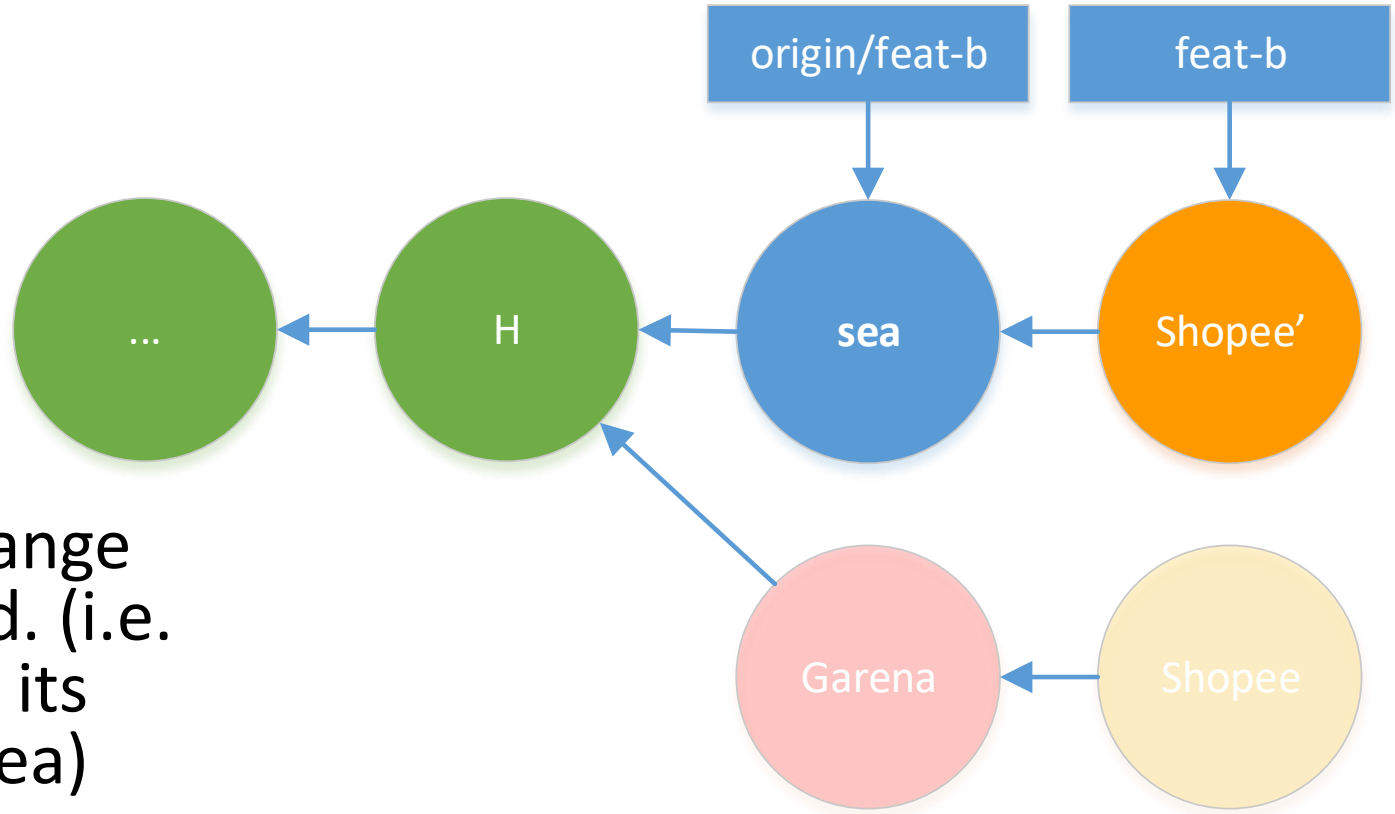
- This command fetches the remote branch and updates the origin/feat-b remote tracking branch.



Altering History

git rebase origin/feat-b

- Then you use the rebase command to move your commits over
- Rebase does this using successive cherry-picks



If a cherry-pick makes no change then the commit is discarded. (i.e. Garena is discarded because its end result exactly matches sea)

Altering History

Now you can push:

git push origin feat-b

```
angk@angk-Latitude-E5570:~/workspace/repo-helloworld2$ git push origin feat-b
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 500 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote:
remote: ========================================================================
remote:
remote:                We Garenians work hard, and play harder!!!
remote:
remote: ========================================================================
remote:
remote: To create a merge request for feat-b, visit:
remote:   https://git.garena.com/angk/repo-helloworld/merge_requests/new?merge_re
remote:
To ssh://gitlab@git.garena.com:2222/angk/repo-helloworld.git
   663d60d..b6ee881  feat-b -> feat-b
```

Altering History

This is only one scenario that can cause issues.

There are many ways altering history can cause inconveniences or strange problems.

- As a general rule, **avoid altering history if**:
  - The branch is being used by someone else
  - The commits to be altered has been merged into another branch
  - Any other scenario that can alter someone else's history
- In general, you can freely alter things that others have not seen yet.

Git Branching Models

Git in the real world

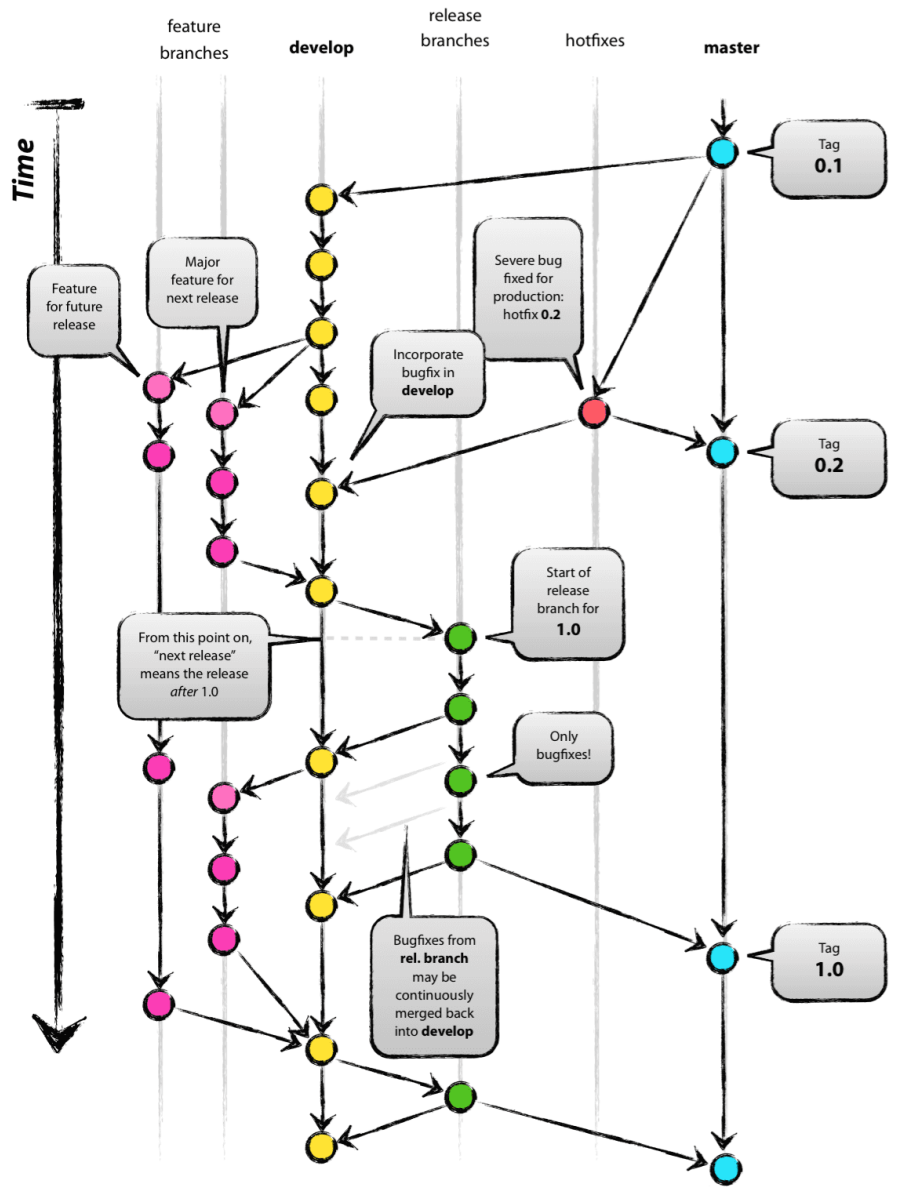
Famous Git Branching Models

- Git Flow - Comprehensive
- GitHub Flow - Minimalist
- GitLab Flow – Hybrid + Environments
- Labs Flow – Recommended

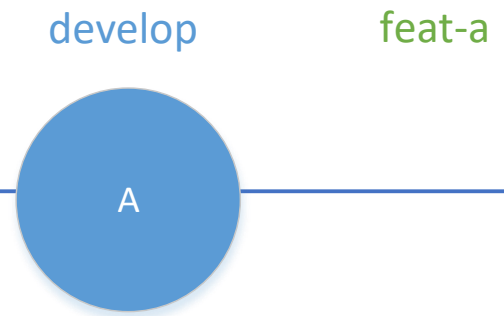
It is useful to know the idea behind these Git Flows, as the concepts are mostly universal.

Git Flow

- One of the first branching models proposed
- Two main branches:
  - develop (For development)
  - master (Actual deployable code)
- Three topic branch types:
  - feature branches
  - release branches
  - hotfix branches



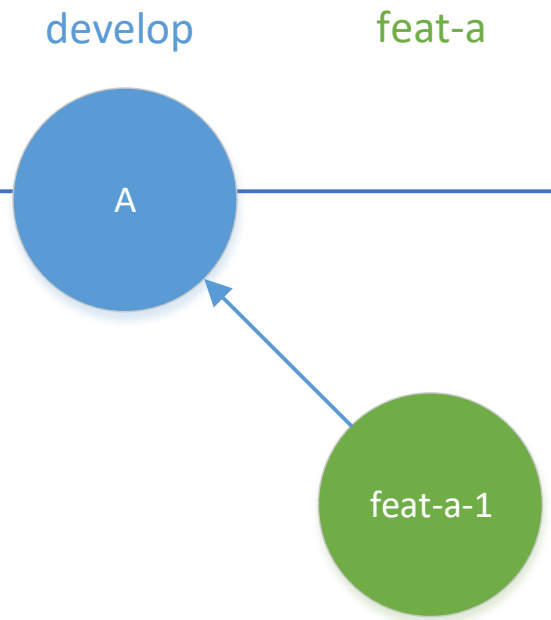
Git Flow – New Feature



- Creating a new feature is exactly the same as discussed in Git Foundations (LC126).

- Simply create a feature branch, work on it, and then merge it back.

Git Flow – New Feature



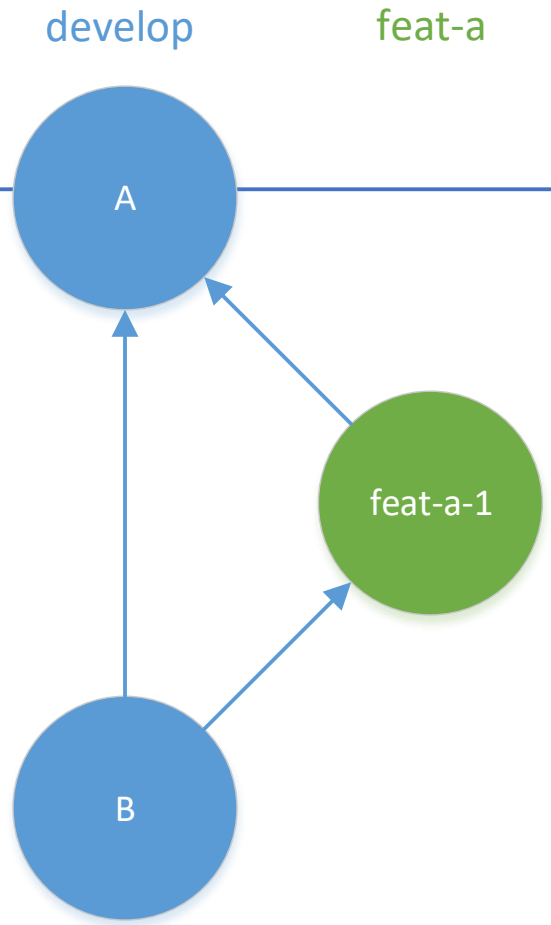
- Creating a new feature is exactly the same as discussed in Git Foundations (LC126).

- Simply create a feature branch, work on it, and then merge it back.

Git Flow – New Feature

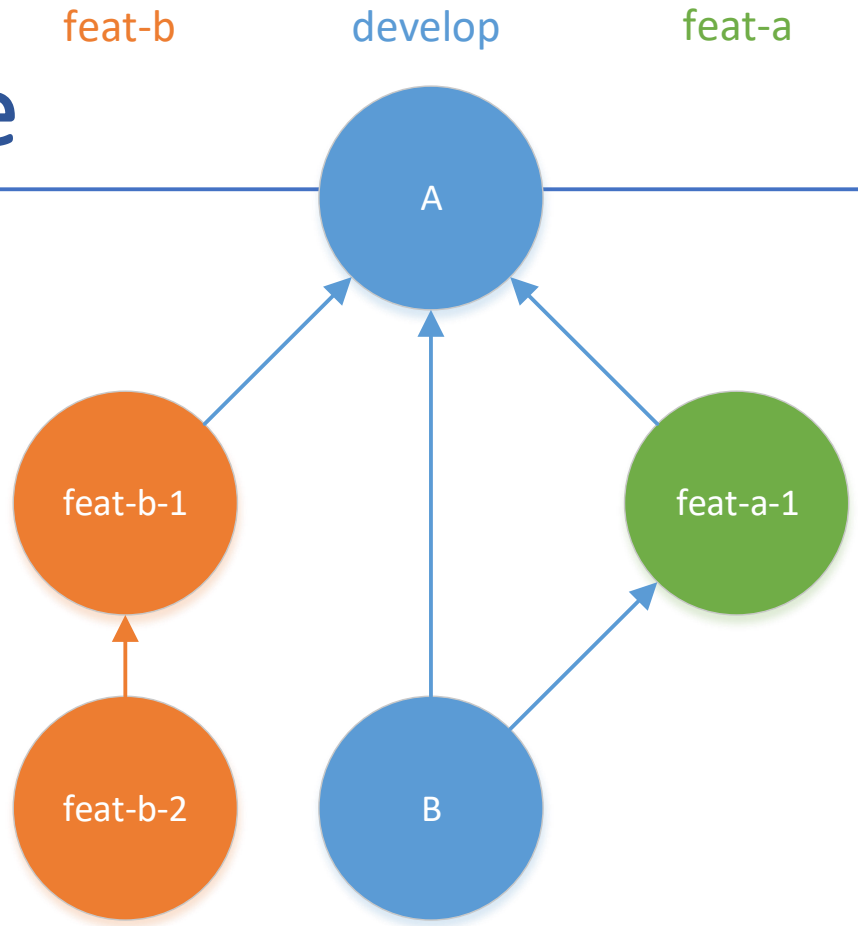
- Creating a new feature is exactly the same as discussed in Git Foundations (LC126).

- Simply create a feature branch, work on it, and then merge it back.



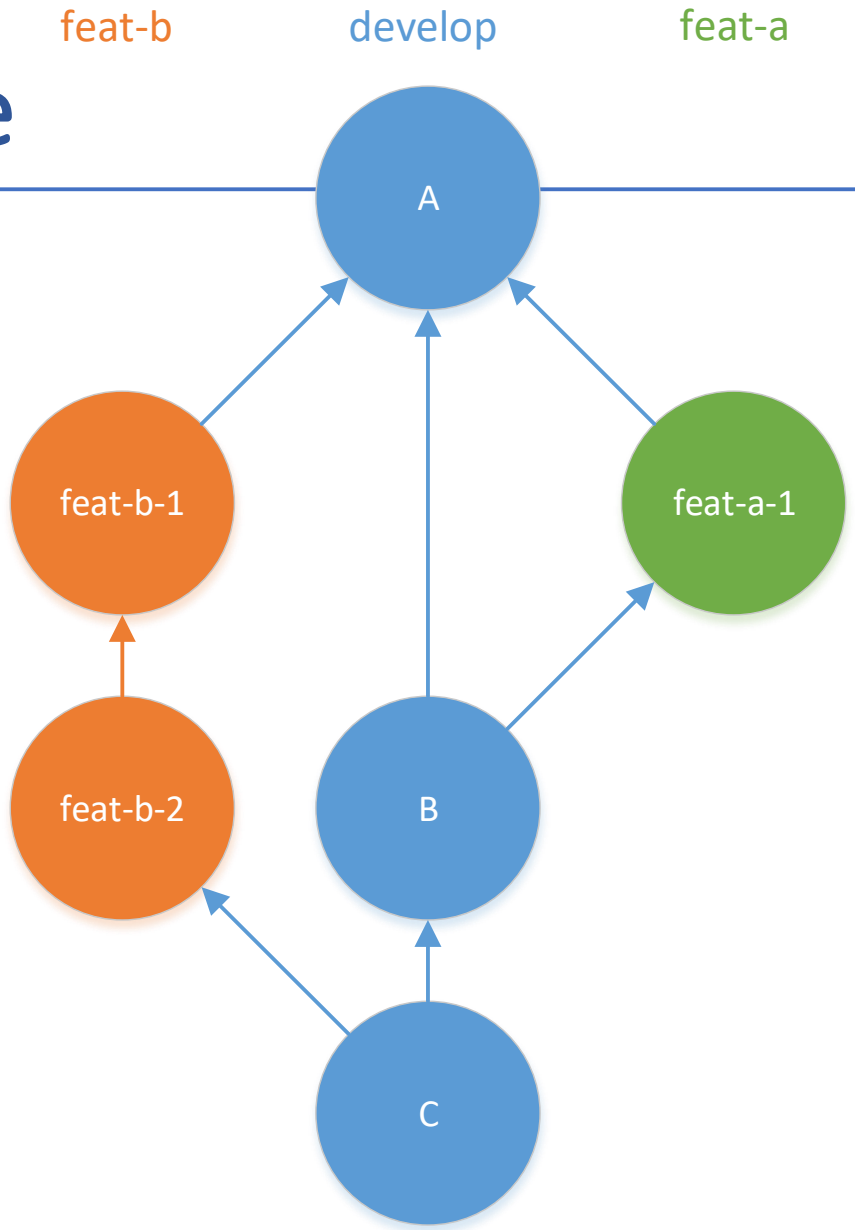
Git Flow – New Feature

- Another person can be simultaneously working on another feature in **feat-b** branch, and he can merge it back when he is done.



