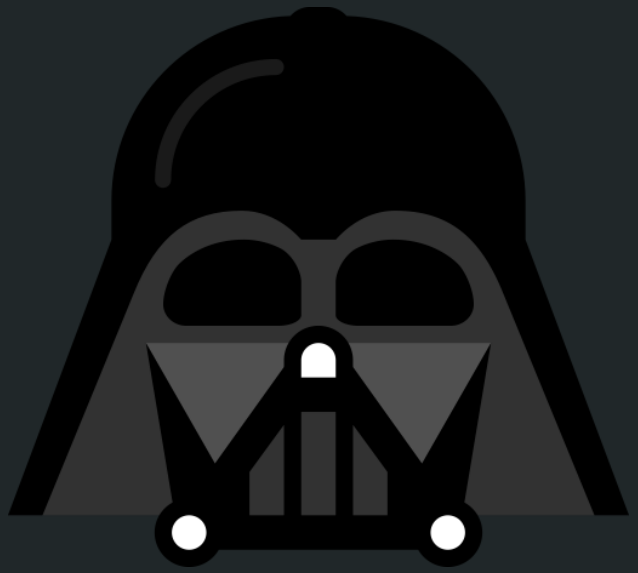
The Dark Side of Git

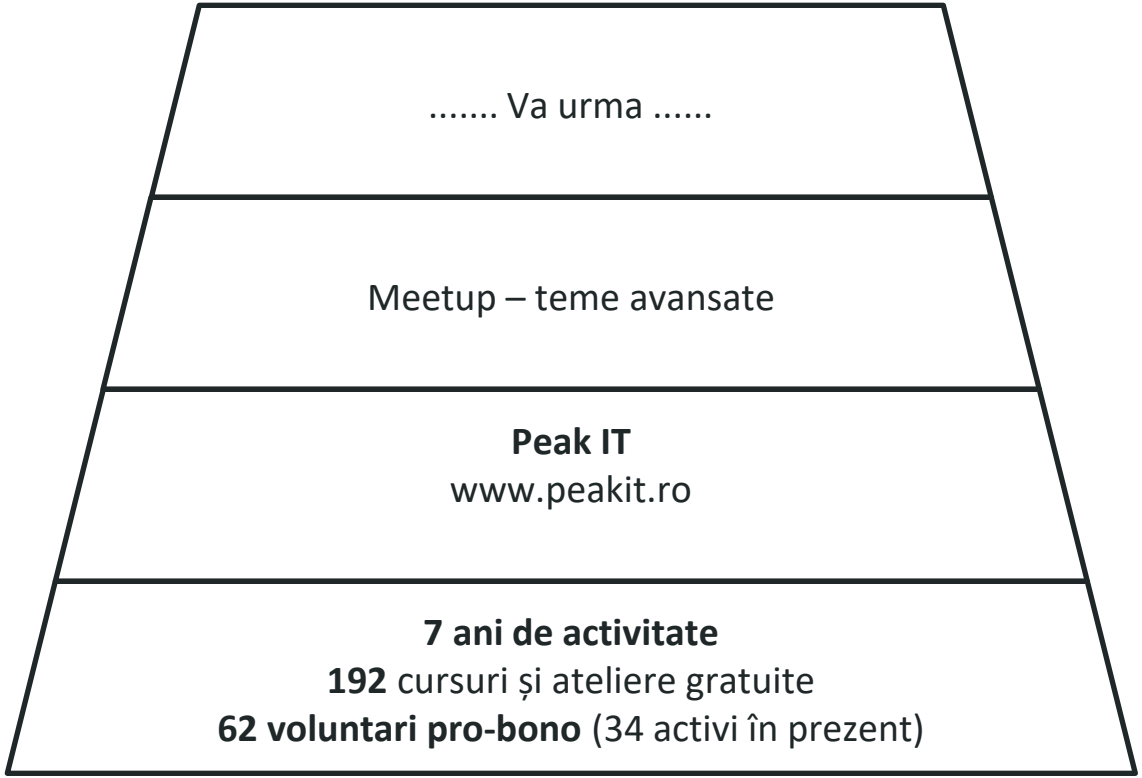


Or how to not shoot yourself in the foot while being a badass Git Jedi.

First up

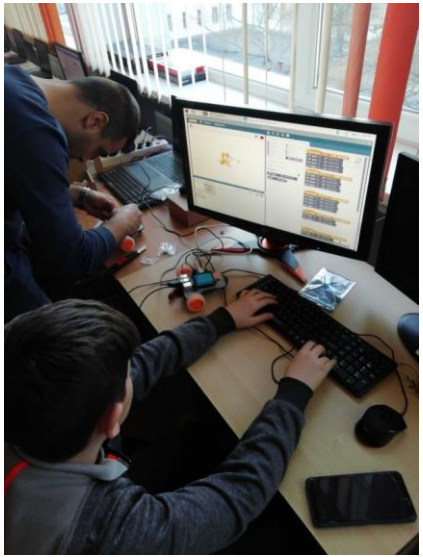
A message from our sponsor ☺

AgileHub


..... Va urma .....
Meetup – teme avansate
Peak IT
www.peakit.ro
7 ani de activitate
192 cursuri și ateliere gratuite
62 voluntari pro-bono (34 activi în prezent)

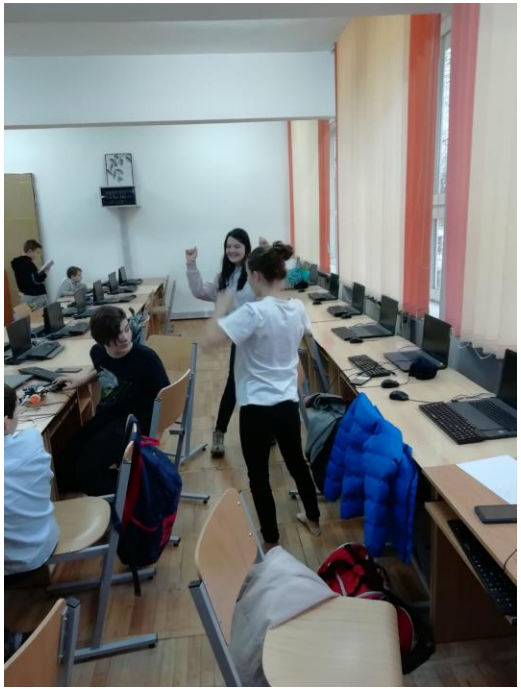
AgileHub investește în educația **gratuită**

Curs robotica, durata 8 luni, pentru 2 grupe de elevi de la Școala 11, Brașov



AgileHub investește în educația **gratuită**

Curs robotica, durata 8 luni, pentru 2 grupe de elevi de la Școala 11, Brașov



Did you ever have trouble with Git?

Did you find yourself:

- Copying and pasting files **to other directories and back** to your repo?
- Deleting branches **only to check them out again** from origin right after?
- Needing to **push --force** your changes
- Not knowing **how to undo some changes** you made
- Cherry picking **on a regular basis**
- Following some best practices someone laid out **blindly**?

I’m Paul Negoescu

And I want to address all this with you today!

Let’s find out what’s behind the git commands and understand the more advanced practices like:

- Resetting
- Reverting
- Rebasing
- Squashing
- Being Interactive
- Cherry Picking
- Branching models
- Forcing

RISE OR FALL

Git Basics Recap

What is Git?

A version control tool, for tracking changes in source code and enabling collaboration, coordination and integrity.

Important Concepts:

- Staging area or index
- Working directory
- Repository
- Commit
- Branch
- Remote

Git Basics Recap

Flow of Files

Flow of Files

```
$ |
```

Working Directory

Readme.md

Flow of Files

```
$ git init

$ |
```

Local Repository

Staging Area

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ |
```

Local Repository

Staging Area

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ |
```

Local Repository

Staging Area

Readme.md

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ git commit

$ |
```

Local Repository

Staging Area


Readme.md

Working Directory


Readme.md

Flow of Files

```
$ git init

$ git add

$ git commit

$ |
```

Local Repository

Readme.md

Staging Area

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ git commit

$ git remote add

$ |
```

Remote Repository

Local Repository

Readme.md

Staging Area

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ git commit

$ git remote add

$ git push
```

Remote Repository

Local Repository

Readme.md

Staging Area

Working Directory

Readme.md

Flow of Files

```
$ git init

$ git add

$ git commit

$ git remote add

$ git push
```

Remote Repository

Readme.md

Local Repository

Readme.md

Staging Area

Working Directory

Readme.md

Git Basics Recap

Branching

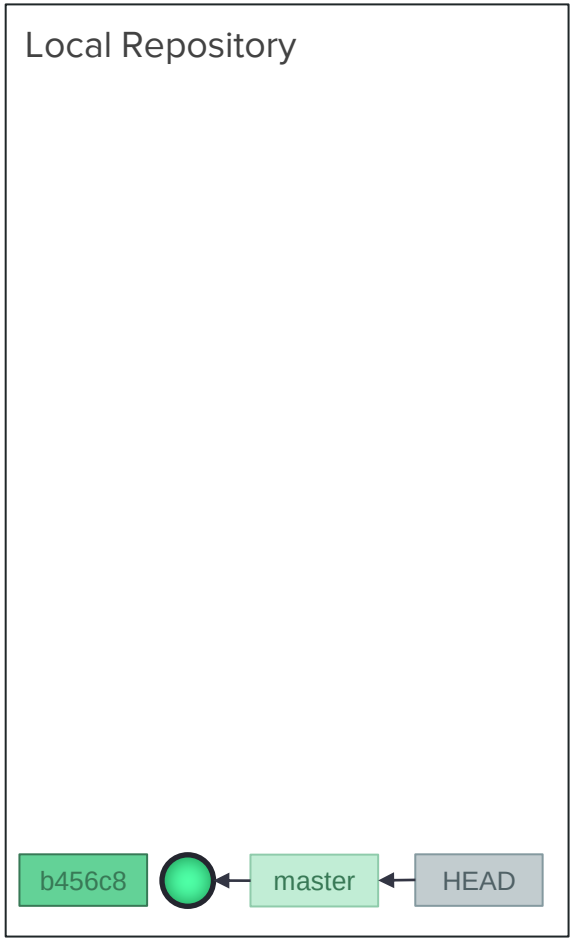
Branching