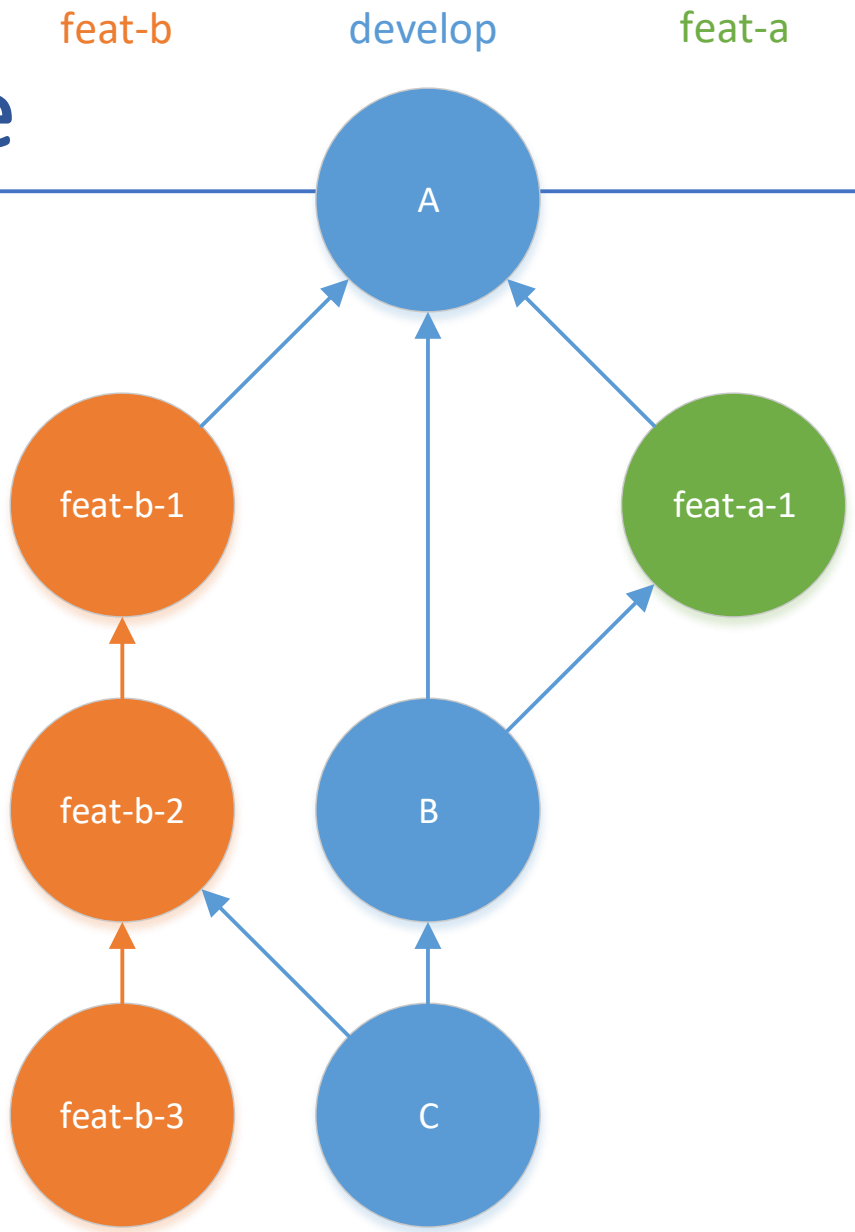
Git Flow – New Feature

- Another person can be simultaneously working on another feature in **feat-b** branch, and he can merge it back when he is done.



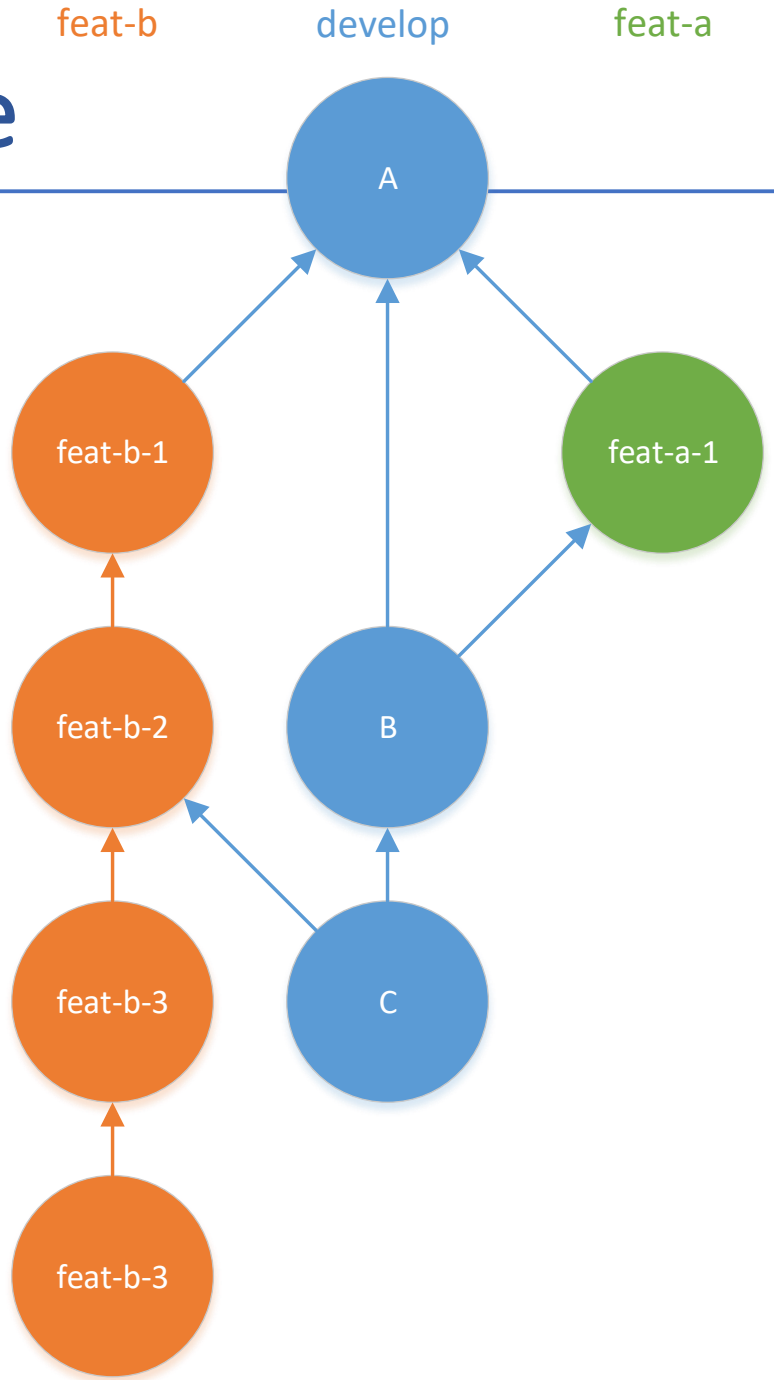
Git Flow – New Feature

- When you are on a feature branch, you can continue developing and merging multiple times without interfering with others.



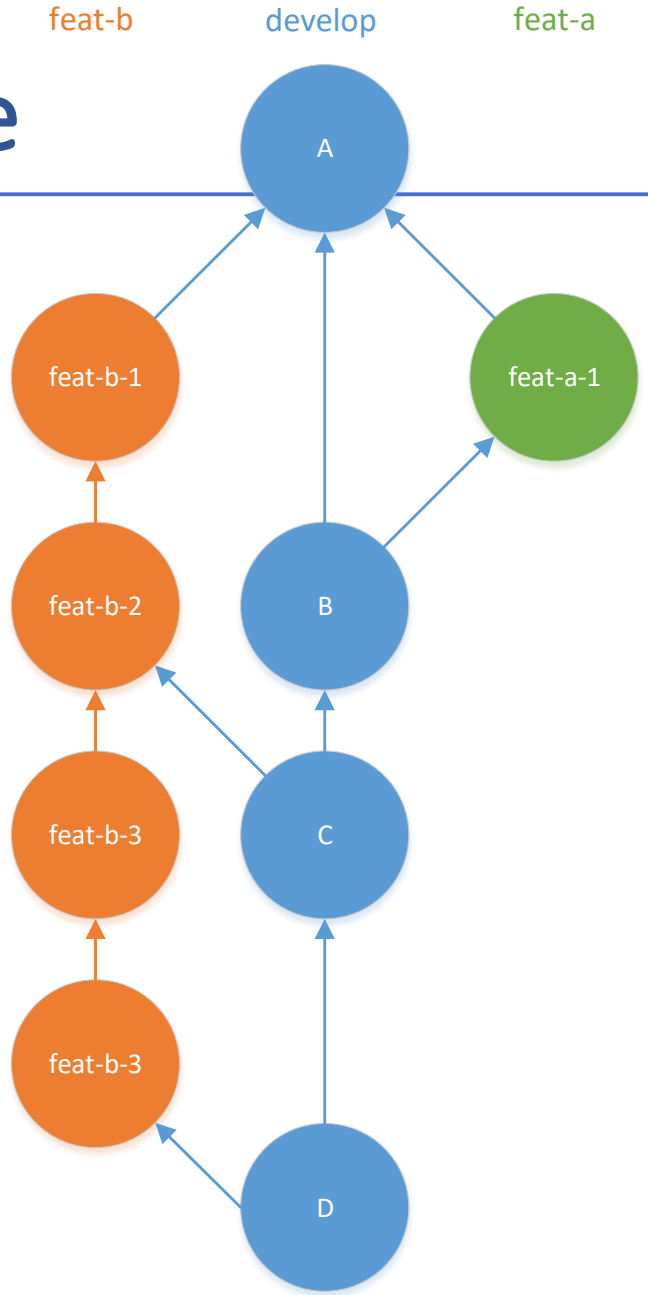
Git Flow – New Feature

- When you are on a feature branch, you can continue developing and merging multiple times without interfering with others.



Git Flow – New Feature

- When you are on a feature branch, you can continue developing and merging multiple times without interfering with others.



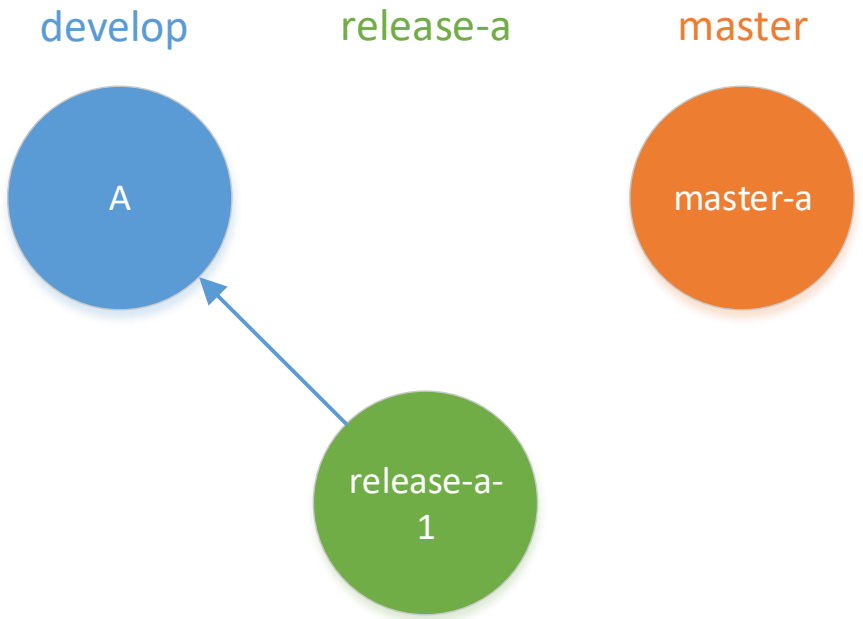
Git Flow – Preparing for Release

- A release branch is used to prepare the development branch to be merged into master.

- (Warning: This is different from the release branch in Labs Flow)

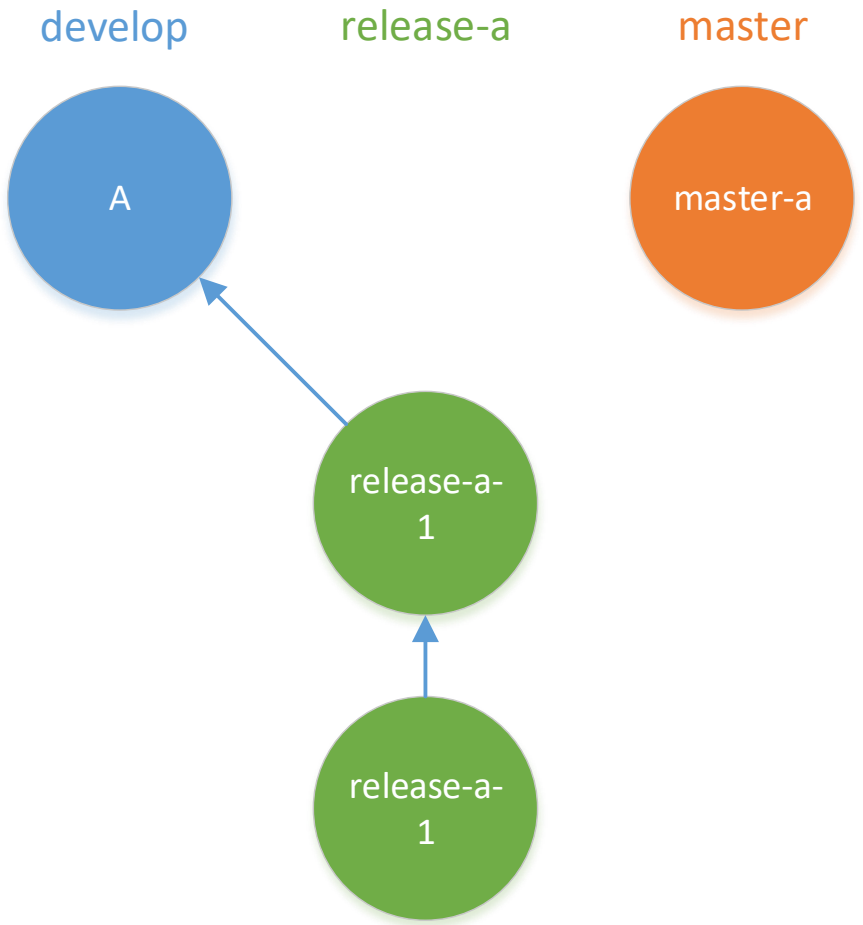
Git Flow – Preparing for Release

- You make some changes to ensure production quality



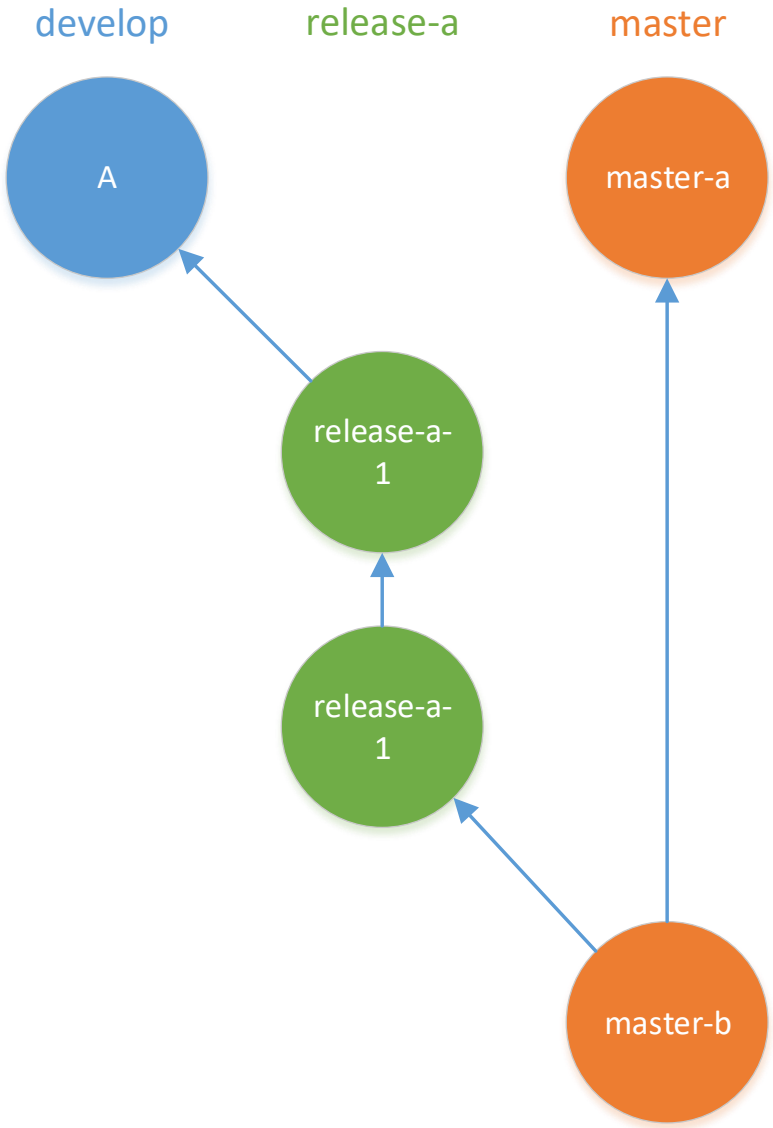
Git Flow – Preparing for Release

- You make some changes to ensure production quality



Git Flow – Preparing for Release

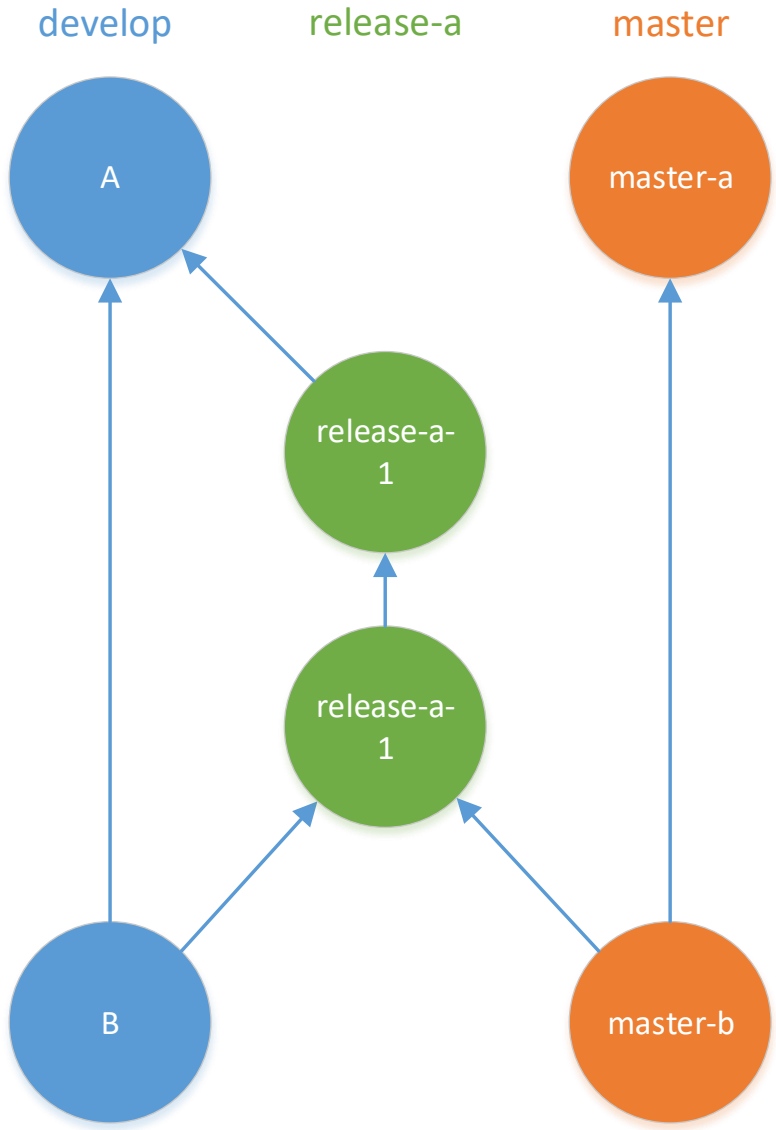
- When ready, you can merge the release branch into the master branch.



Git Flow – Preparing for Release

- But you also need to ensure that your changes are merged back into master, so your changes are reflected in development.

- If you did not merge back into master…
  - Another developer checks out a release branch from develop branch (e.g. Commit A instead of B)…
  - Makes the changes to the same places as you…
  - Merges it back to master branch…
  - CONFLICT!!!



Git Flow – Preparing for Release

- This is also a good time to add tags.

```
git tag -a 0.2 –m "message here"
```

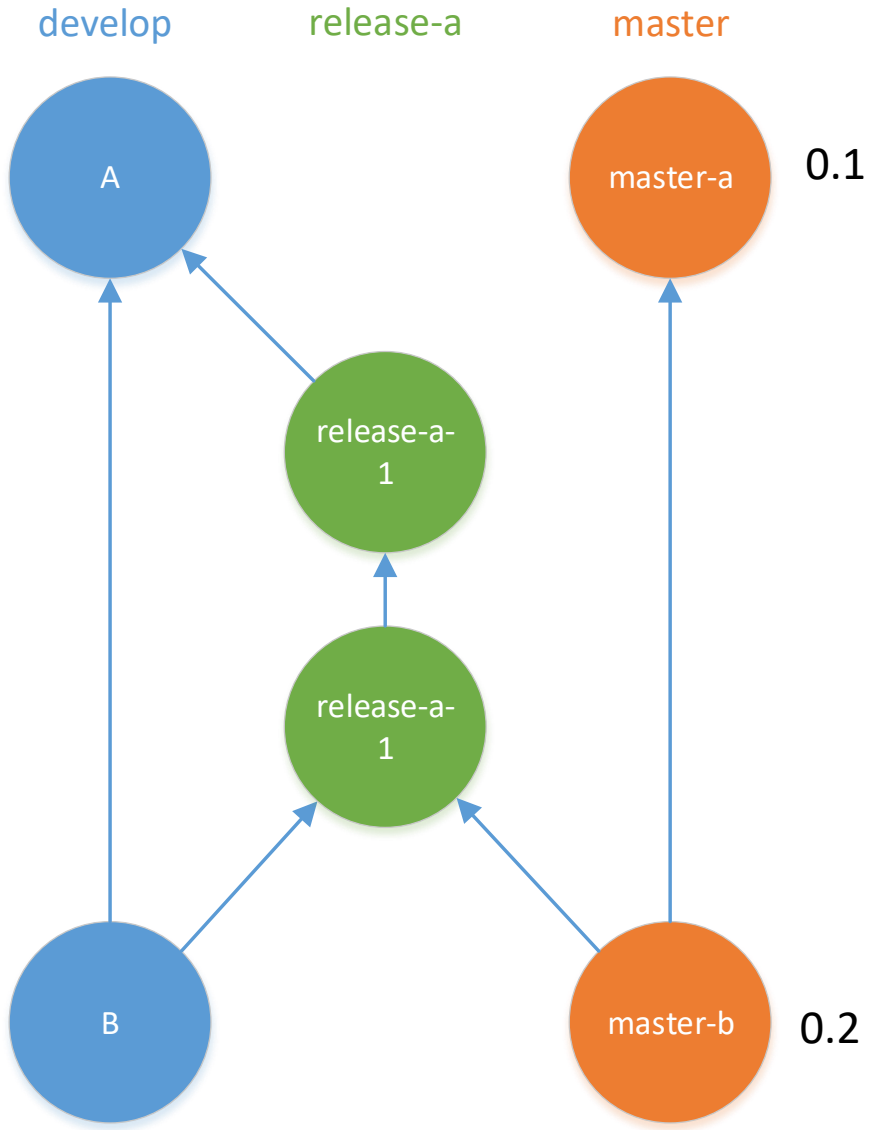
The above command will tag the current commit.

- To see all tags, use:

```
git tag -l
```

- To checkout a tag, use:

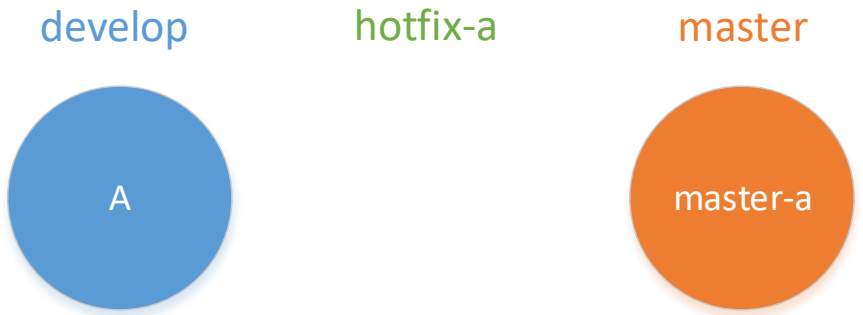
```
git checkout tags/version
```



Git Flow – Hotfix

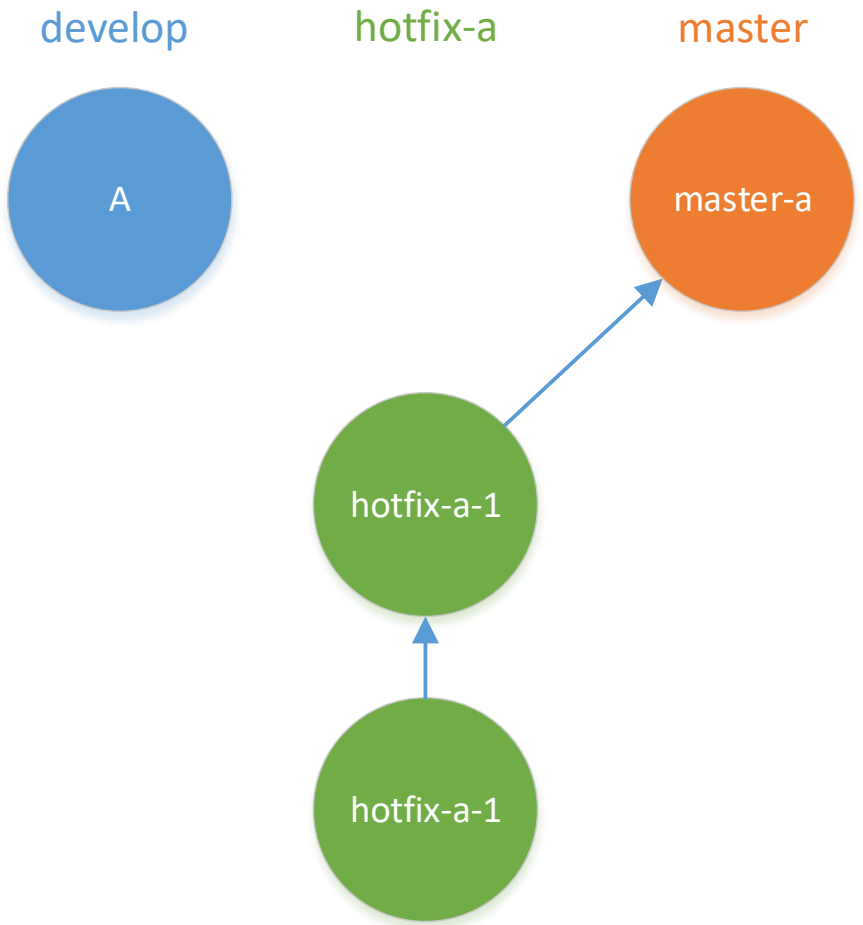
- A hotfix is used to fix something in production quickly.

- First, we check out a hotfix branch from master.



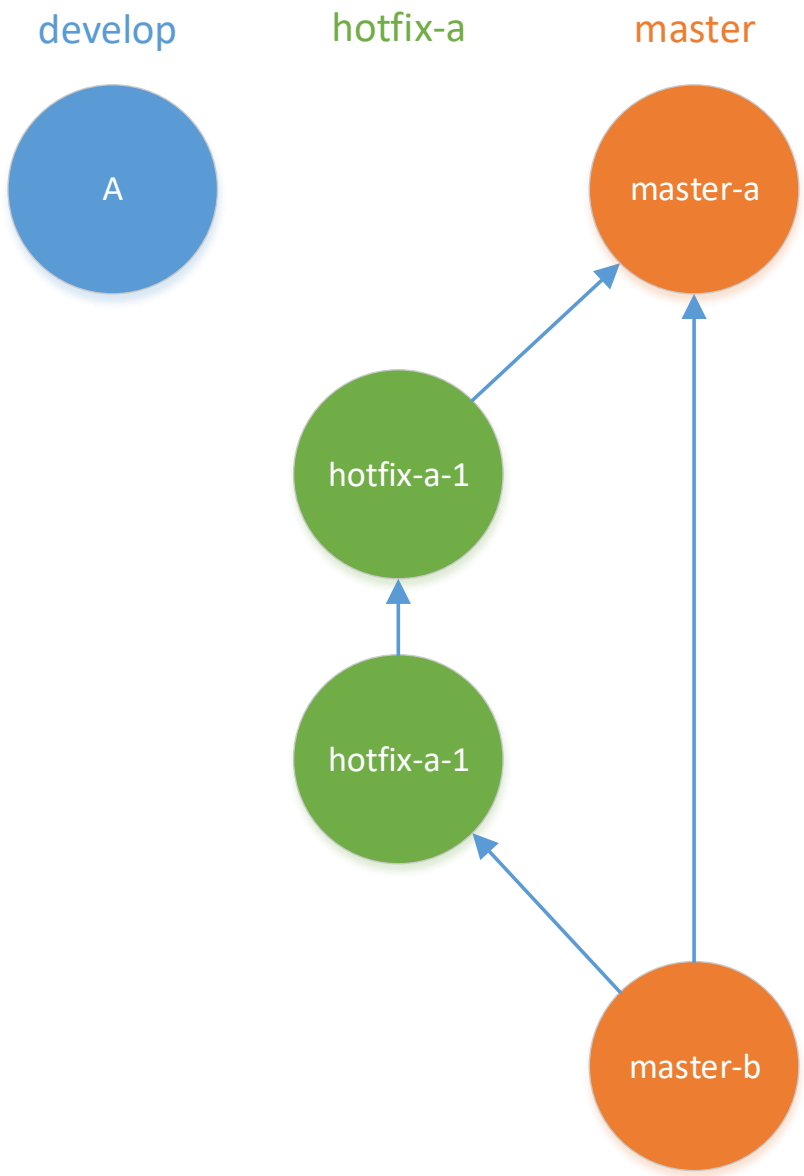
Git Flow – Hotfix

- We make the necessary changes… (e.g. to fix a bug)



Git Flow – Hotfix

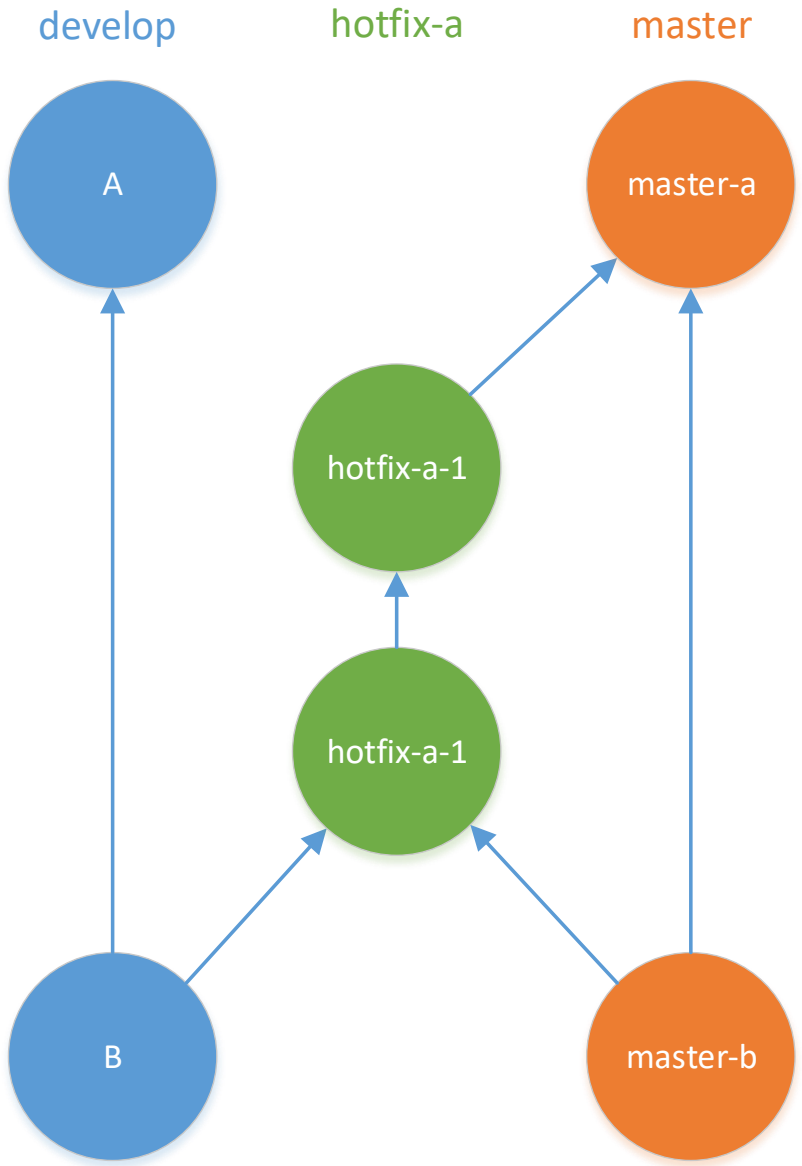
- Then we merge it back into master so the bug is fixed.



Git Flow – Hotfix

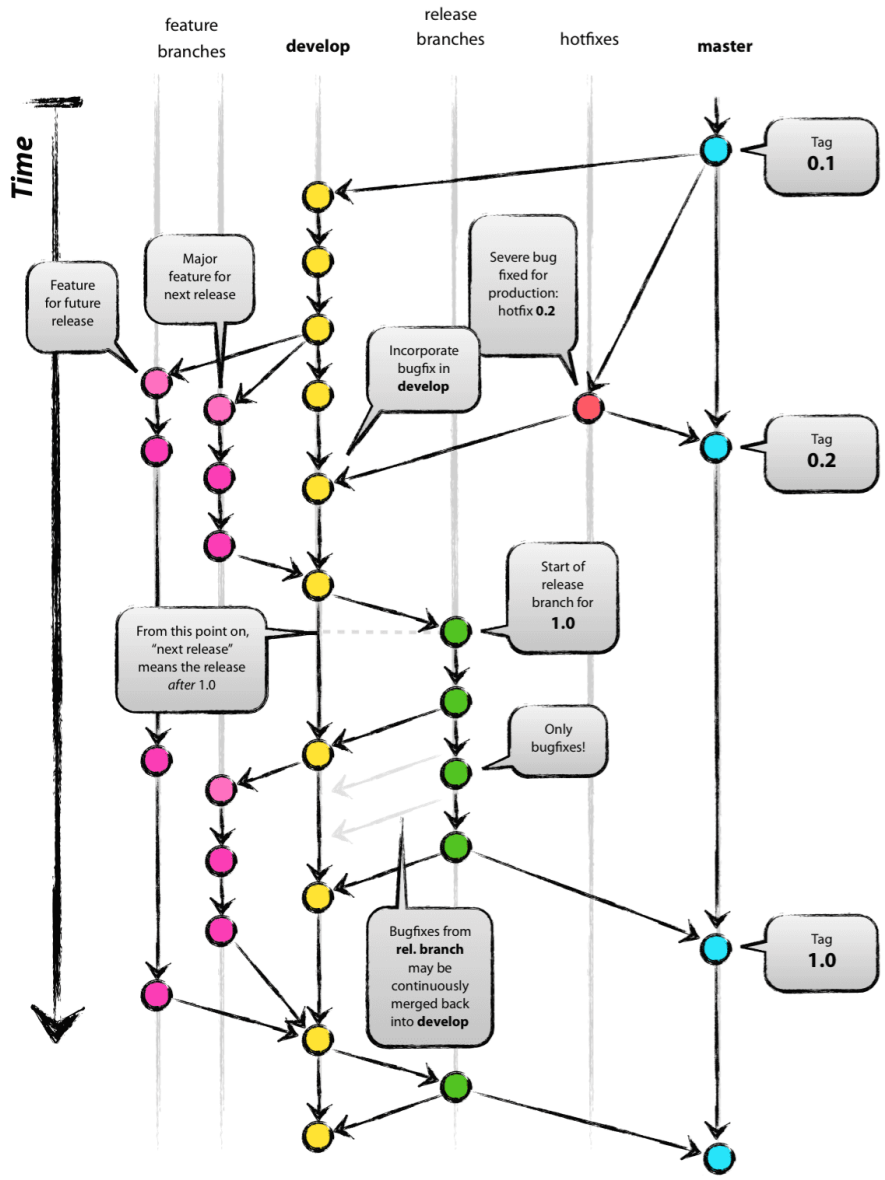
- But always remember to merge it back into develop!
- I will leave it as an exercise to reason why there would be conflicts!