```
$
```

Local Repository

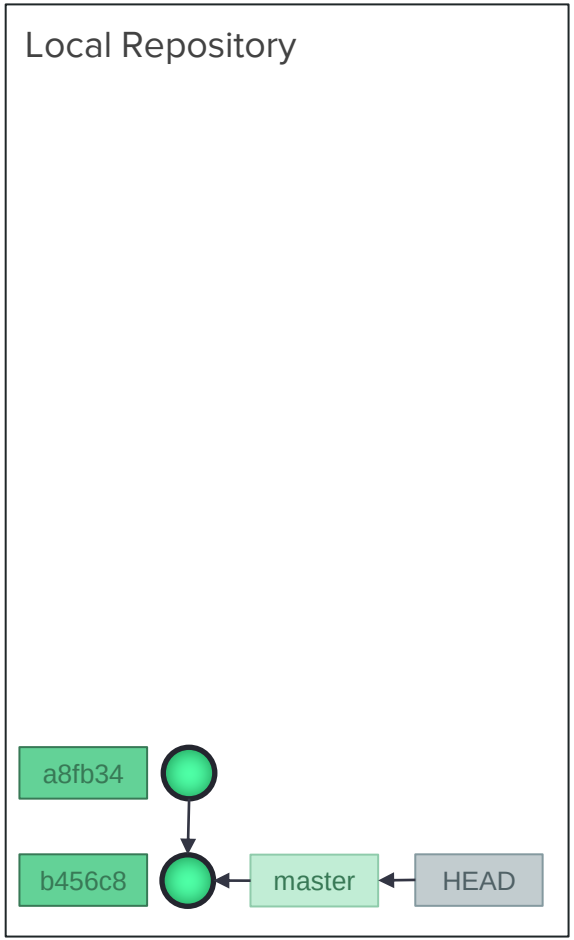
Branching

```
$ git commit

$ |
```



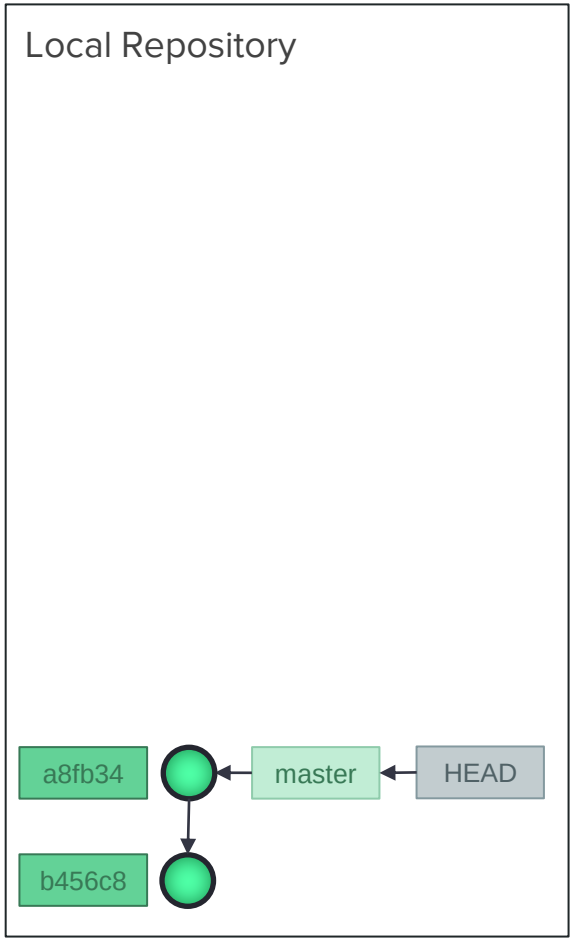
Branching

```
$ git commit

$ git commit

$ |
```



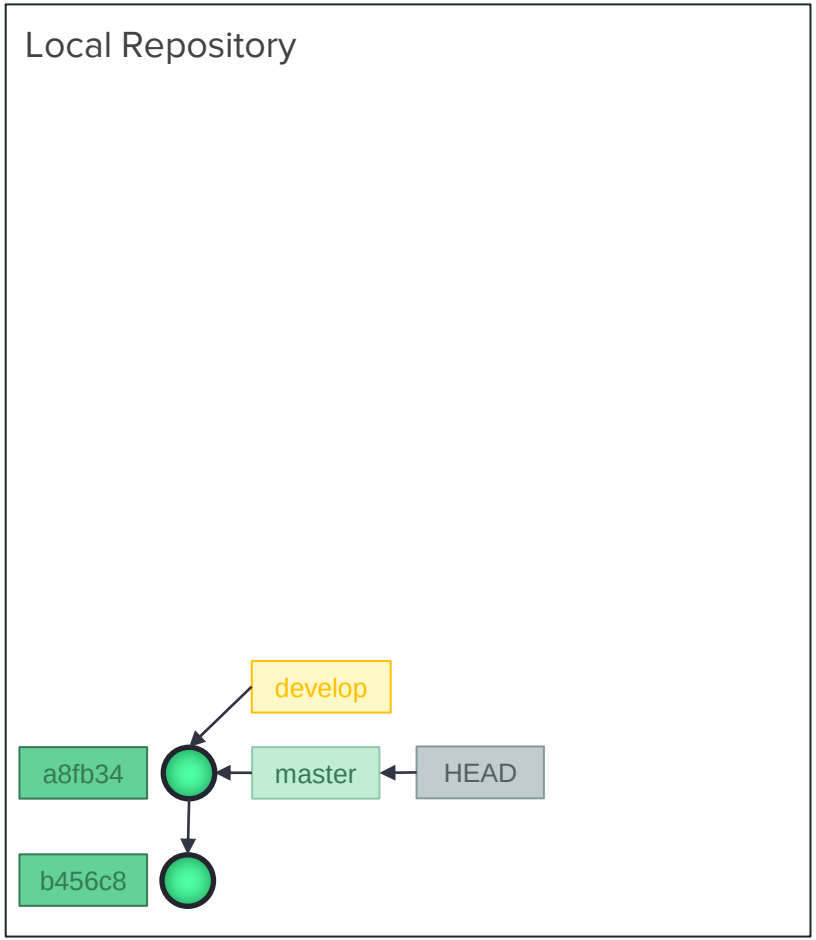
Branching

```
$ git commit

$ git commit

$ |
```



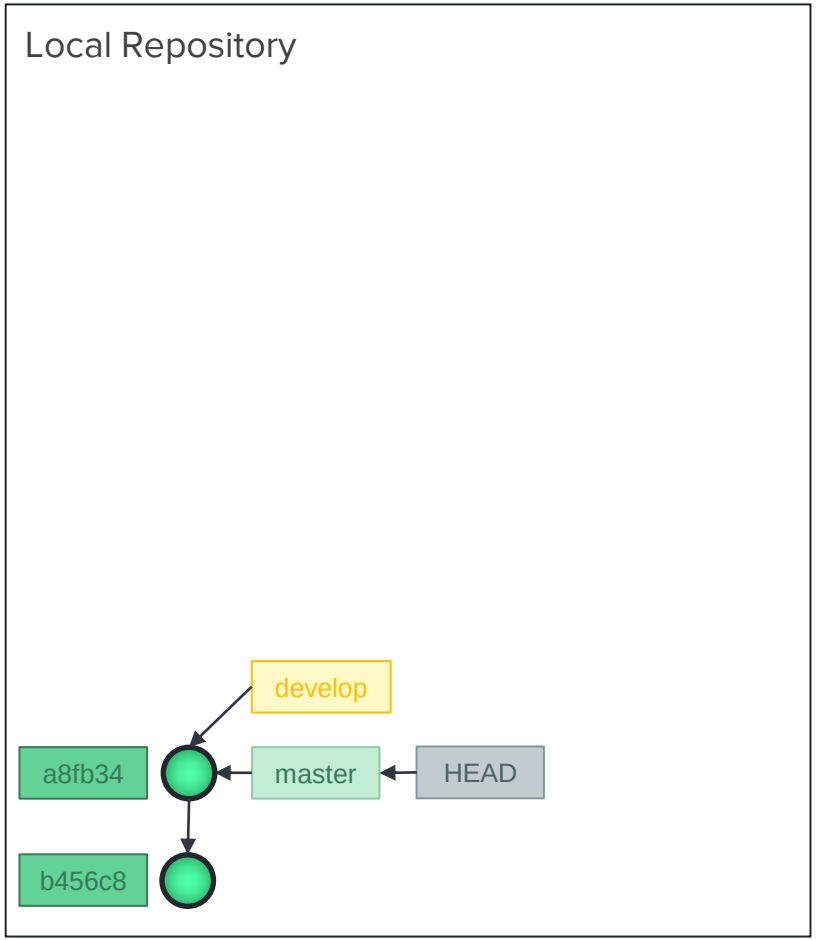
Branching

```
$ git commit

$ git commit

$ git branch develop

$ |
```



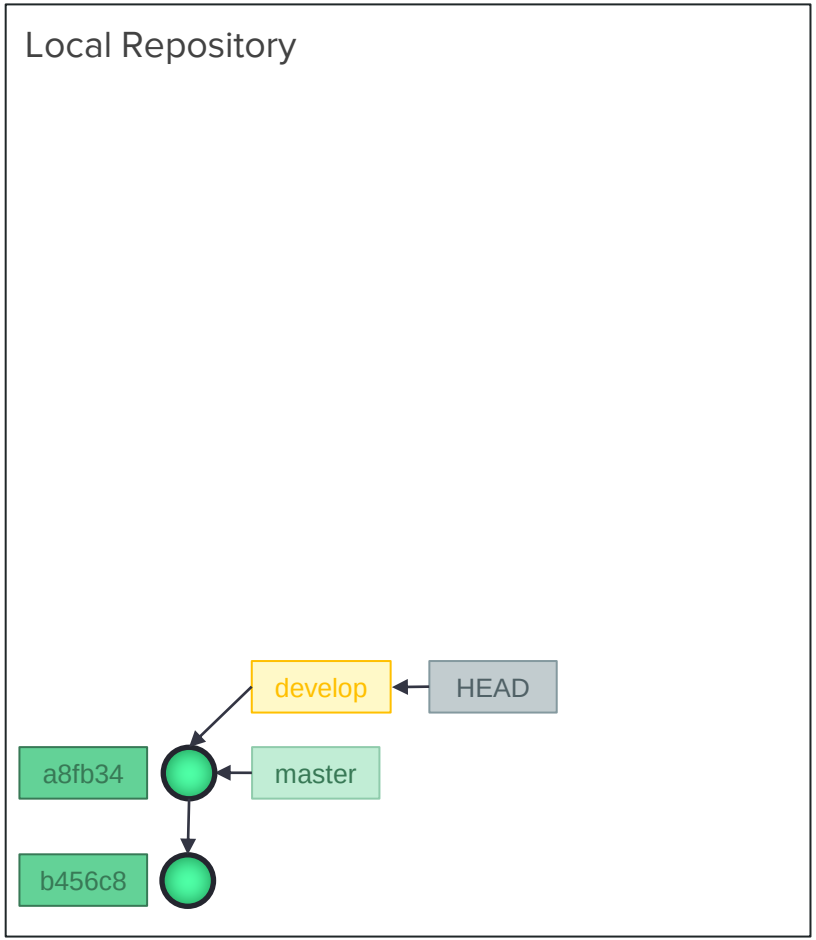
Branching

```
$ git commit