  (Hint: If you make a release branch that modifies the same code as the hotfix, then…)



Git Flow - Summary

- Main branches:
  - **master branch**
    Deployed code, always buildable
  - **develop branch**
    Used for development
- Topic branches:
  - **feature branch**
    develop -> develop
  - **release branch**
    develop -> develop and release
  - **hotfix branch**
    release -> develop and release

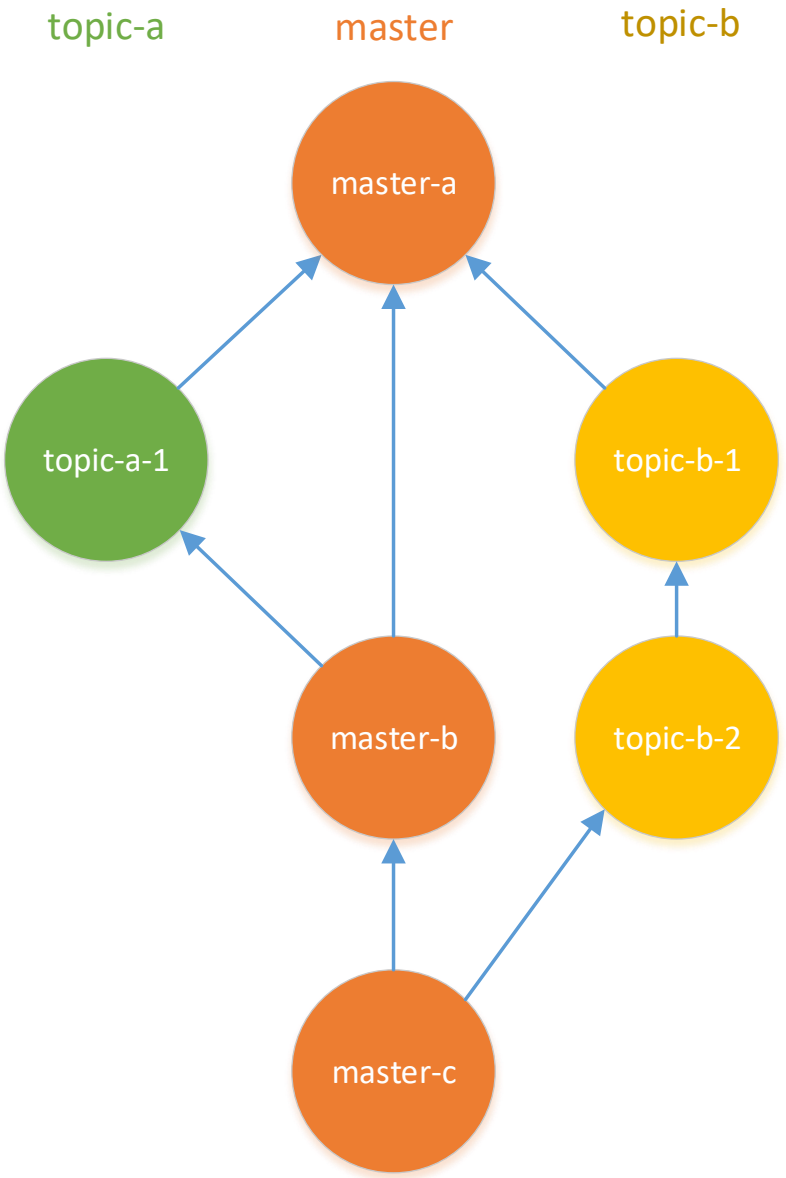


Git Flow - Summary

- Pros:
  - Very comprehensive
  - Very disciplined and controlled
- Cons:
  - Master branch is not the default development branch.
  - Very complex, too complex for most use cases.
  - Feature and hotfix branches not necessary in CI/CD

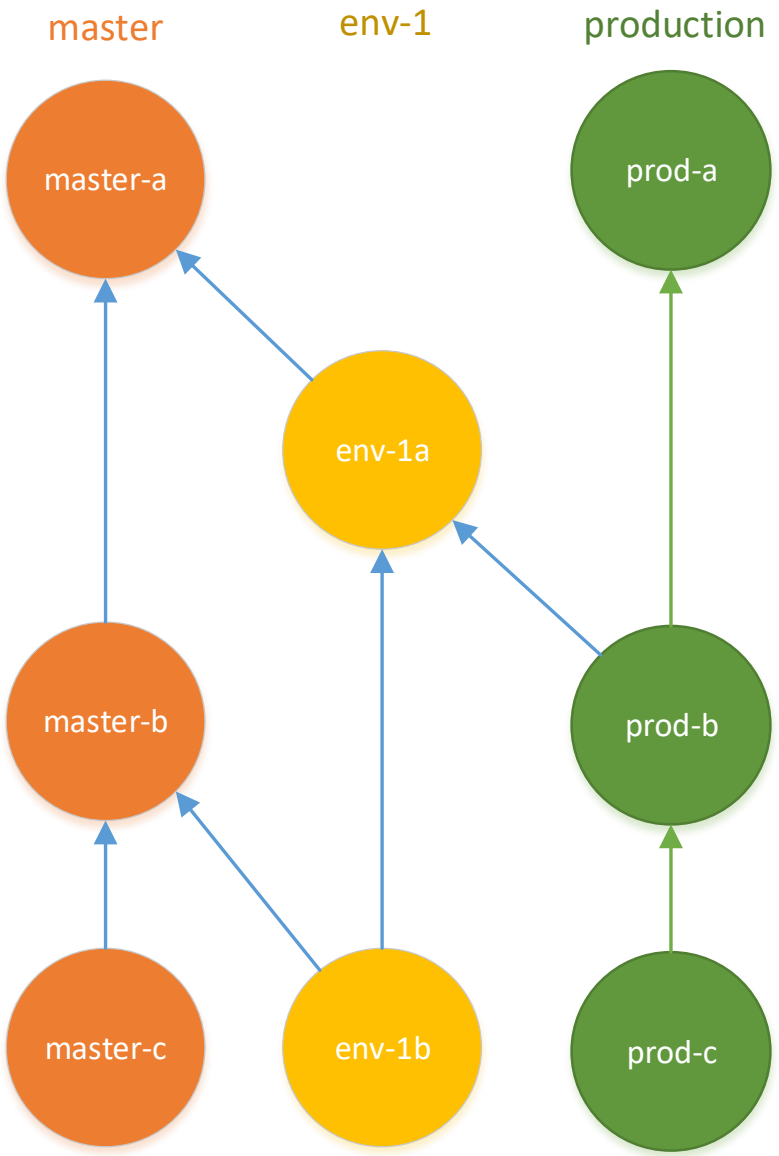
GitHub Flow

- Main branches:
  - master branch
- Topic branches:
  - any topic branch
  - features, hotfix, bugs, etc.



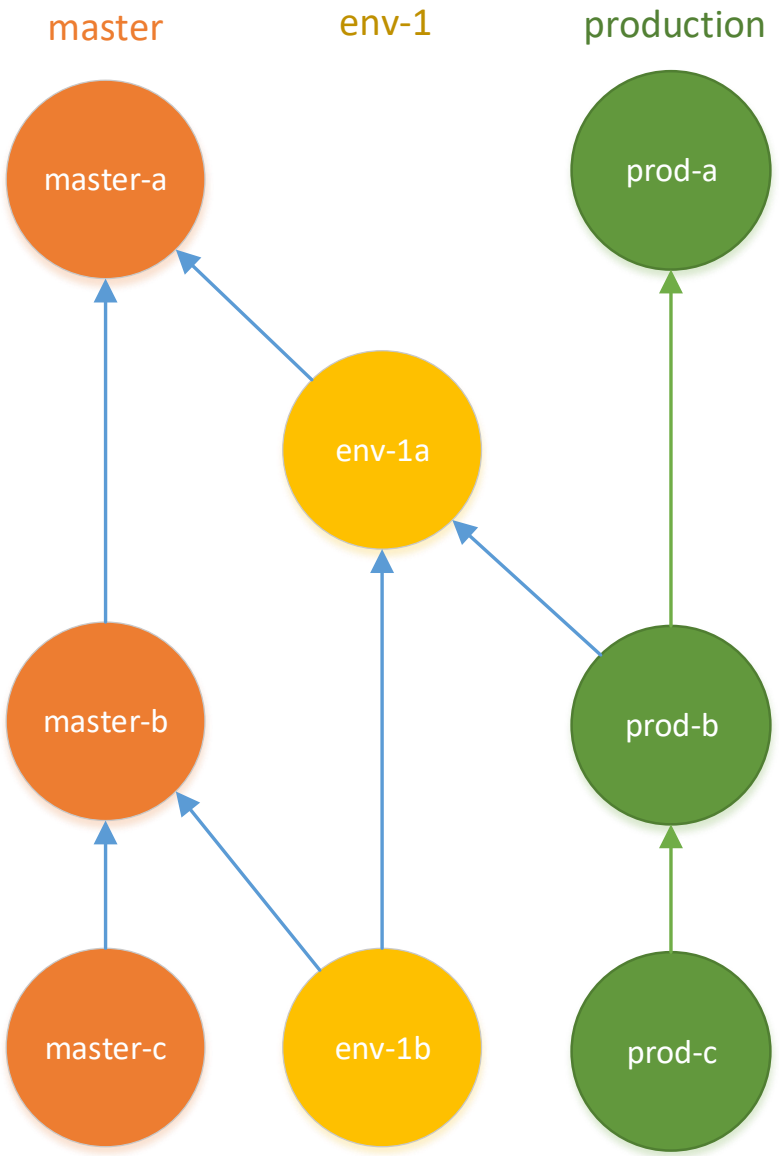
GitLab Flow

- Main branches:
  - master branch
  - production branch
  - any number of environment branches
- Topic branches:
  - any topic branch
  - features, hotfix, bugs, etc.
- (Something like GitHub flow… But duplicated many times, one for each environment.)



GitLab Flow

- Each branch corresponds to an environment, e.g. staging, test, production, etc.
- The master branch is usually deployed in a staging environment.
- An example of an environment branch can be **pre-production**, etc.
- When a branch is "stable enough", a merge request is made to merge it to the next level environment branch.

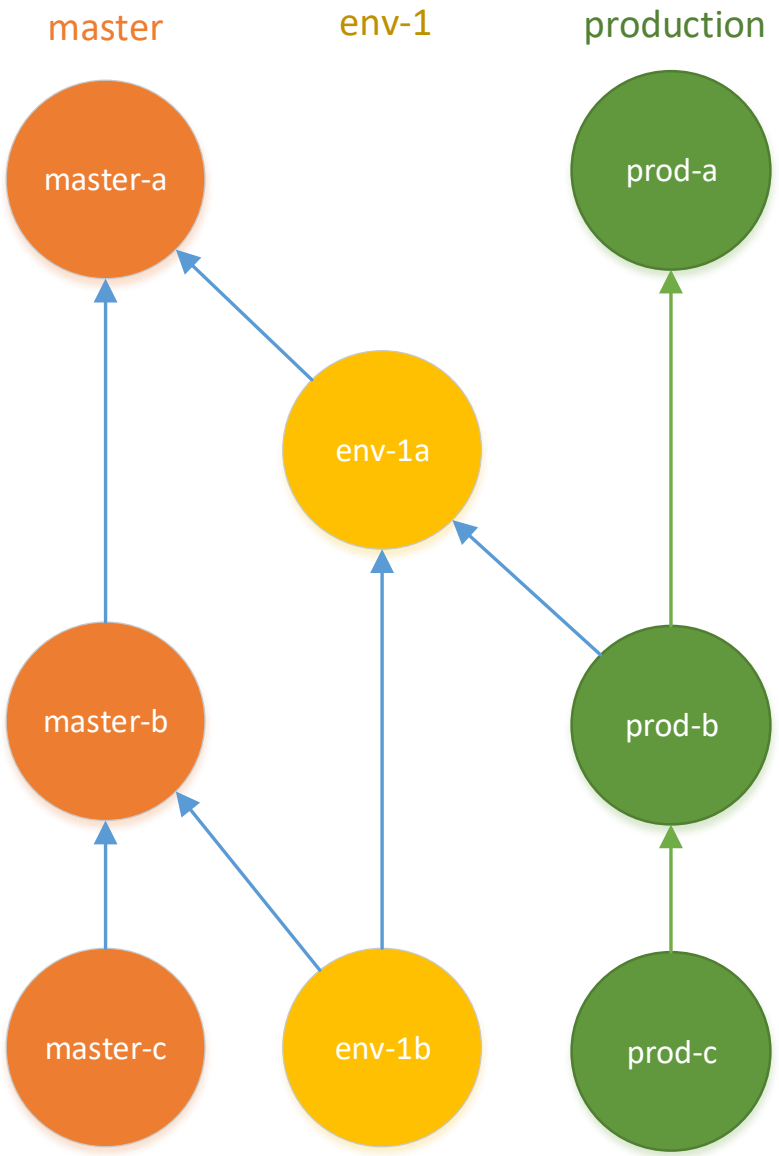


GitLab Flow

Most development is done on master.

Rules:

- All branches must be deployable at all times.
- Branches other than those created on master are hotfix branches and must be merged to all lower environments.



Labs Flow

- Somewhat a combination of all the 3 previous flows.
- This flow is not a hard and fast rule, but more like a guideline.
- You may adopt any feature of the Labs Flow as needed by your project requirements.
- This section will describe the Labs Flow as implemented by the Shopee Mall project.

For the full guideline listing, please visit:

https://git.garena.com/garenalabs/labs_dev/wikis/git_guide

Labs Flow

- Main branches:
  - test
  - master
  - liveish (optional)
  - release
- Topic branches:
  - feature branches
  - hotfix branches

Branch	Environment
test	test
master	staging
liveish	liveish
release	live

Labs Flow

Always Deployable!

- **master**, **liveish** and **release** should always be deployable.

(This is the same as in GitLab flow.)

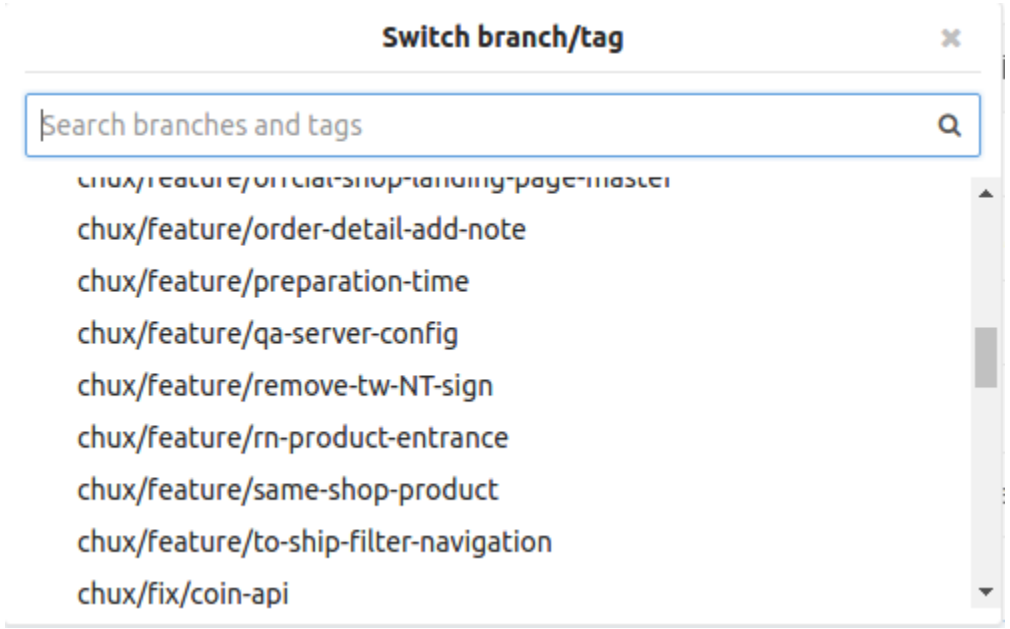
Can Fail…

- **test** branch can fail, and it is used for deploying features for testing. It will be reset regularly into master branch.

Labs Flow

- **1. Developers push code into own topic branch…**
- 2. Merge requests are made to merge into environment branches (after code review)…
- 3. Environments branches are deployed to actual servers via Jenkins…

- Checkout a new branch (usually from master) and follow your team's branch naming convention.



Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

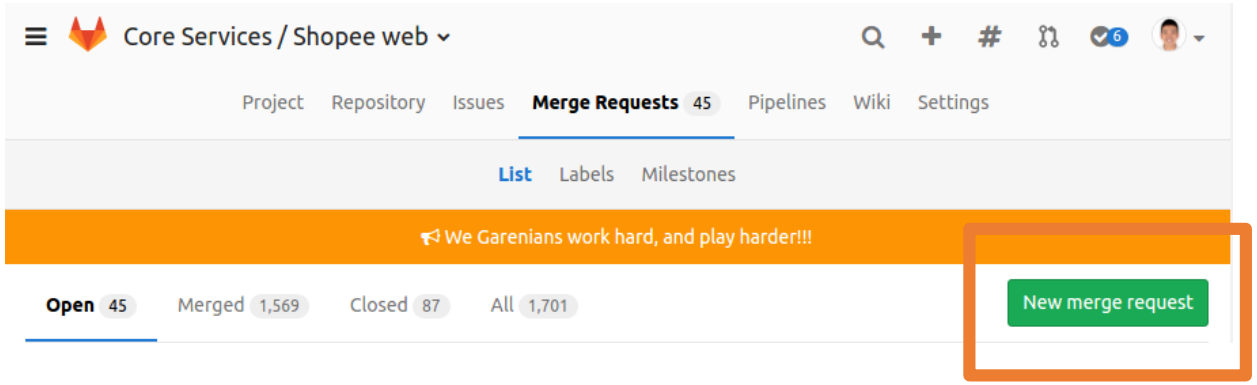
- Make your necessary changes, and push into your topic branch

git push origin angk/hotfix/change-live-rabbit-to-new-idc

Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

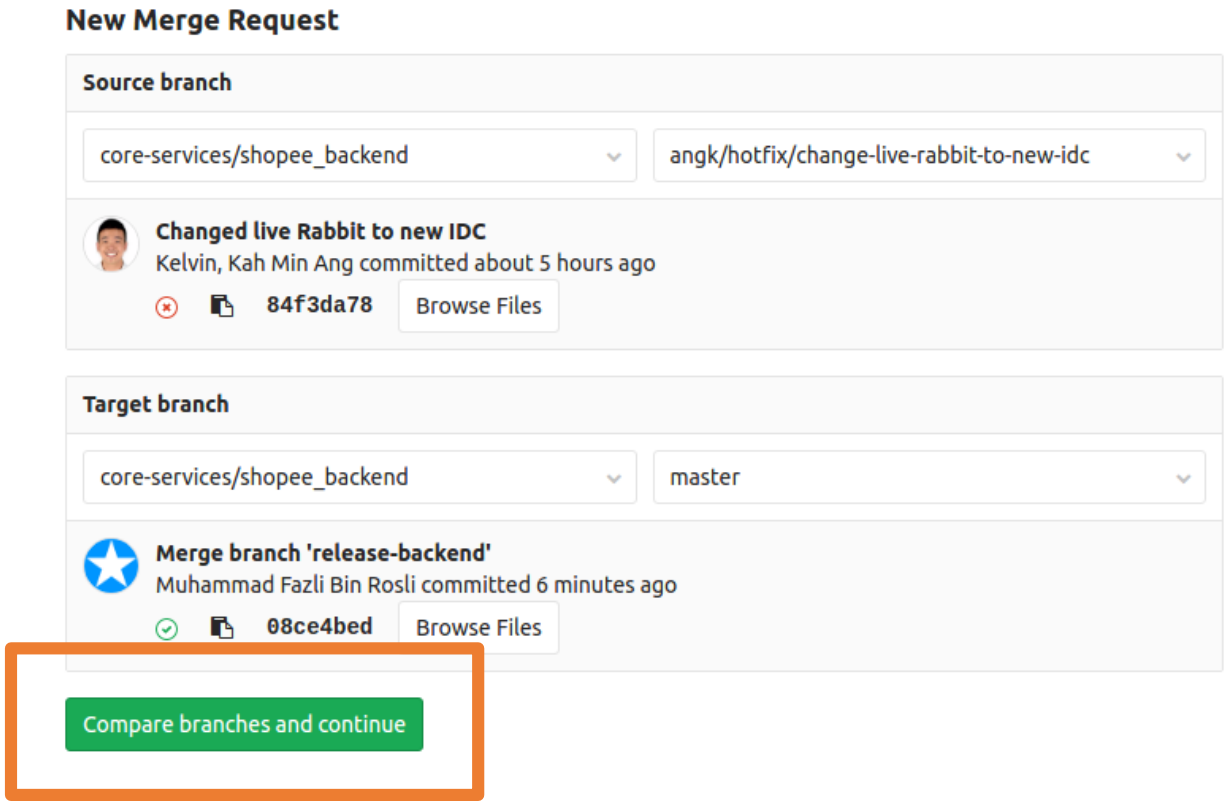
- Create a new merge request by clicking on the big green button.



Labs Flow

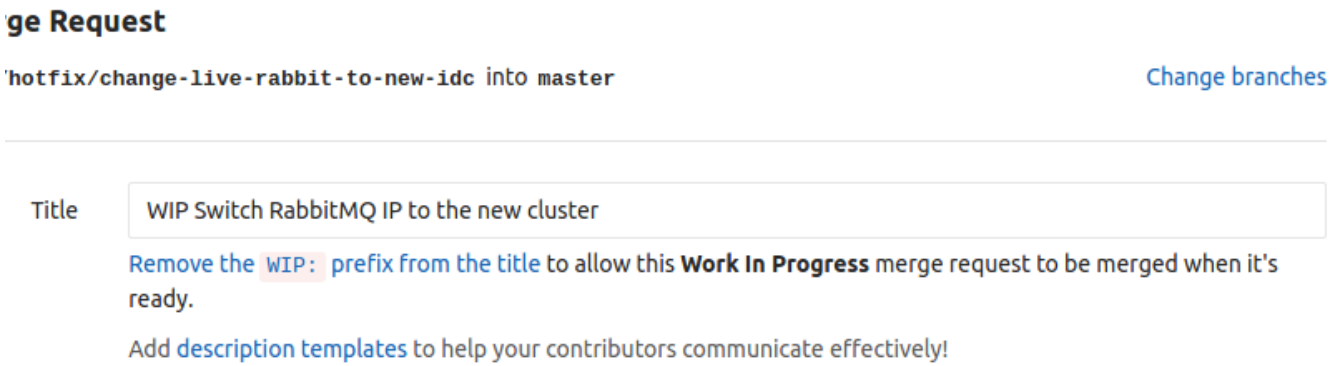
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- Choose correct source and target branches



Labs Flow

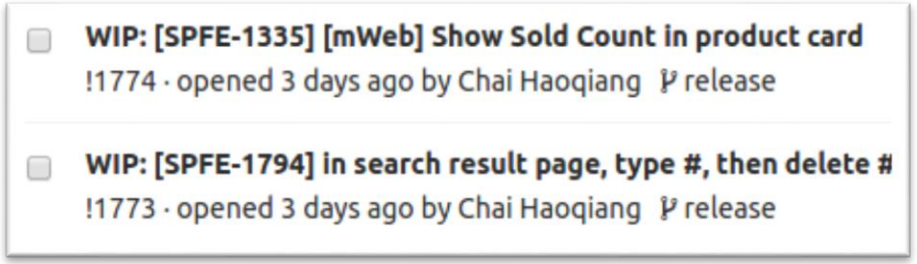
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…
- Fill in the title. Add "WIP" in the title if it is not ready to disable merging.



Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- For the release manager to track releases, if your team uses JIRA, you will have to add the JIRA ticket ID(s) into your Merge Request title.

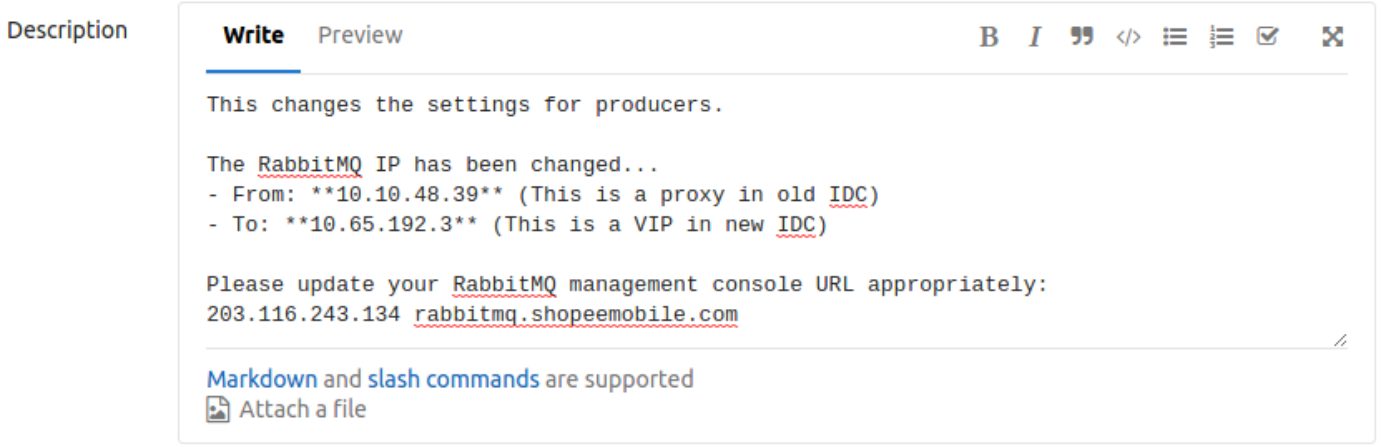


- It is recommended to have one ticket per merge request. If you have more, you have to list every ticket ID:

Labs Flow

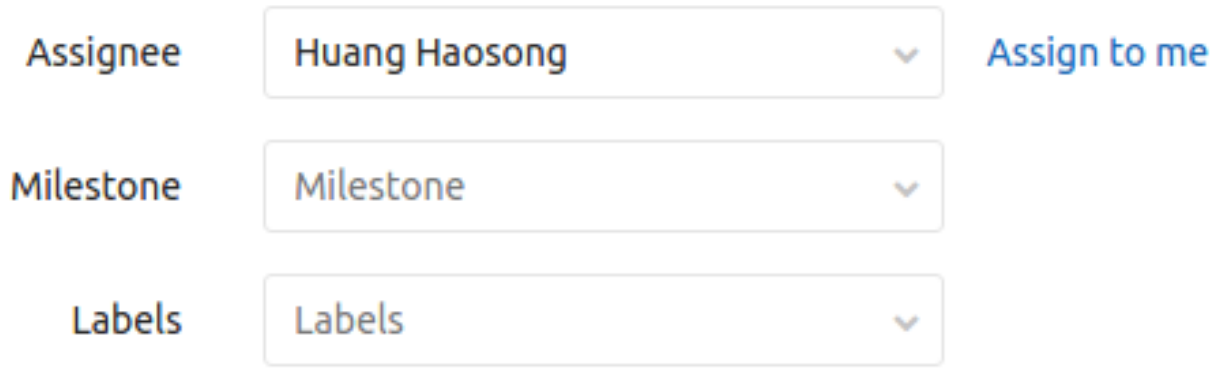
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- Fill in the content in markdown. Give as much details as appropriate to help with code review.



Labs Flow

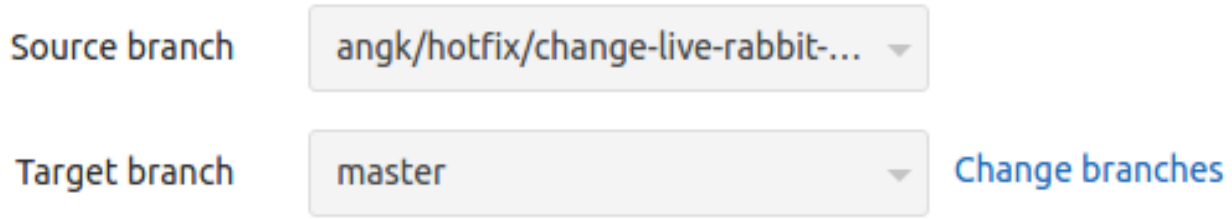
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…
- Assign the merge request to repository master for code review.



Labs Flow

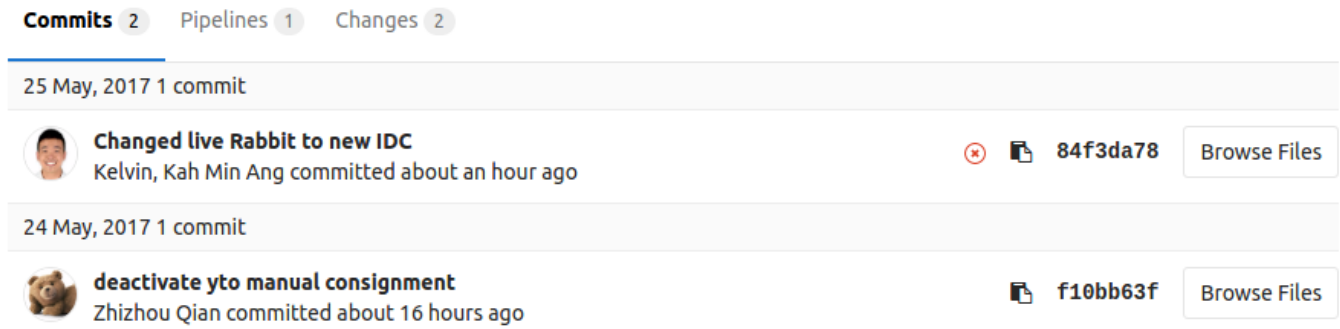
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- Check the source and target branches



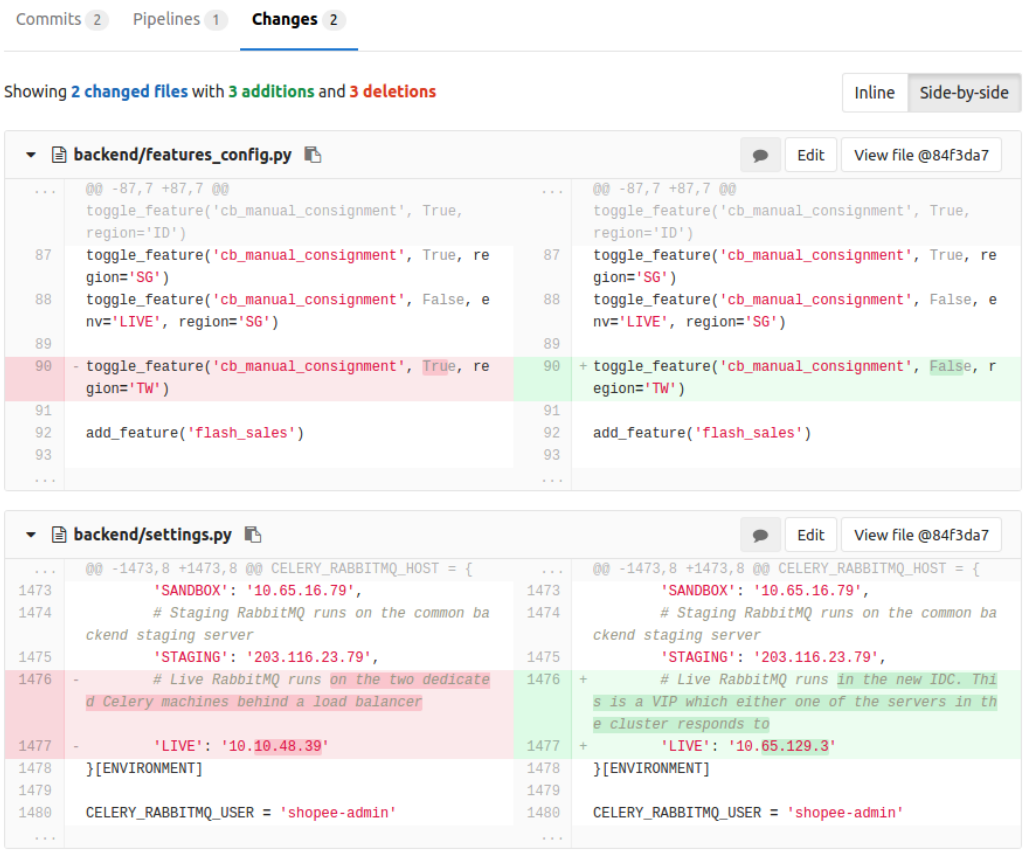
Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…
- Check the commits added to the target branch



Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…
- Check the changes being made to the target branch



Labs Flow

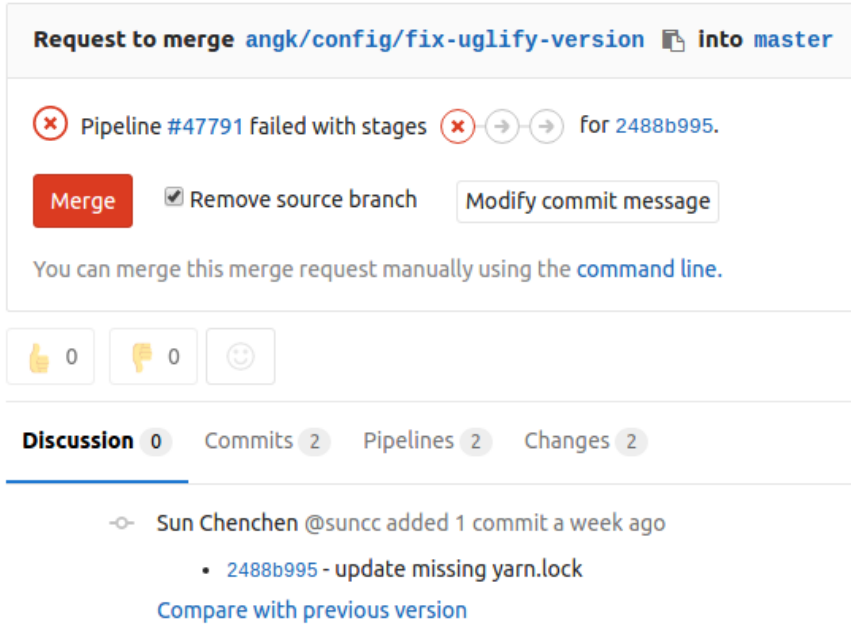
- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- When you are ready, click on the green button to submit your merge request.

Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- You will be brought to a merge request page that looks like this. Code review / automated checks will be done here and a master will merge your changes.



Labs Flow

- 1. Developers push code into own topic branch…
- **2. Merge requests are made to merge into environment branches…**
- 3. Environment branches are deployed to actual servers via Jenkins…

- To keep the repository tidy, remove your source branch if no other merge requests use it.