$ git commit

$ git branch develop

$ git checkout develop

$ |
```



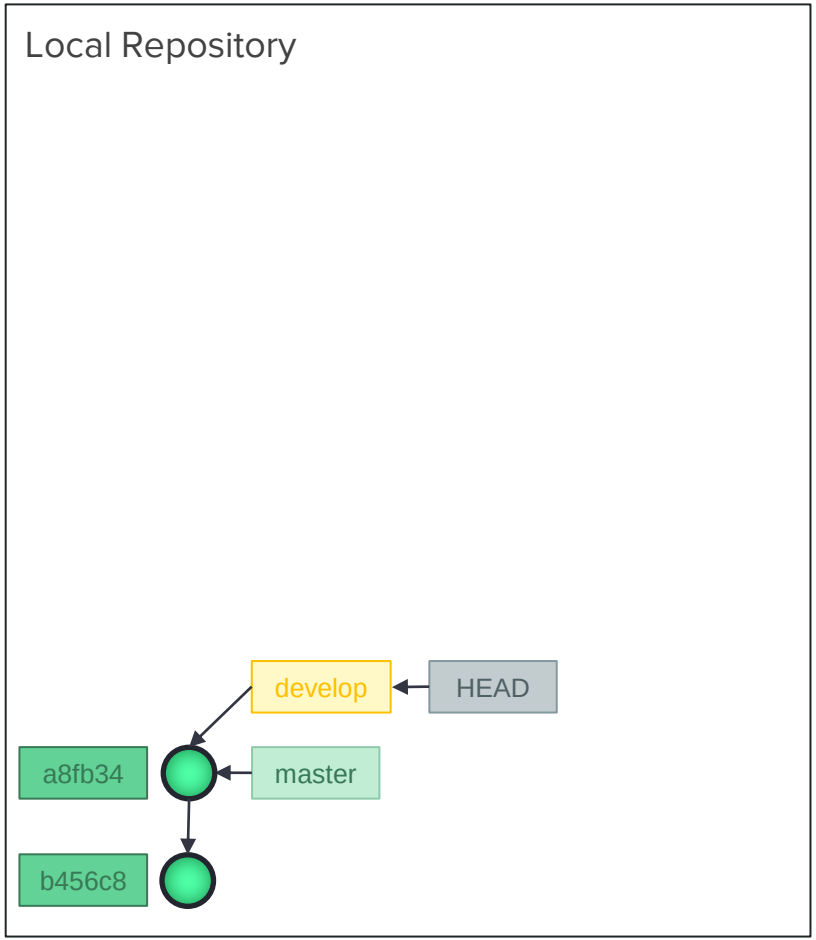
Branching

```
$ git commit

$ git commit

$ git branch develop

$ git checkout develop

$ |
```



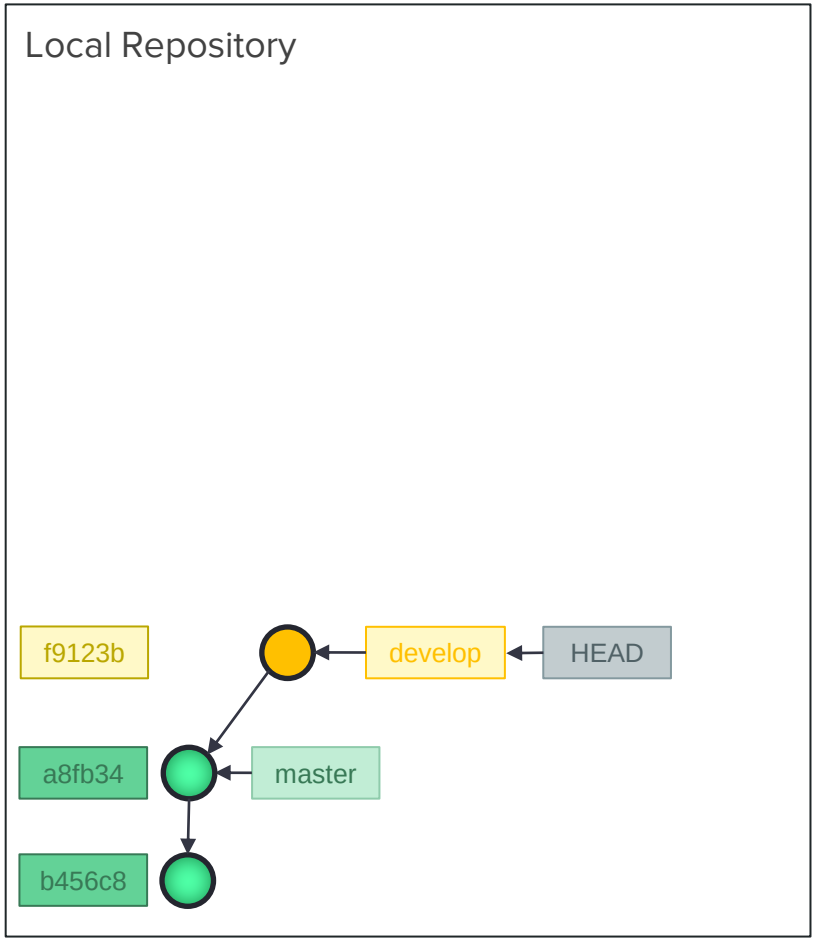
Branching

```
$ git commit

$ git branch develop

$ git checkout develop

$ git commit

$ |
```



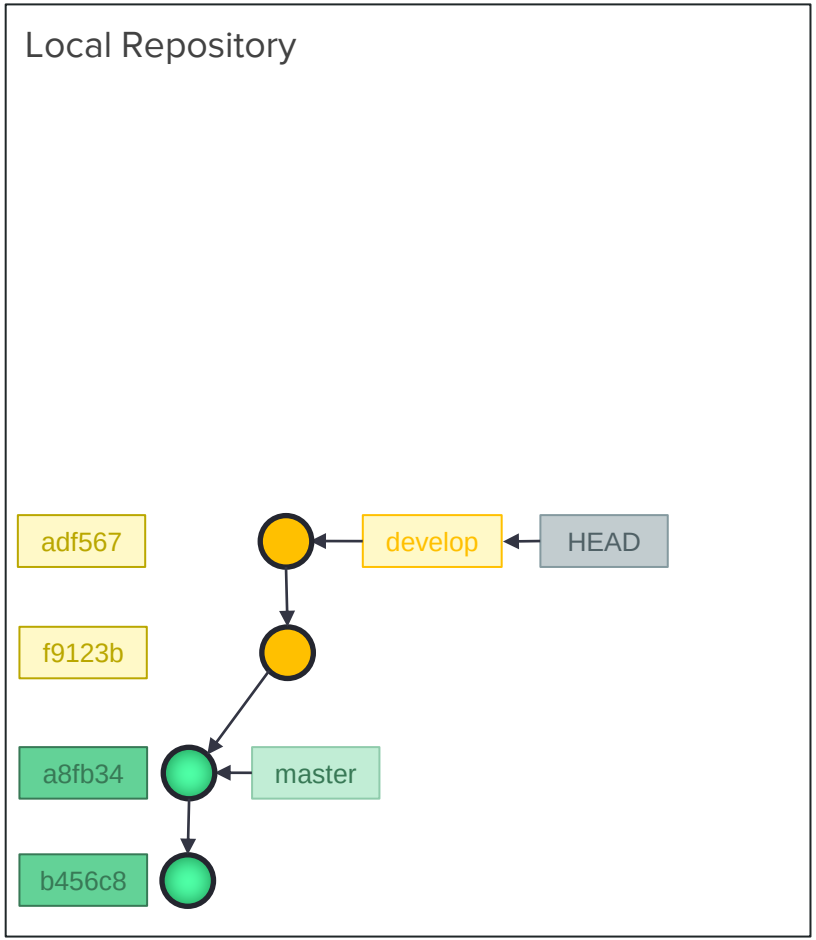
Branching

```
$ git commit

$ git branch develop

$ git checkout develop

$ git commit

$ |
```



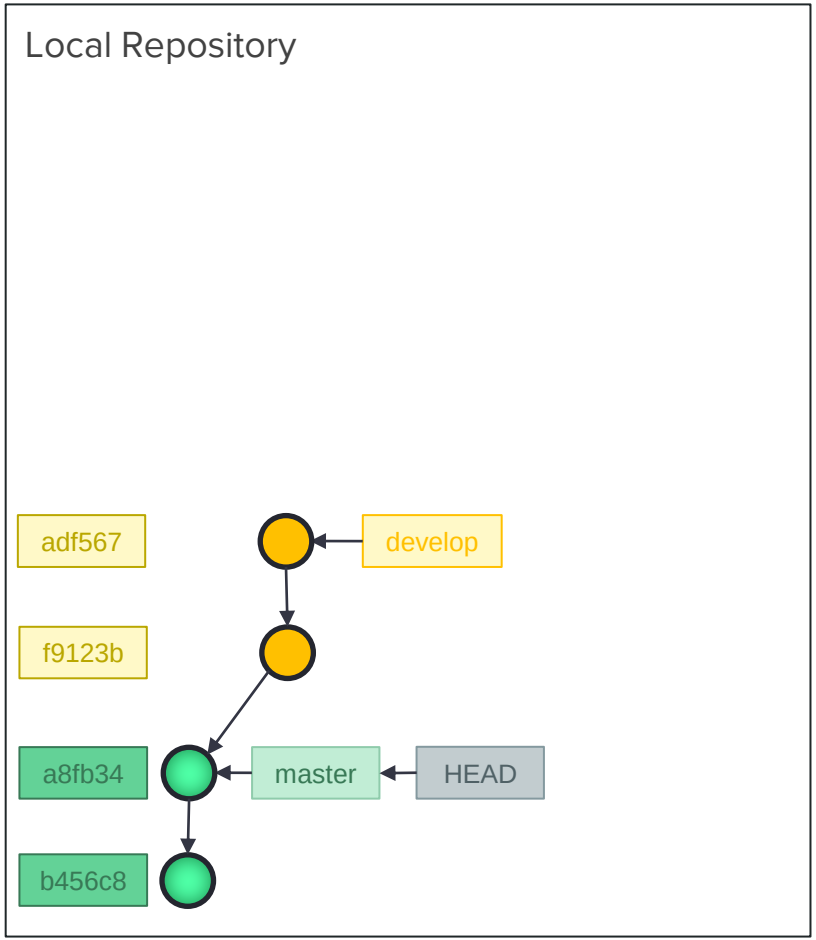
Branching

```
$ git branch develop

$ git checkout develop

$ git commit

$ git commit

$ |
```



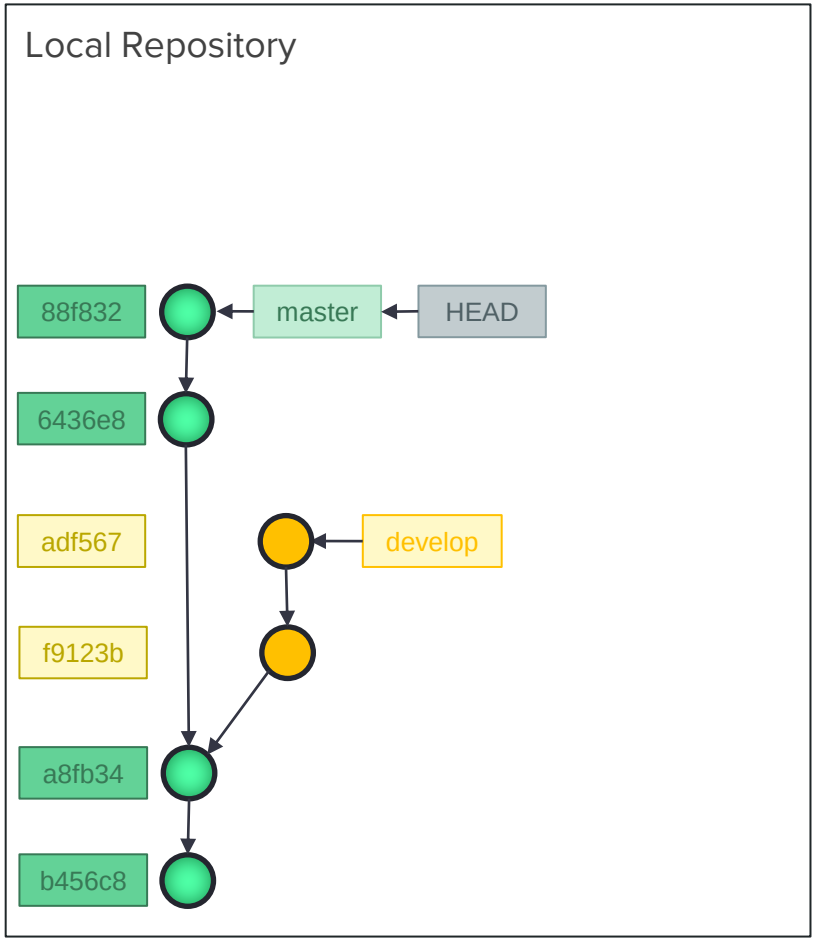
Branching

```
$ git checkout develop

$ git commit

$ git commit

$ git checkout master

$ |
```



Branching

```
$ git commit

$ git commit

$ git checkout master

$ git pull

$ |
```

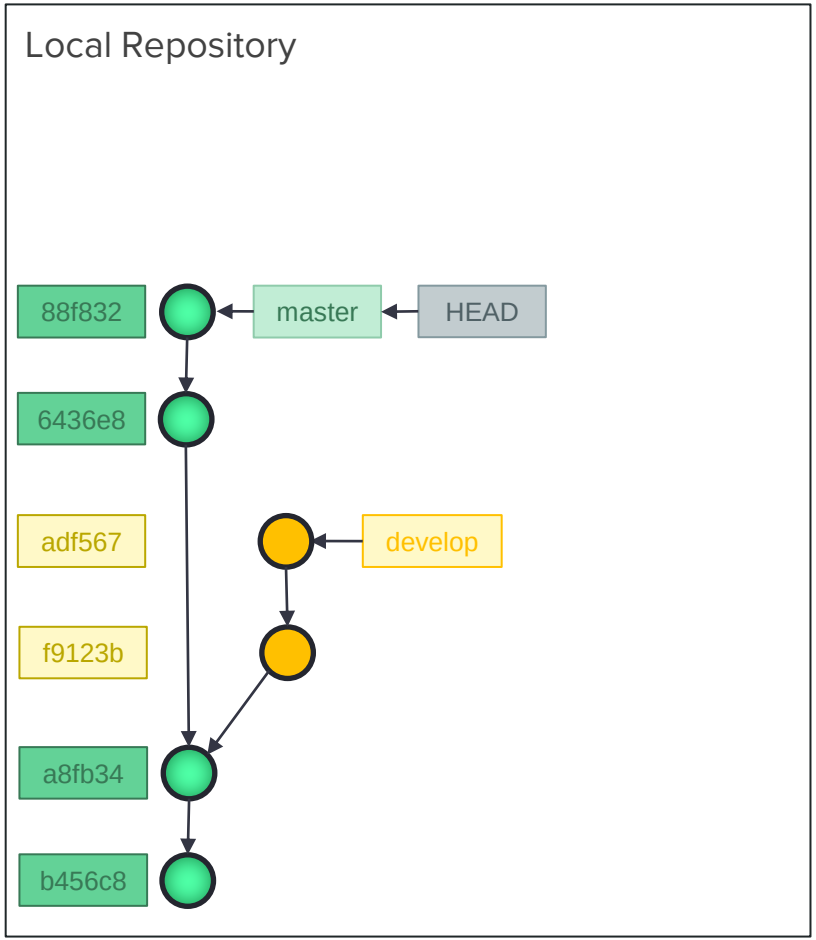


Git Basics Recap

Merging

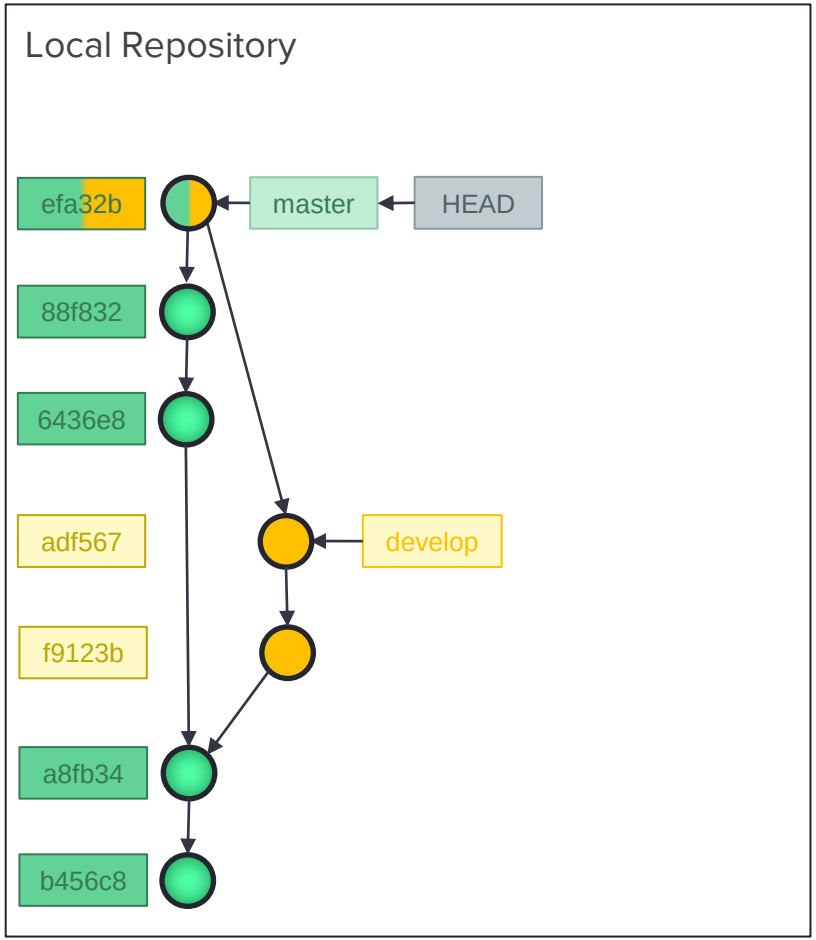
Merging