Labs Flow

- 1. Developers push code into own topic branch…
- 2. Merge requests are made to merge into environment branches…
- **3. Environment branches are deployed to actual servers via Jenkins…**

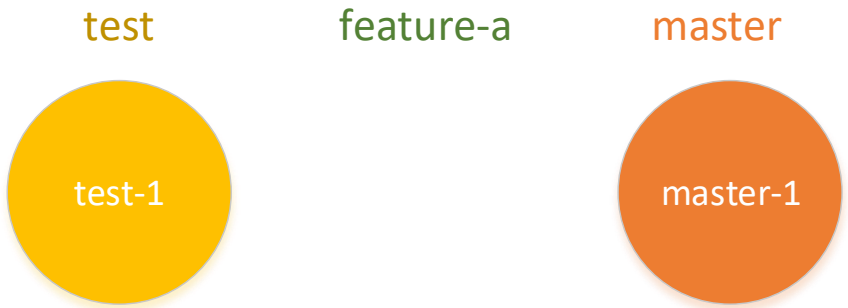
- Deploy to the environment via the appropriate Jenkins job when ready.

pnl | resolver | rn | sa | seller | sls | snr | sticker | webchat | website | +

S	W	Name ↓	Last Success	Last Failure	Last Duration
		shopee-admin-live	16 hr - tw--Zhizhou Qian	2 mo 1 day - #1646	1 min 14 sec
		shopee-admin-sandbox	1 mo 20 days - all--Fazli Sapuan	1 mo 20 days - sg--Fazli Sapuan	50 sec
		shopee-admin-staging	1 hr 24 min - all--Fazli Sapuan	28 days - all--Muhammad Fazli Bin Rosli	5 min 11 sec
		shopee-admin-test	7 min 45 sec - all--Junhong Gao	1 day 21 hr - all-Edward Sujono-Haosong Huang	38 sec
		shopee-celery-live	16 hr - tw--Zhizhou Qian	2 days 16 hr - all--Muhammad Fazli Bin Rosli	2 min 25 sec
		shopee-celery-staging	2 days 18 hr - all--Junhong Gao	5 days 19 hr - all-Xinzi Zhou-Haosong Huang	2 min 20 sec
		shopee-celery-test	17 hr - id--Van Quang Huynh	1 day 20 hr - all--Muhammad Fazli Bin Rosli	56 sec
		shopee-newcelery-live	2 hr 5 min - all--Kelvin, Kah Min Ang	1 day 1 hr - all--Kelvin, Kah Min Ang	7 min 51 sec
		shopee-restart-rabbitmq-live	1 day 22 hr - sg--Kelvin, Kah Min Ang	2 days 16 hr - sg--Kelvin, Kah Min Ang	31 sec
		shopee-taskqueue-live	2 mo 10 days - sg--Xian You Lim	15 days - sg-Xian You Lim-Haosong Huang	3 min 19 sec
		shopee-taskqueue-staging	9 days 0 hr - all--Kelvin, Kah Min Ang	9 days 0 hr - all--Kelvin, Kah Min Ang	1 min 44 sec
		shopee-taskqueue-test	5 days 14 hr - all--Kelvin, Kah Min Ang	5 days 14 hr - all--Kelvin, Kah Min Ang	36 sec
		shopee-webapi-live	16 hr - tw--Zhizhou Qian	18 hr - all--Muhammad Fazli Bin Rosli	2 min 2 sec
		shopee-webapi-sandbox	4 mo 14 days - all--Muhammad Fazli Bin Rosli	9 mo 11 days - sg--Haosong Huang	1 min 18 sec
		shopee-webapi-staging	18 hr - sg-Xian You Lim-Haosong Huang	21 days - #996	47 sec
		shopee-webapi-test	1 hr 3 min - sg-Zhizhou Qian-Haosong Huang	13 days - tw--Wenjie Chen	36 sec

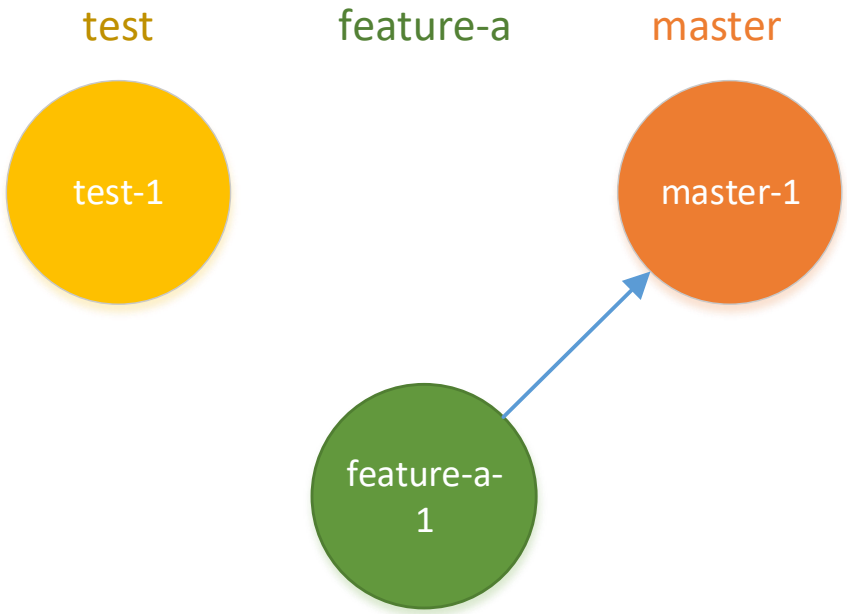
Labs Flow – Developing a new feature

- Check out a feature branch from master
- Merge it into test until you are ready
- Create a merge request to master and wait for review



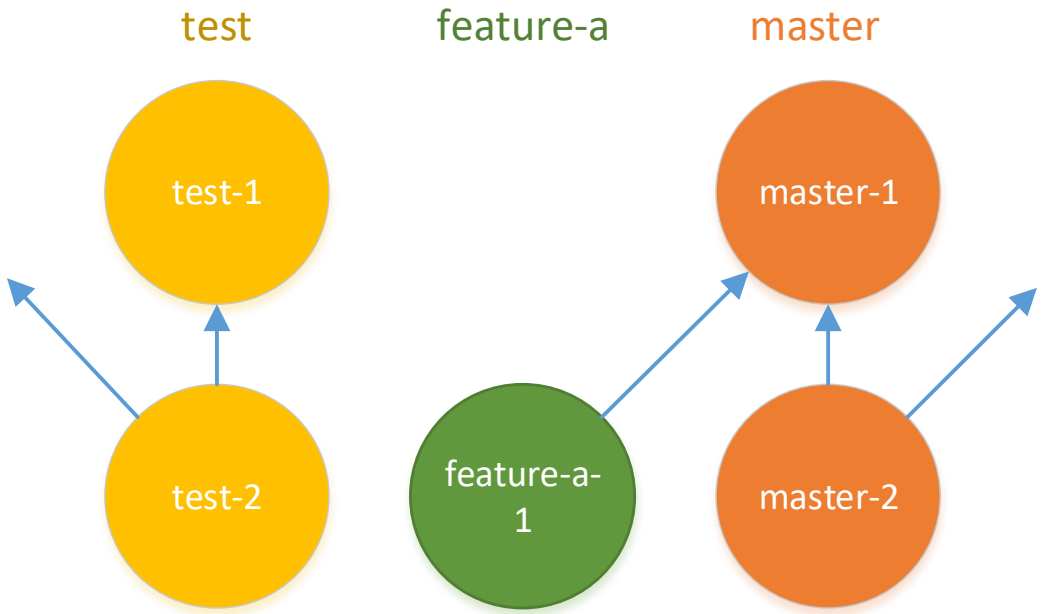
Labs Flow – Developing a new feature

- **Check out a feature branch from master**
- Merge it into test until you are ready
- Create a merge request to master and wait for review



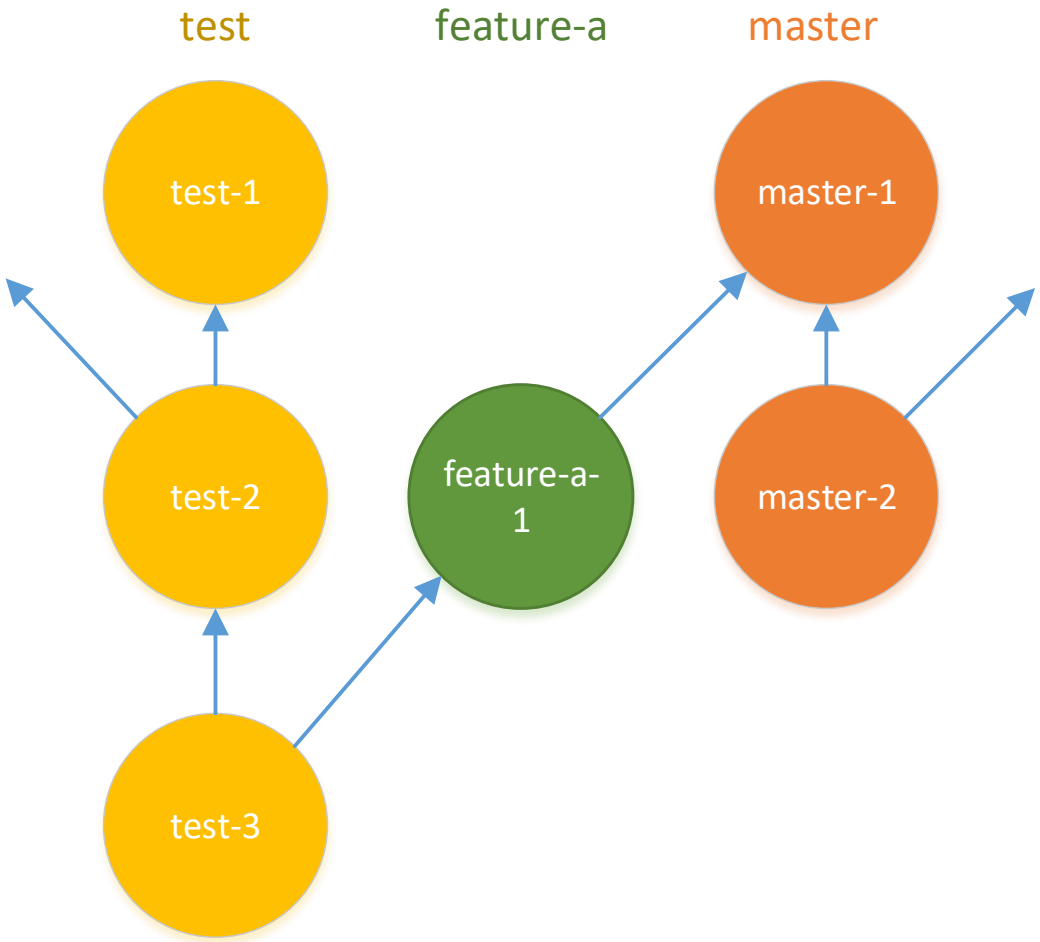
Labs Flow – Developing a new feature

- **Check out a feature branch from master**
- Merge it into test until you are ready
- Create a merge request to master and wait for review



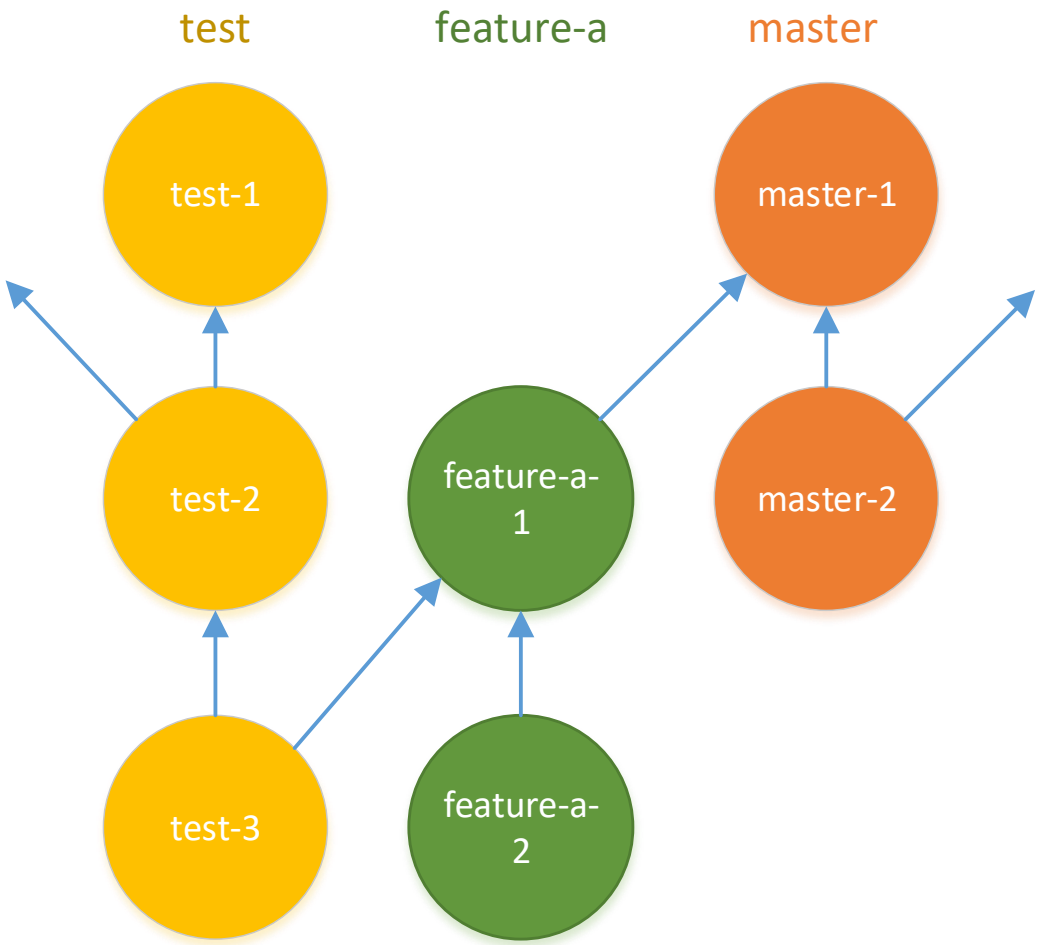
Labs Flow – Developing a new feature

- Check out a feature branch from master
- **Merge it into test until you are ready**
- Create a merge request to master and wait for review



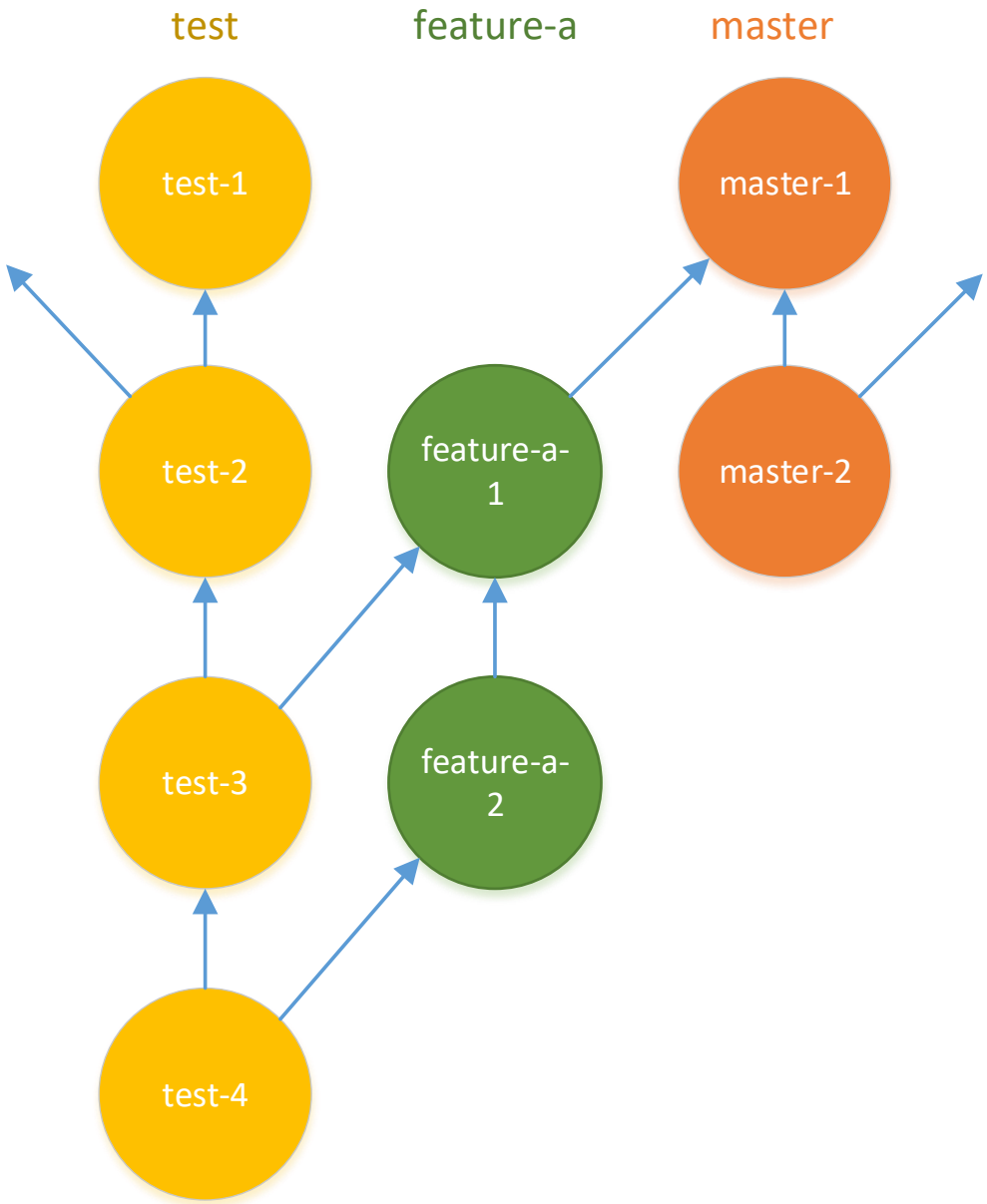
Labs Flow – Developing a new feature

- Check out a feature branch from master
- **Merge it into test until you are ready**
- Create a merge request to master and wait for review



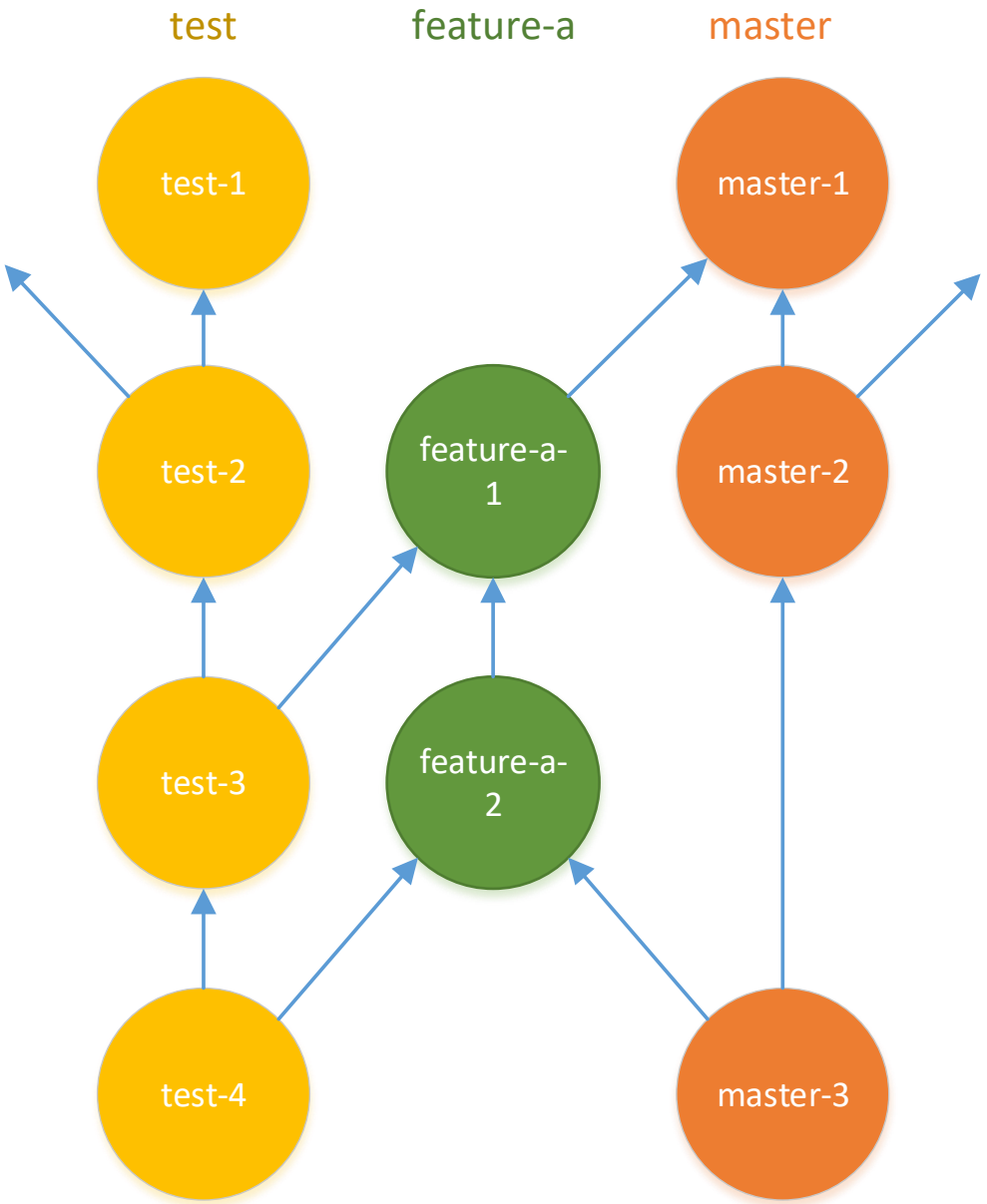
Labs Flow – Developing a new feature

- Check out a feature branch from master
- **Merge it into test until you are ready**
- Create a merge request to master and wait for review