```
$ git commit

$ git checkout master

$ git pull

$ git merge develop

$ |
```



Merging

```
$ git commit

$ git checkout master

$ git pull

$ git merge develop

$ |
```

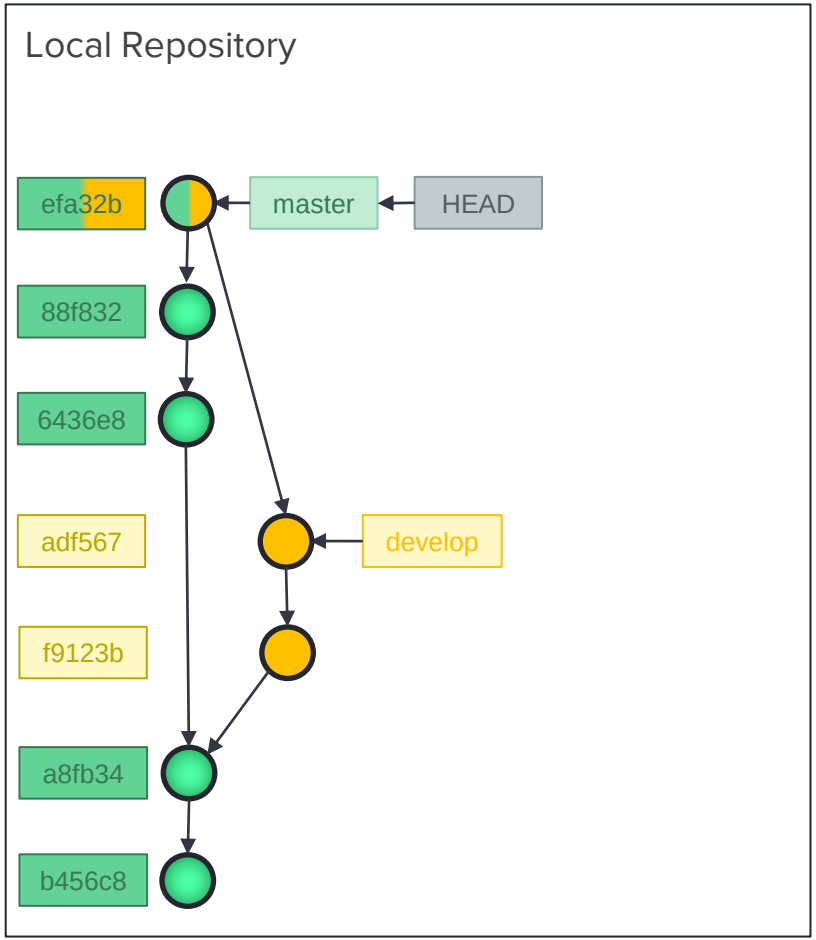


Merging

- In the case where the branches **have diverged** a **merge commit** is created
- A merge commit is a commit with two parent commits and incorporates changes from both parents
- Merging can cause conflicts which have to be solved **only once** in order for the merge to finish

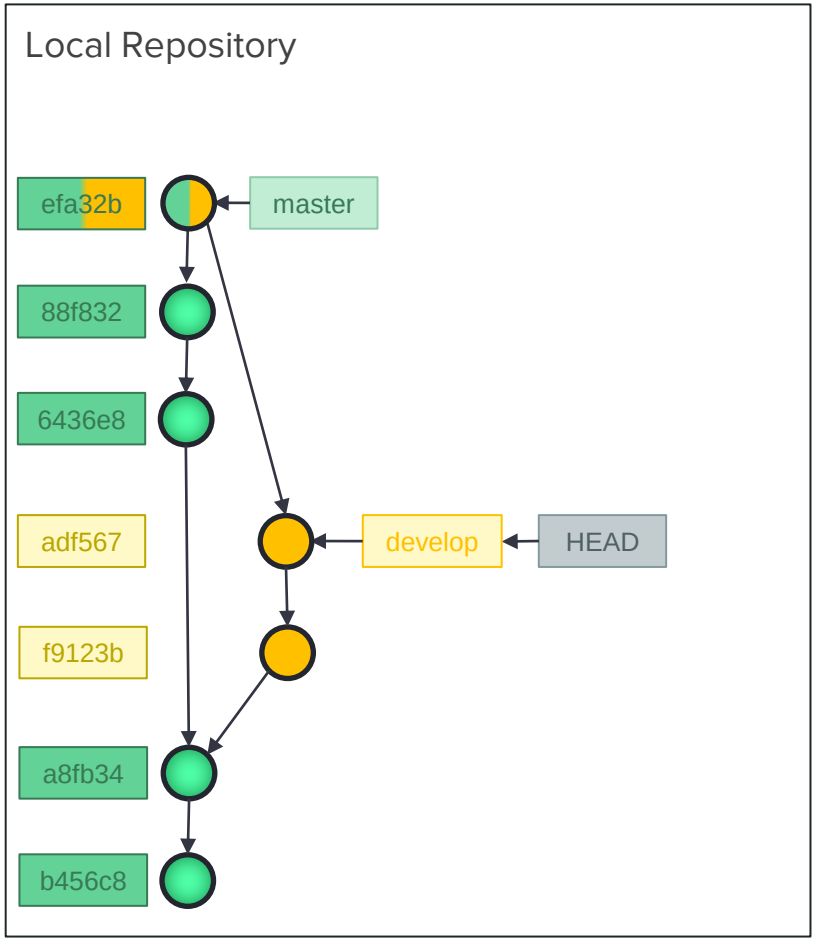
Merging