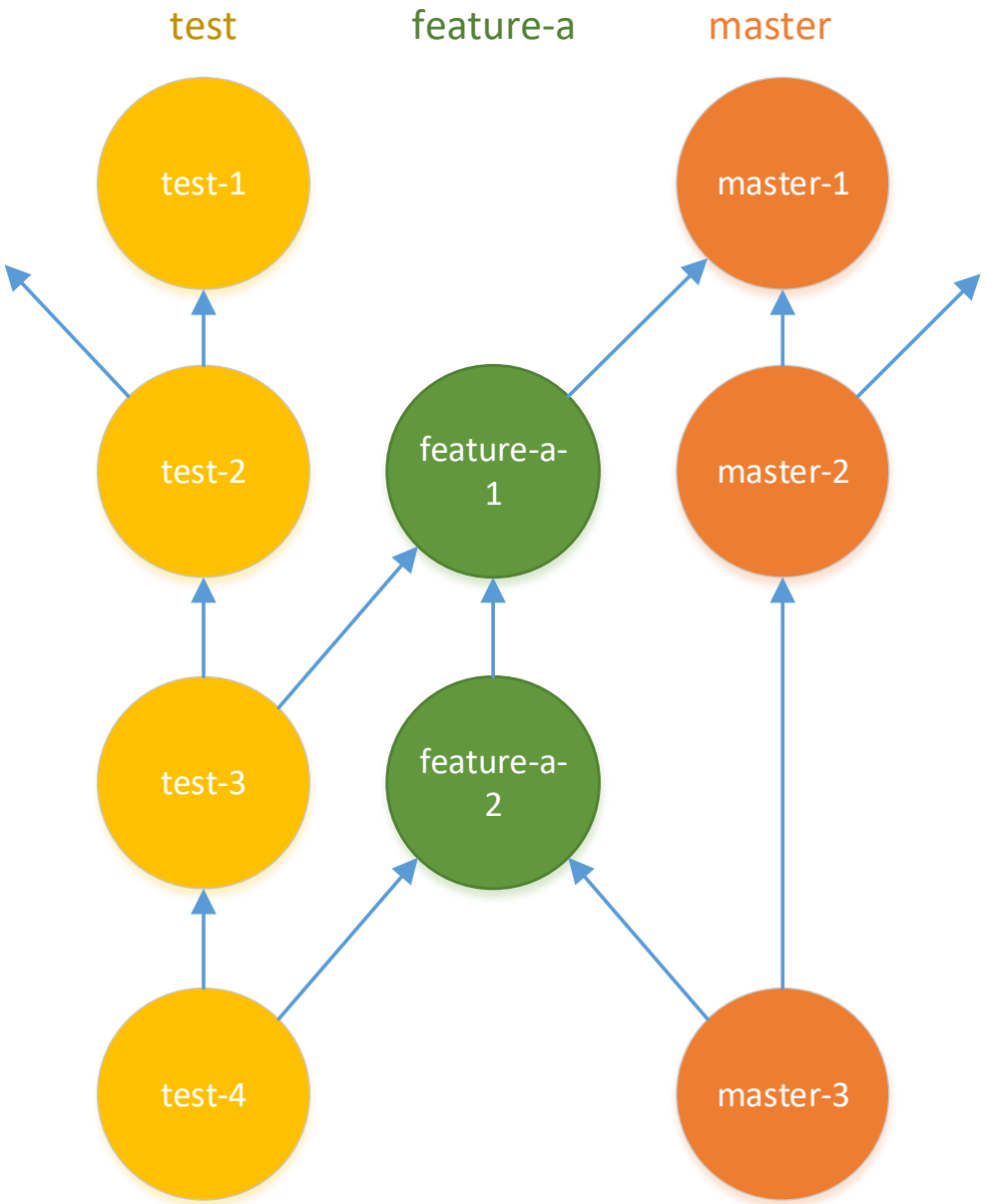
Labs Flow – Developing a new feature

- Check out a feature branch from master
- Merge it into test until you are ready
- **Create a merge request to master and wait for review**



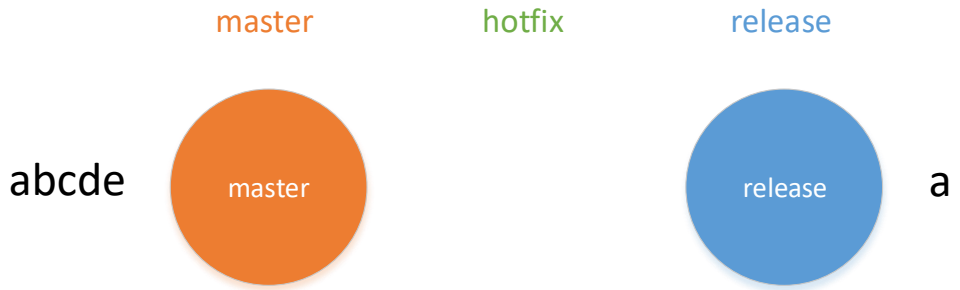
Labs Flow – Developing a new feature

- The merging from master to release will be done by the release manager in your team.
- This is the flow you should use 95% of the time.



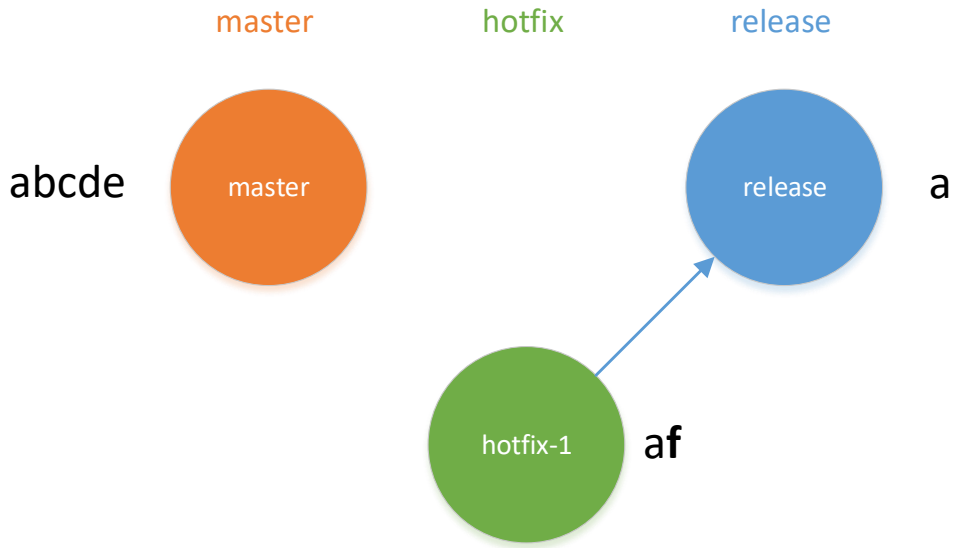
Labs Flow – Making a Hotfix

- Creating a hotfix is similar to creating a feature.



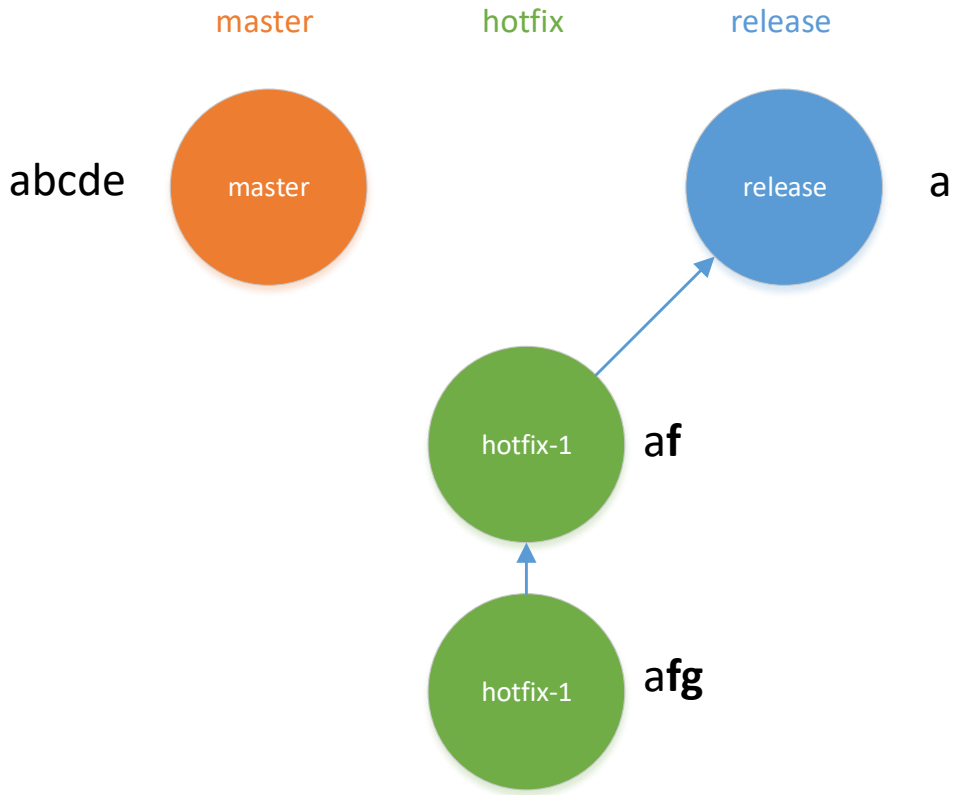
Labs Flow – Making a Hotfix

- First, you check out from release (instead of master, in the case of a feature)



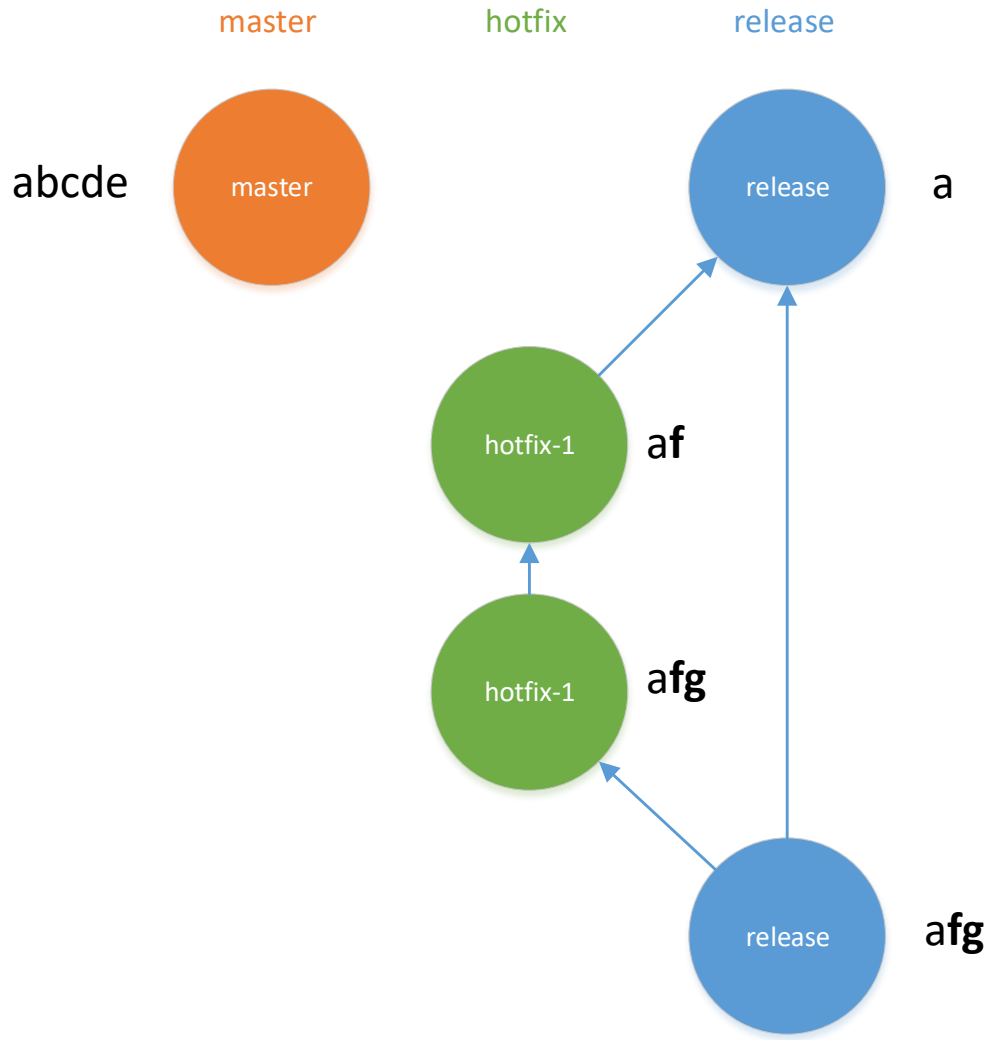
Labs Flow – Making a Hotfix

- After making some developments and testing appropriately (usually in live-ish or staging environment), you are ready to merge it back.



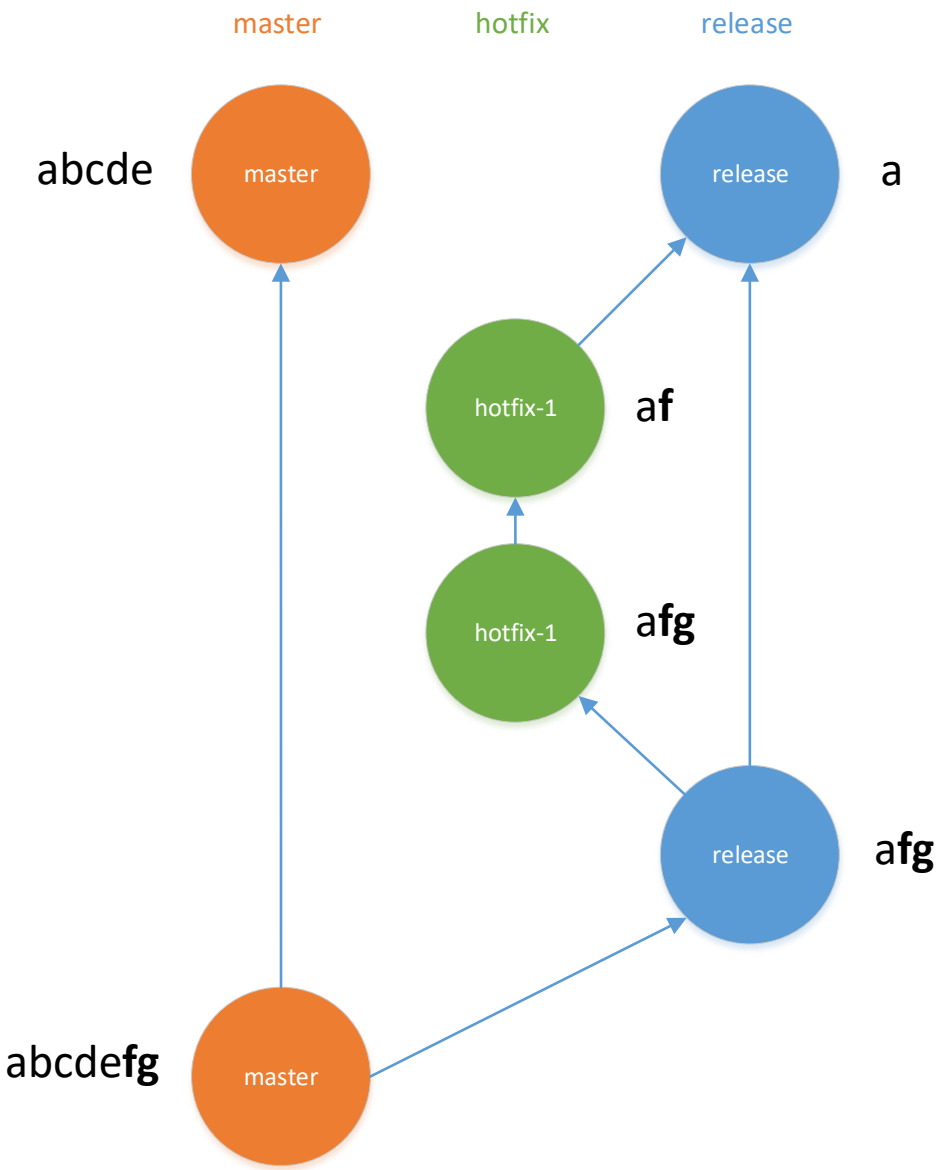
Labs Flow – Making a Hotfix

- You merge your hotfix into release branch.



Labs Flow – Making a Hotfix

- And merge release back into master branch.
- Why? (Answer on the next slide)



Rules (more in the Wiki page)

- Developers MUST create feature branches for development.
- All features branches must be merged via merge requests.
- The release branch must be a subset of master branch. It cannot contain commits/changes that are not in master branch.
- test branch should NEVER be merged to any other branch.
- All merge requests for master and beyond should be reviewed by one or more repository master(s).
- Each release must be tagged using the Semantic Versioning Standard, and the version should always increase.

Rules (more in the Wiki page)

- New projects must use your team's namespace if you are developing for your team.
- Please request for a project repository by your team's GitLab master.