```
$ git commit

$ git checkout master

$ git pull

$ git merge develop

$ |
```



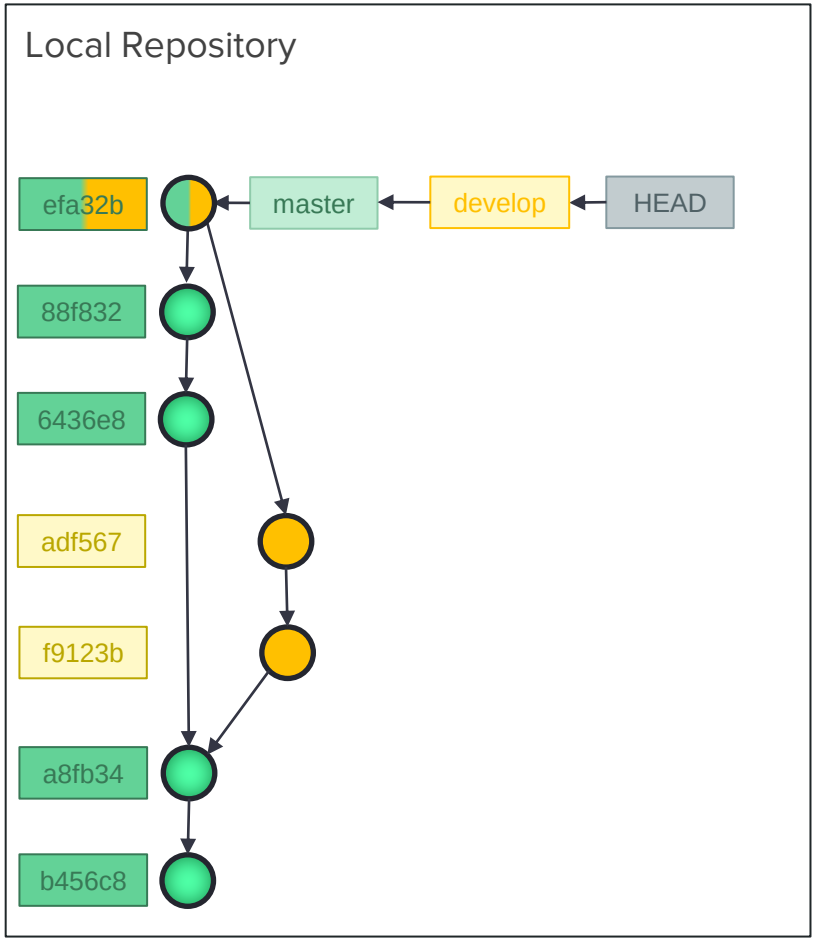
Merging

```
$ git checkout master

$ git pull

$ git merge develop

$ git checkout develop

$ |
```



Merging

```
$ git pull

$ git merge develop

$ git checkout develop

$ git merge master

$ |
```

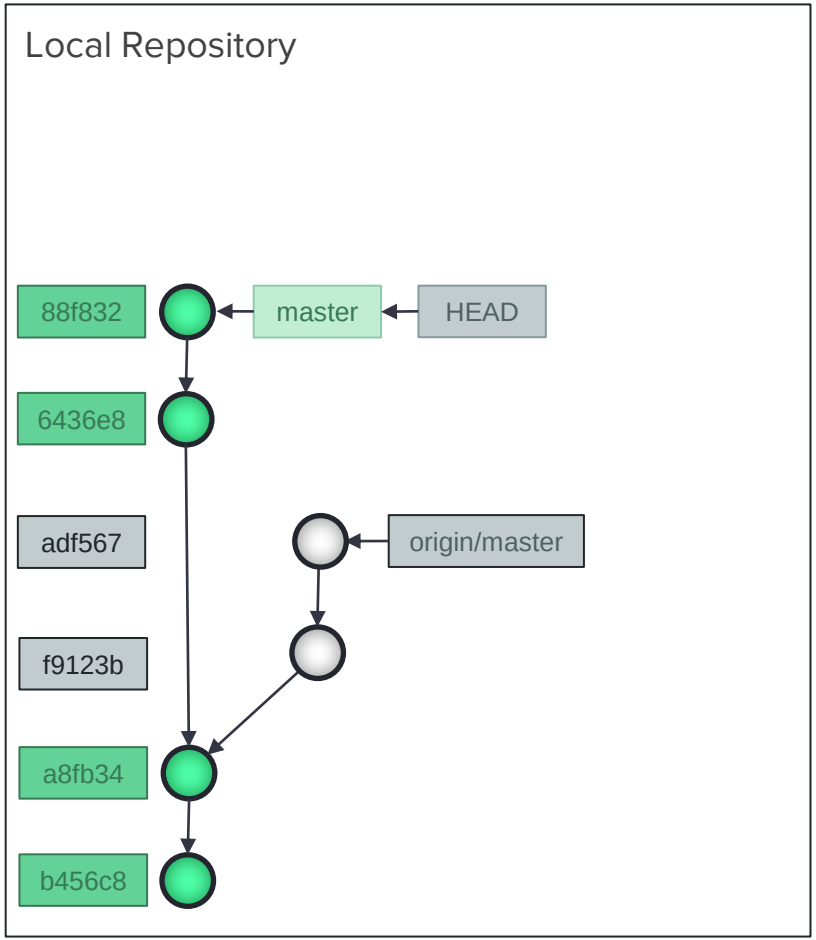


The Fast Forward

- In the case where we **CAN** connect to the latest commit on the other branch by going straight back along the line of commits a **Fast Forward** is possible
- A fast forward just moves the curent branch to the tip of the other branch, no commits are created
- A fast forward cannot end in a conflict by definition
- You can force the creation of a merge commit if you want to even if fast forwarding is possible (`git merge develop --no-ff`)
- You can constrain merges to only happen if fast forwarding is possible (--ff-only)

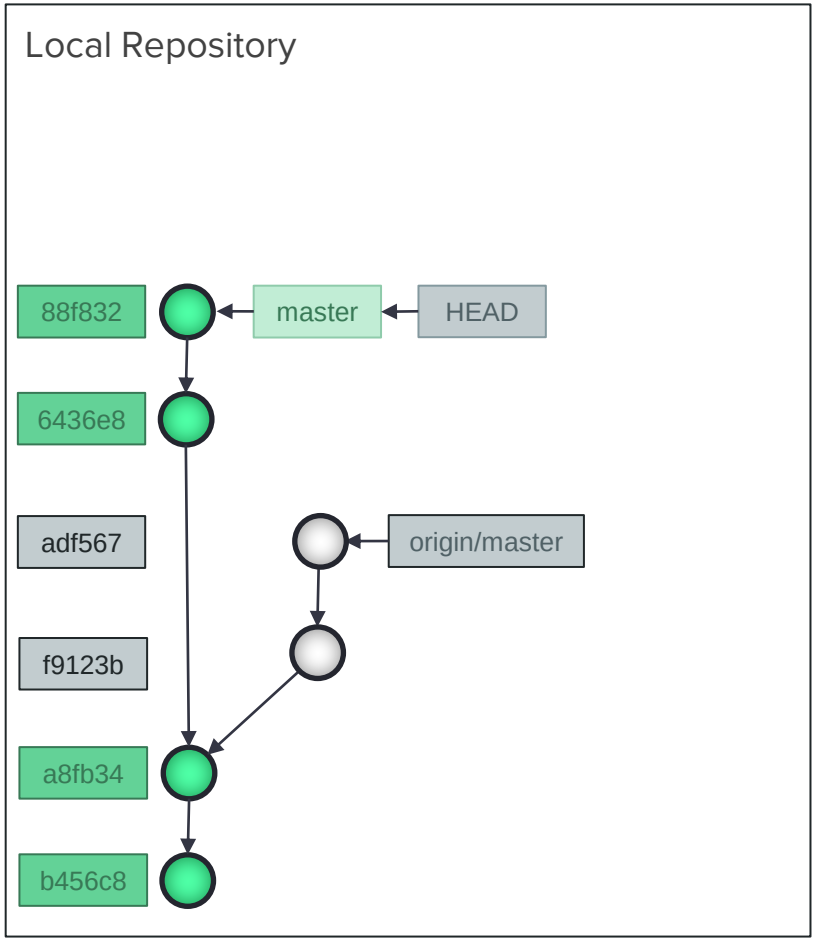
Git Pull

```
$ |
```


Local Repository
88f832
master
HEAD
6436e8
adf567
origin/master
f9123b
a8fb34
b456c8

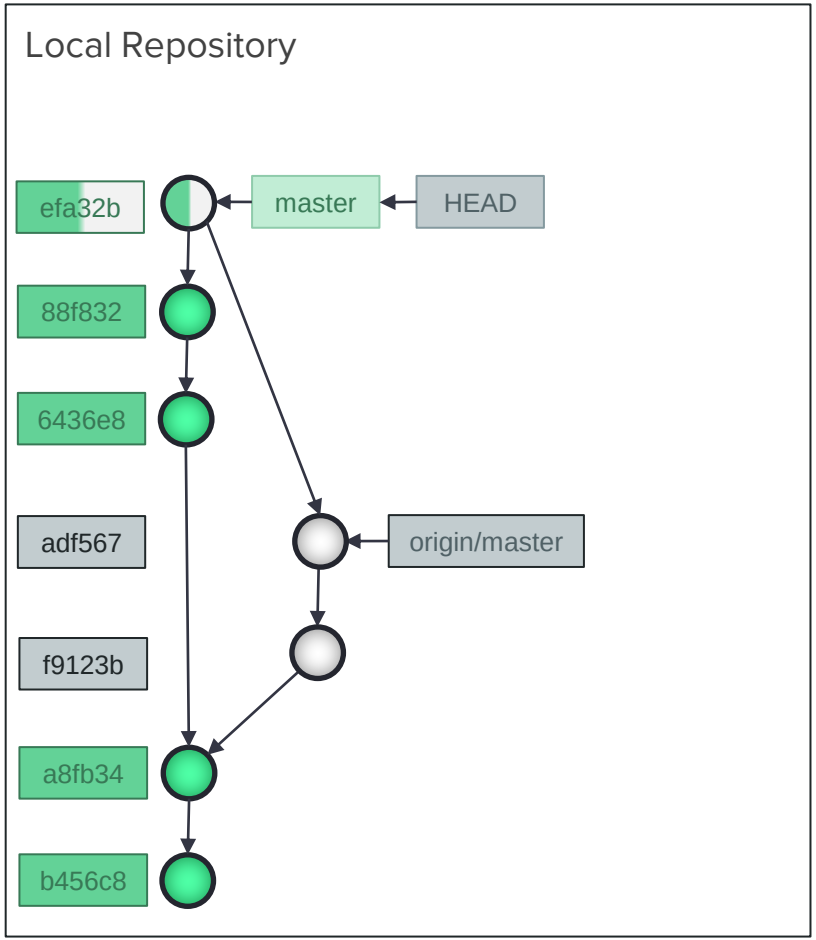
Git Pull

```
$ git pull

$ |
```



Git Pull

```
$ git pull

$ |
```



History Manipulation

History Manipulation

The practice of changing the git history

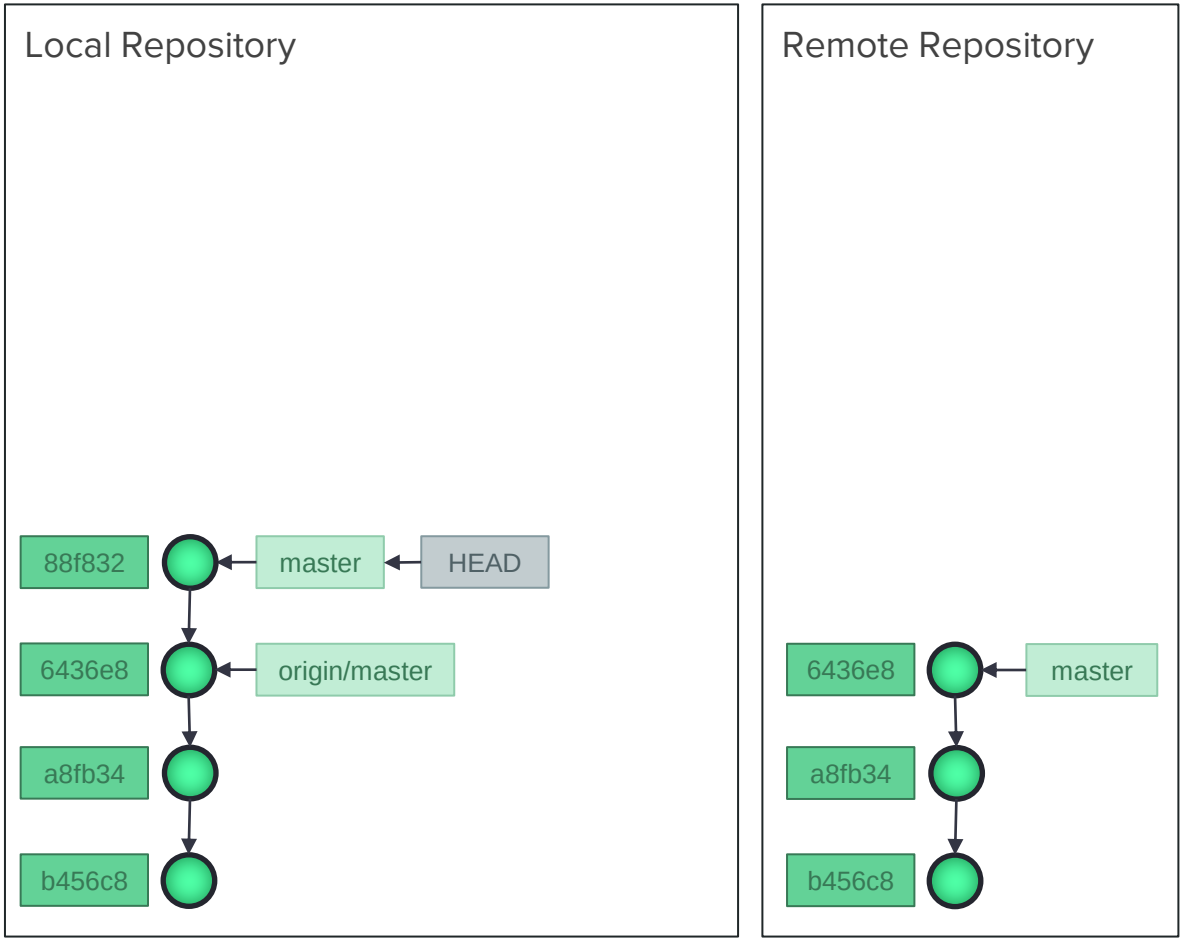
- Essentially: actions which cause commits to disappear, change their hash etc. in such a way that the original history of actions cannot be seen in the git commit log.

History Manipulation

Resetting

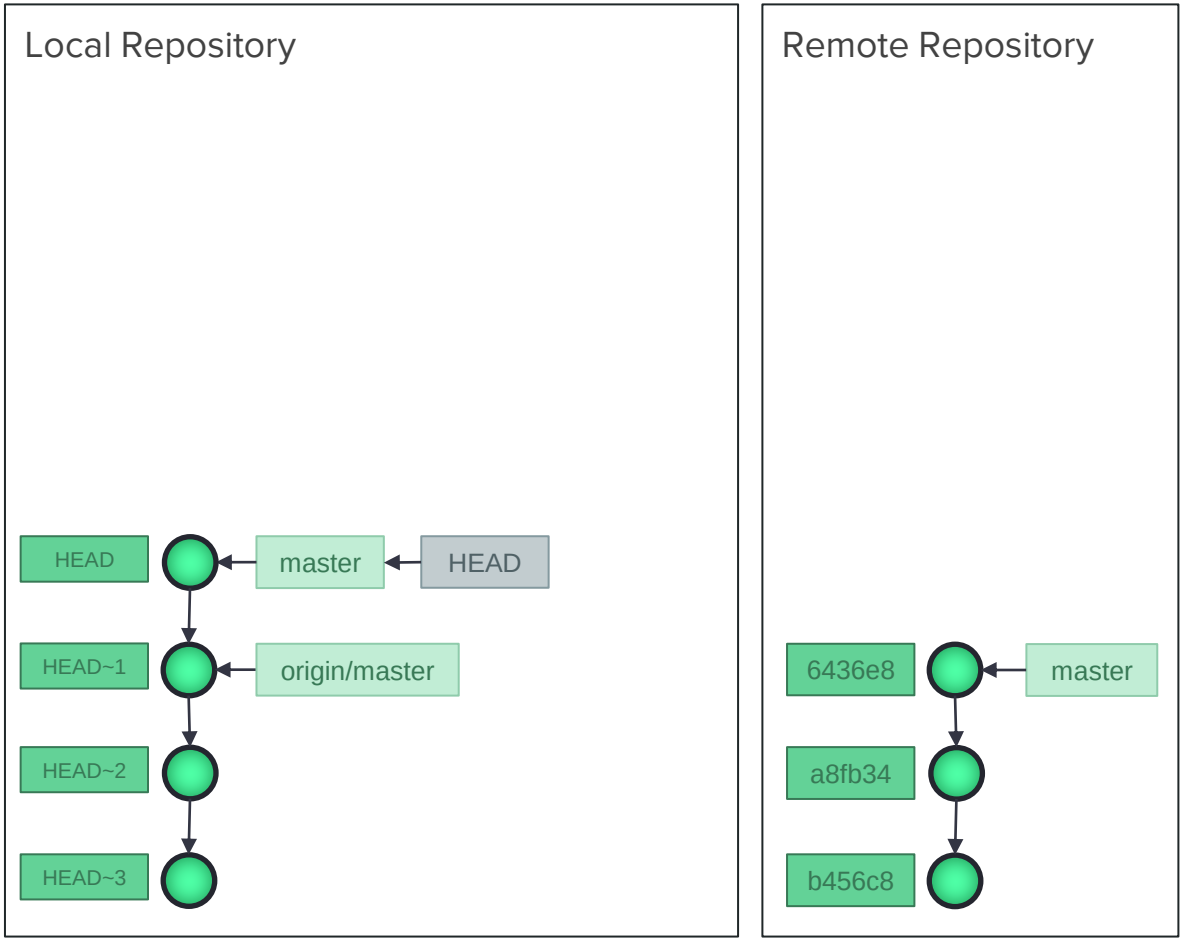
Resetting