- Wrong:

angk/shopee-web ✗

- Correct:

shopee-server/shopee-web ✓

Rules (more in the Wiki page)

- Name branches accordingly to the following format:

`<email-prefix>/<type>/<description>`

email-prefix:

angk@seagroup.com

- Wrong:

kelvin/Add_Some_Scripts ✗

- Correct:

angk/feature/initial-deployment-scripts ✓

Rules (more in the Wiki page)

- Name branches accordingly to the following format:

`<email-prefix>/<type>/<description>`

type:

- **feature**: major features, UI updates, configuration updates, etc.

- **bugfix**: fixing a bug that follows the normal deployment flow

- **hotfix**: fixing an urgent bug directly from release

- **refactor**: restructuring of code that do not change external behavior but improves code quality

- Wrong:

kelvin/Add_Some_Scripts ✗

- Correct:

angk/feature/initial-deployment-scripts ✓

Rules (more in the Wiki page)

- Name branches accordingly to the following format:

`<email-prefix>/<type>/<description>`

description:

A useful description that encapsulates the intent of the branch.

- Wrong:

kelvin/Add_Some_Scripts ✗

- Correct:

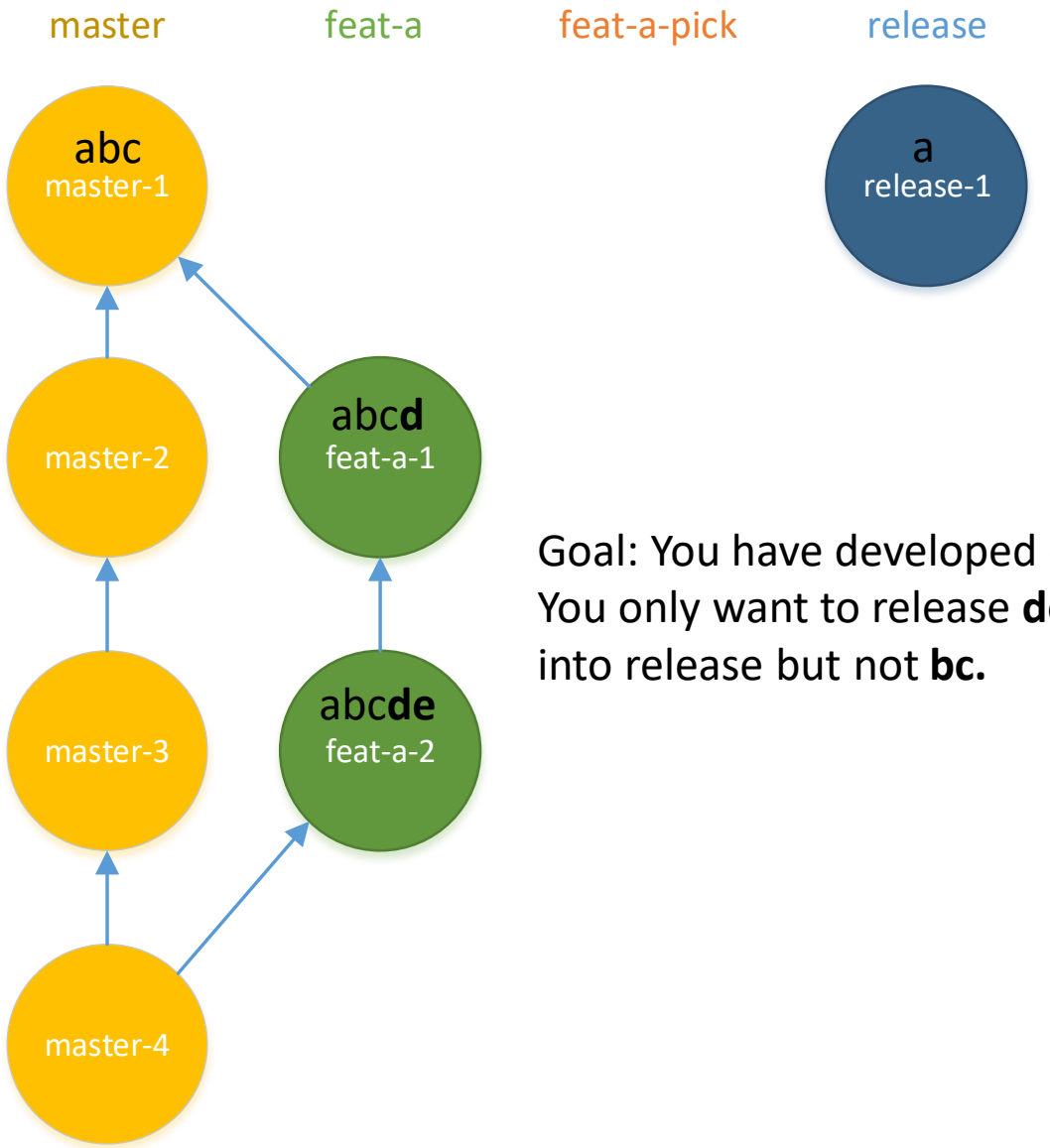
angk/feature/initial-deployment-scripts ✓

Labs Flow – Cherry-picking

- Suppose master branch contains 10 features that are not in release.
- You check master out to develop a feature.
- Suddenly, you realize that your feature needs to go live urgently.
- If you simply merge master to release, then all 10 other features will go live as well.
- This is where cherry-picking comes in handy.
- This allows you to "partially merge" a branch into another.

GarenaLabs Flow – Cherry-picking

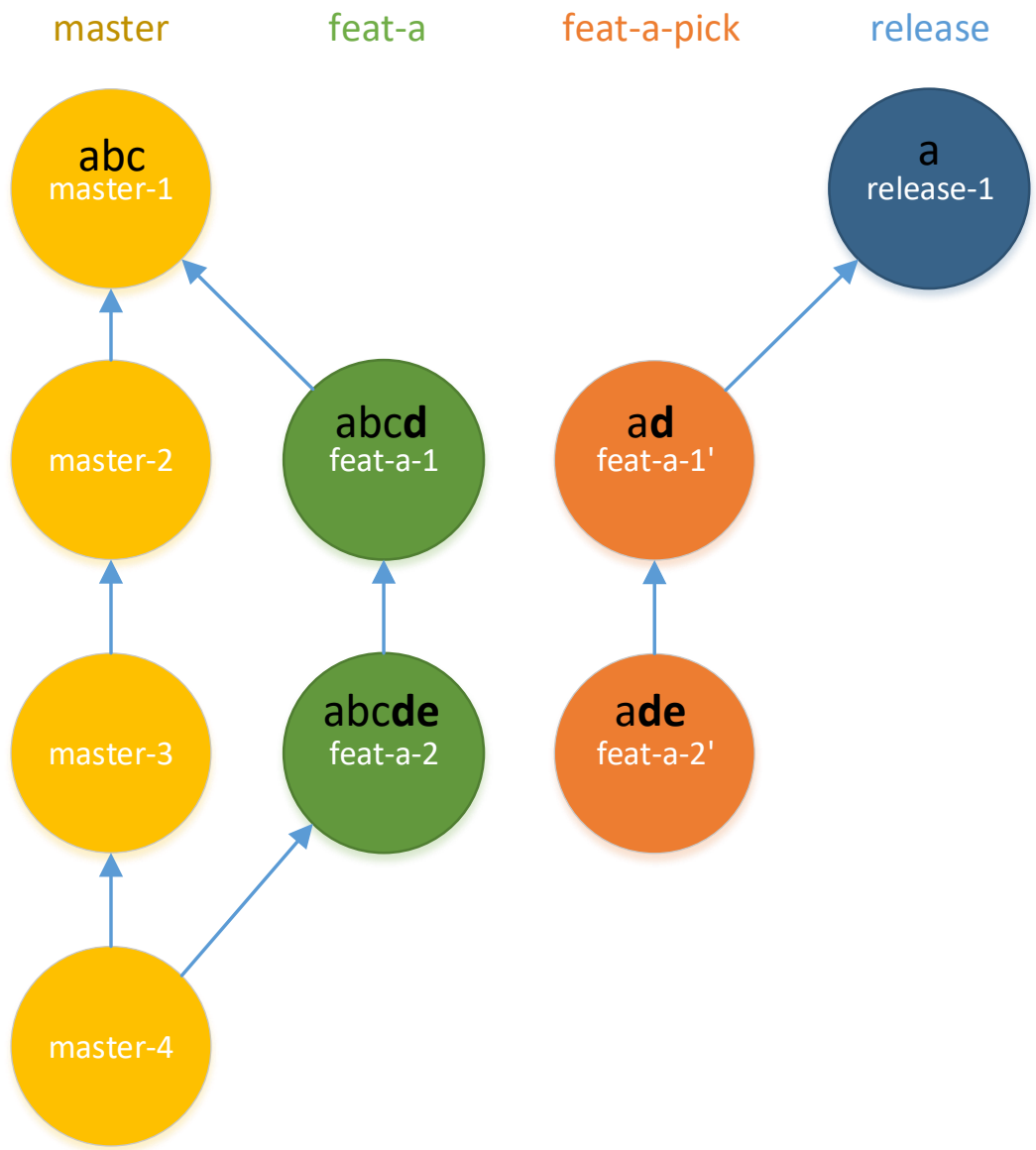
- Suppose that you have already merged your feature branch to master branch (important)
- **You check out a hotfix branch from release.**
- Then you cherry-pick the commits required from your feature branch into the release branch
- Finally, you create a merge request to merge your hotfix branch into release. (No need to merge hotfix back into master, as the changes were already merged by your feature branch)



Goal: You have developed **de.** You only want to release **de** into release but not **bc.**

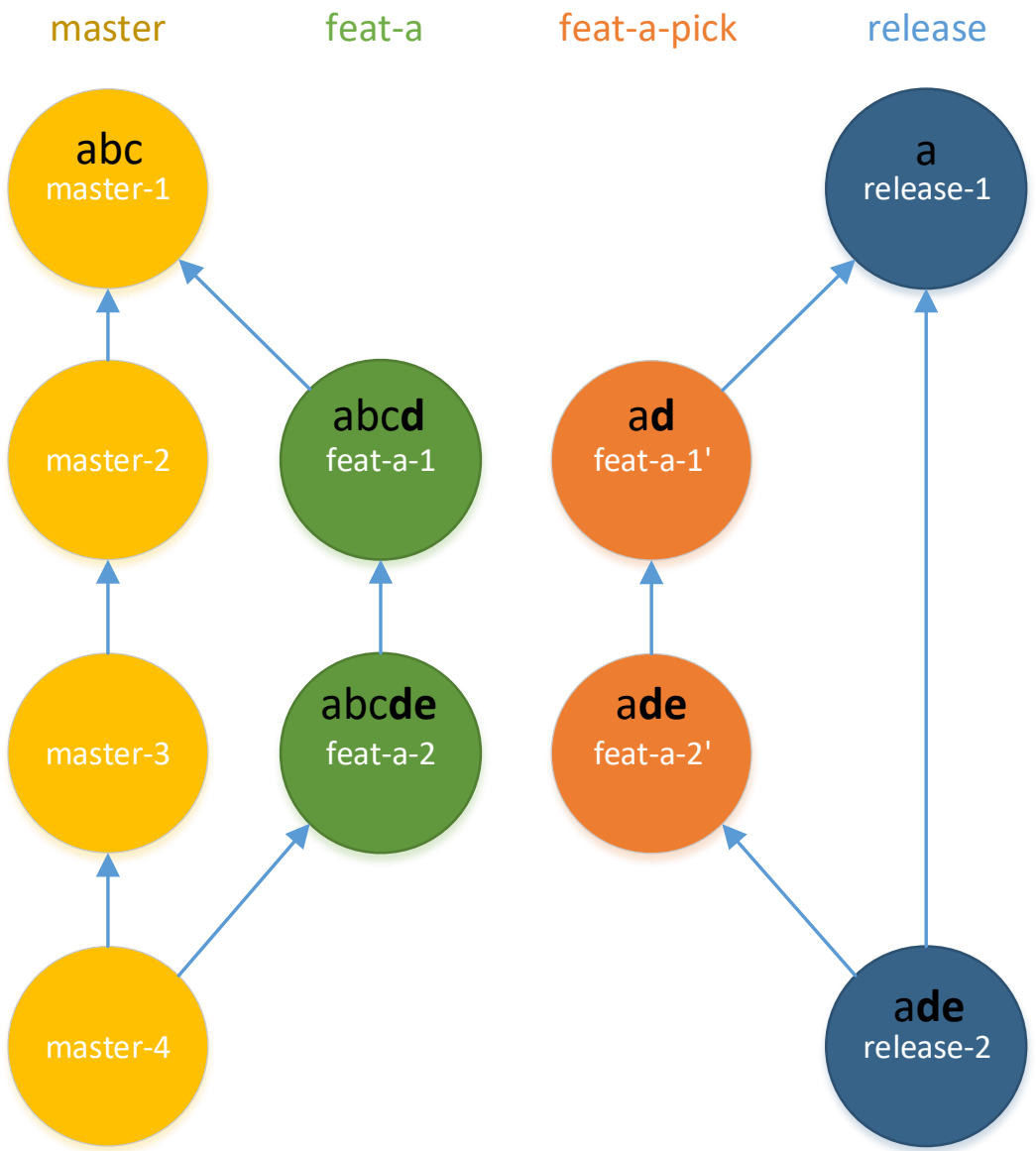
GarenaLabs Flow – Cherry-picking

- Suppose that you have already merged your feature branch to master branch (important)
- You check out a hotfix branch from release.
- **Then you cherry-pick the commits required from your feature branch into the release branch**
- Finally, you create a merge request to merge your hotfix branch into release. (No need to merge hotfix back into master, as the changes were already merged by your feature branch)



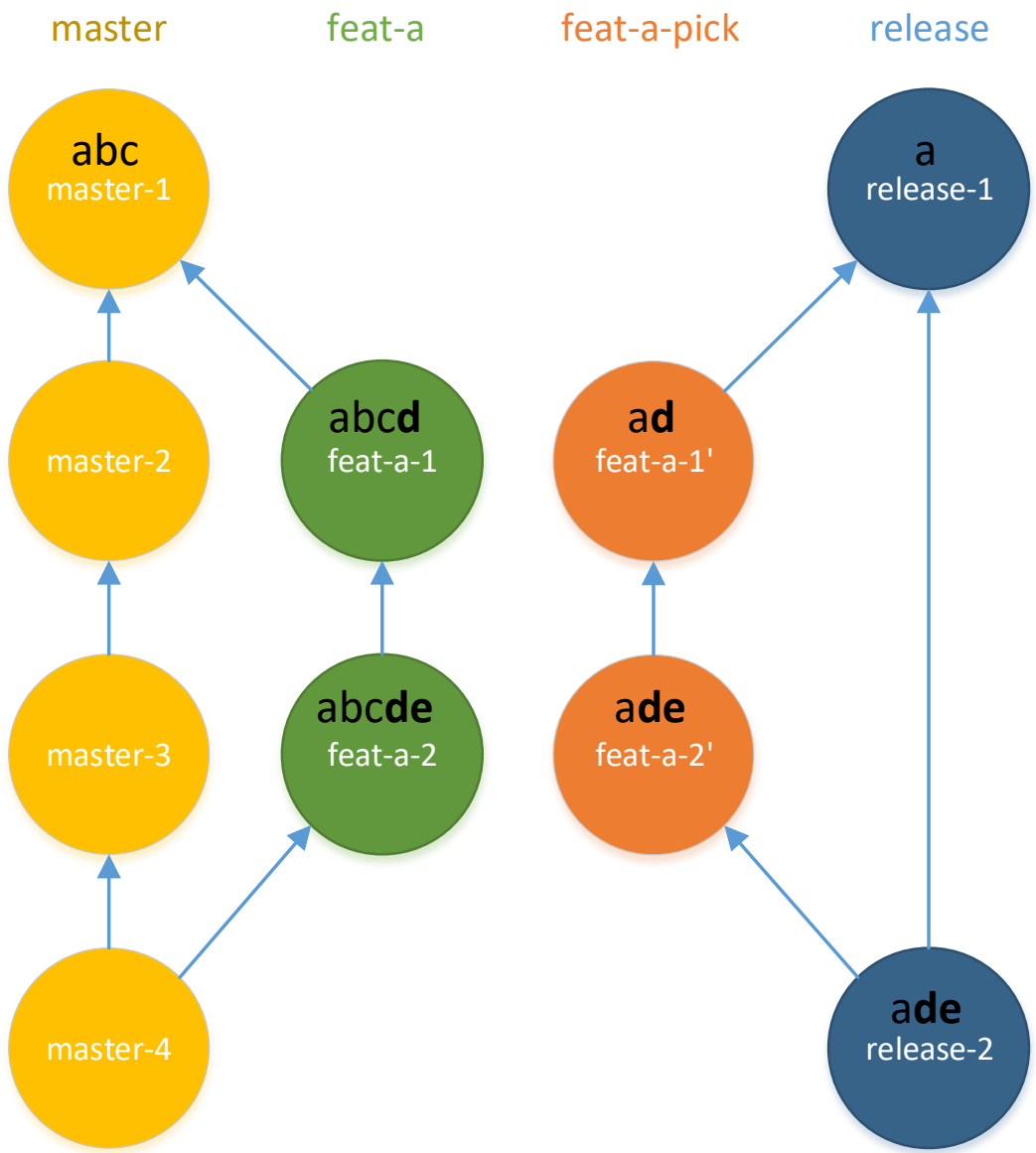
GarenaLabs Flow – Cherry-picking

- Suppose that you have already merged your feature branch to master branch (important)
- You check out a hotfix branch from release.
- Then you cherry-pick the commits required from your feature branch into the release branch
- **Finally, you create a merge request to merge your hotfix branch into release.** (No need to merge hotfix back into master, as the changes were already merged by your feature branch)



GarenaLabs Flow – Cherry-picking

- Notice that feat-a-1' and feat-a-2' are **completely different commits** from feat-a-1 and feat-a-2.
- Cherry-picking works by replaying the "diff" or "patch set" introduced by the picked commits.
- There will be no conflicts as long as the contents are the same when master is merged into release later.



How to graduate from this course...

1. Register for a GitLab account.
2. Set up your SSH keys in your computer if you haven't.
3. Add your SSH key to your GitLab account.
4. Email me (angk@garena.com) your GitLab account ID (it should be your email)

You will receive more instructions in the email :D

Thank you!

- Now you are ready to work with Git!

- Please refer to the Git guide in the Wiki for more details!

https://git.garena.com/garenalabs/labs_dev/wikis/git_guide