```
$ |
```


Local Repository
88f832
master
HEAD
6436e8
origin/master
a8fb34
b456c8
Remote Repository
6436e8
master
a8fb34
b456c8

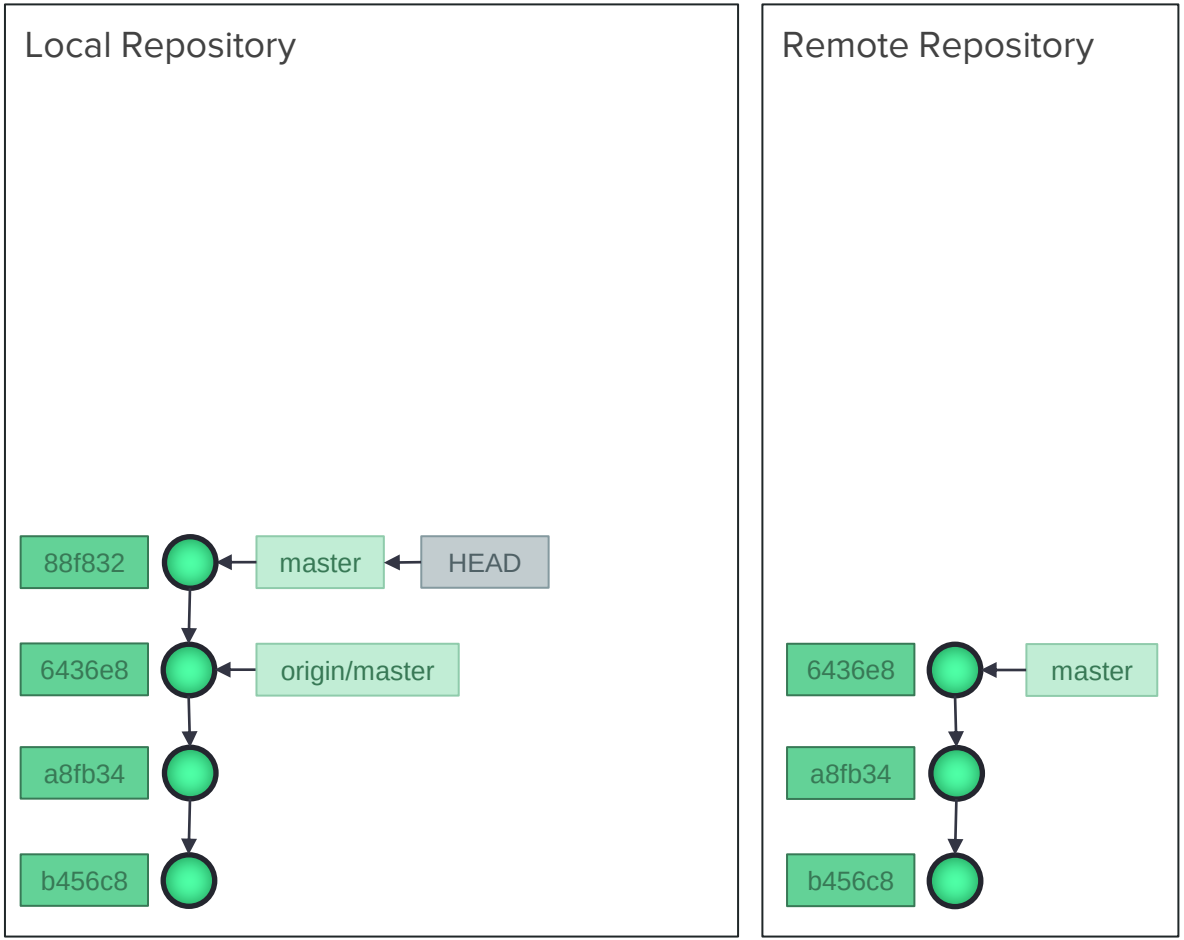
Resetting

```
$ |
```


Local Repository
HEAD
master
HEAD
HEAD~1
origin/master
HEAD~2
HEAD~3
Remote Repository
6436e8
master
a8fb34
b456c8

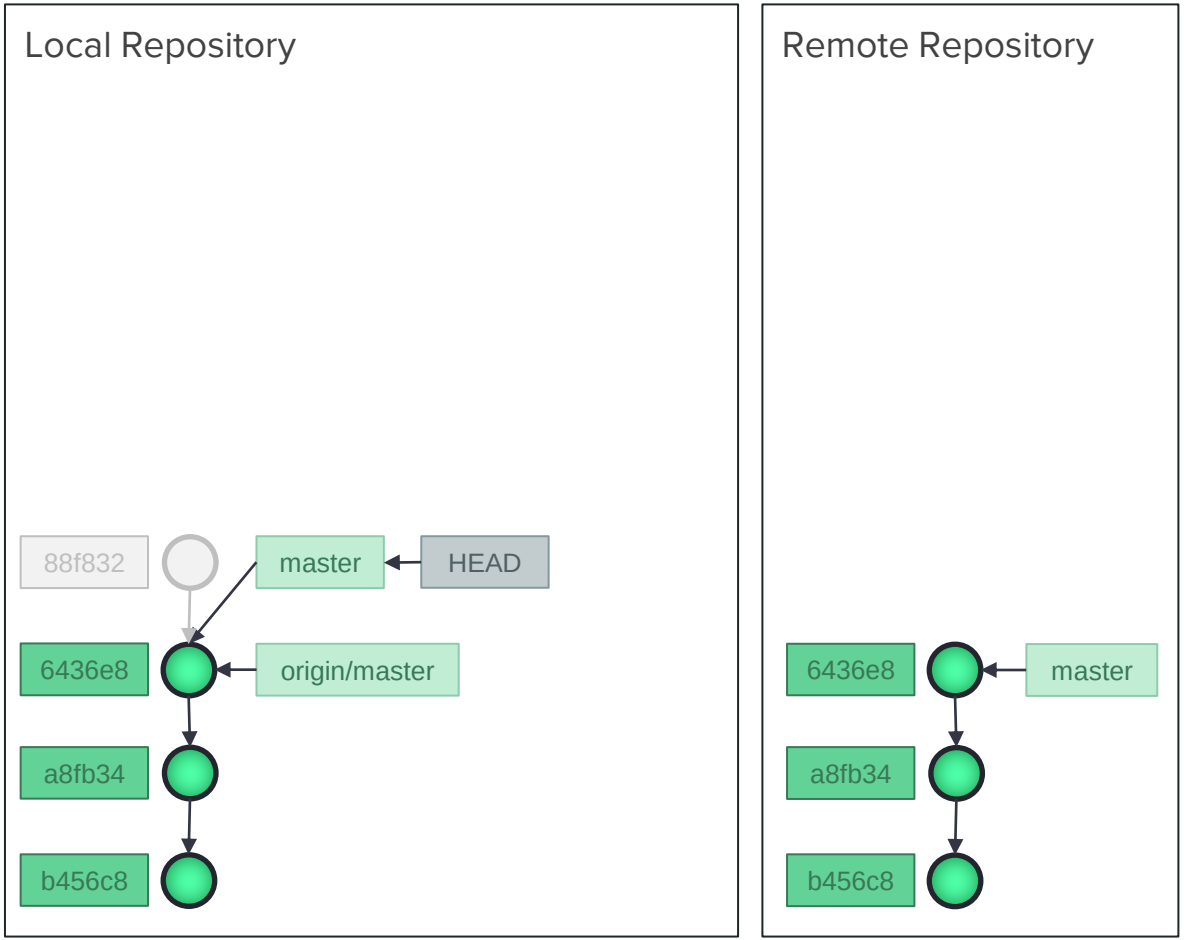
Resetting

```
$ git reset HEAD~1

$ |
```



Resetting

```
$ git reset HEAD~1

$ |
```



Hard vs. Soft

```
$ git reset --soft

$ |
```

Remote Repository

Local Repository


Readme.md v2

Staging Area

Working Directory


Readme.md v2

Hard vs. Soft

```
$ git reset --soft

$ |
```

Remote Repository

Local Repository

Readme.md v1

Staging Area

Readme.md v2

Working Directory

Readme.md v2

Hard vs. Soft

```
$ git reset --hard

$ |
```

Remote Repository

Local Repository

Readme.md v2

Staging Area

Working Directory

Readme.md v2

Hard vs. Soft

```
$ git reset --hard

$ |
```

Remote Repository

Local Repository

Readme.md v1

Staging Area

Working Directory

Readme.md v1

Mixed (default)

```
$ git reset --mixed

$ |
```

Remote Repository

Local Repository

Readme.md v2

Staging Area

Working Directory

Readme.md v2

Mixed (default)

```
$ git reset --mixed

$ |
```

Remote Repository

Local Repository

Readme.md v1

Staging Area

Working Directory

Readme.md v2

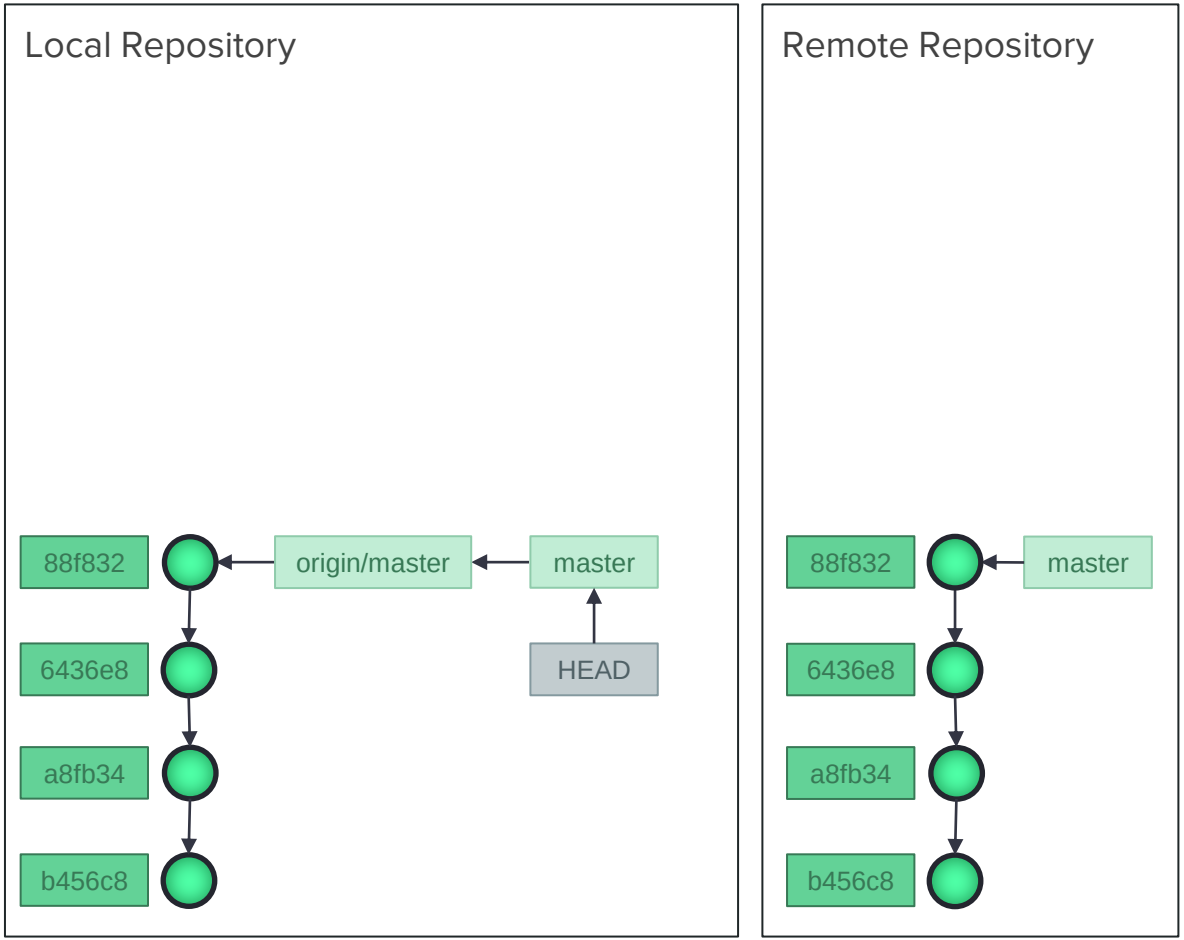
Do’s and Dont’s

- If you **didn’t push** your changes to the remote repo you can reset **soft** and **mixed**
  - You can also reset **hard** but be aware you are going to **lose** ALL changes irreversibly (might as well be irreversible)

- If you **are not the only one** working on the branch **DON’T RESET**!
  - e.g. the changes are already pushed
  - e.g. there are other branches created out of this branch

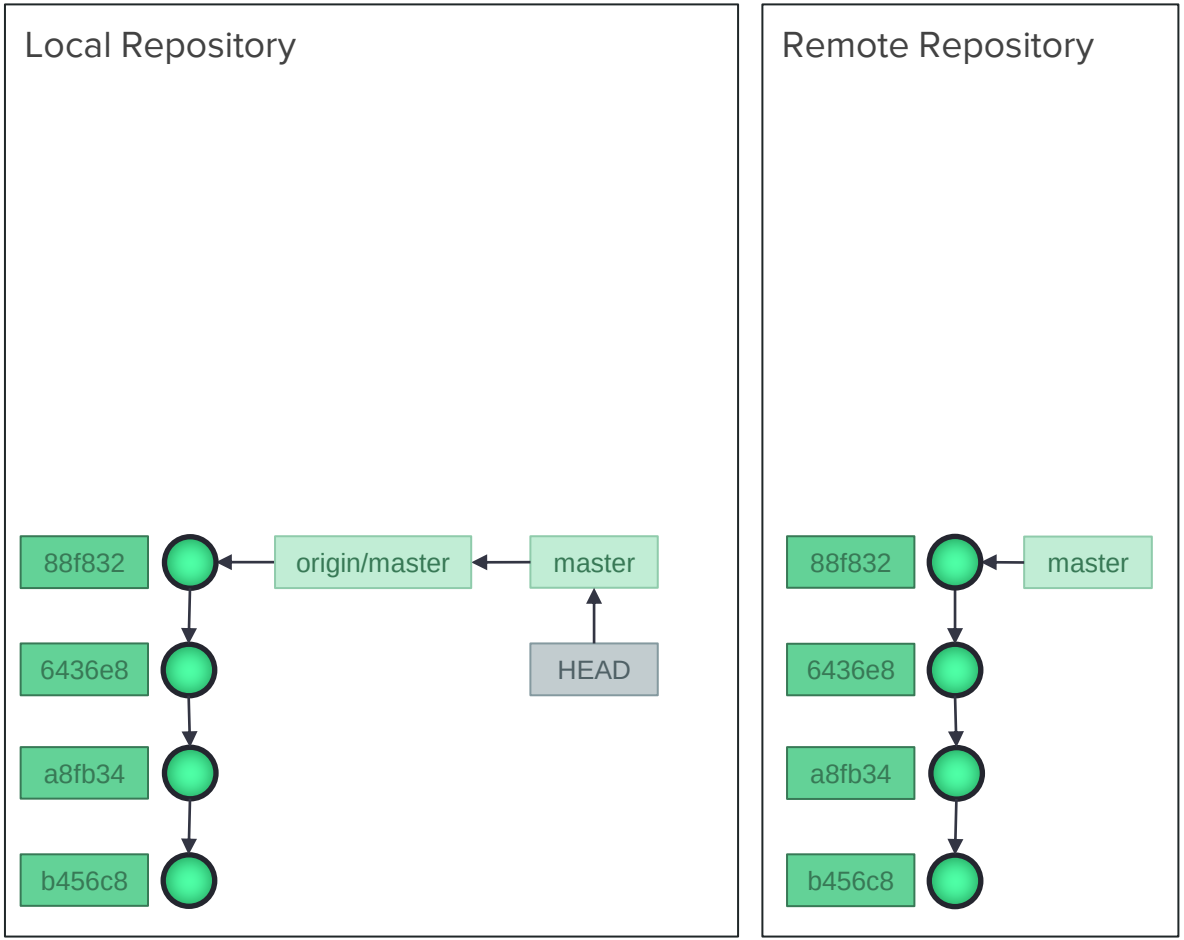
But Why?

```
$ |
```


Local Repository
88f832
origin/master
master
6436e8
HEAD
a8fb34
b456c8
Remote Repository
88f832
master
6436e8
a8fb34
b456c8

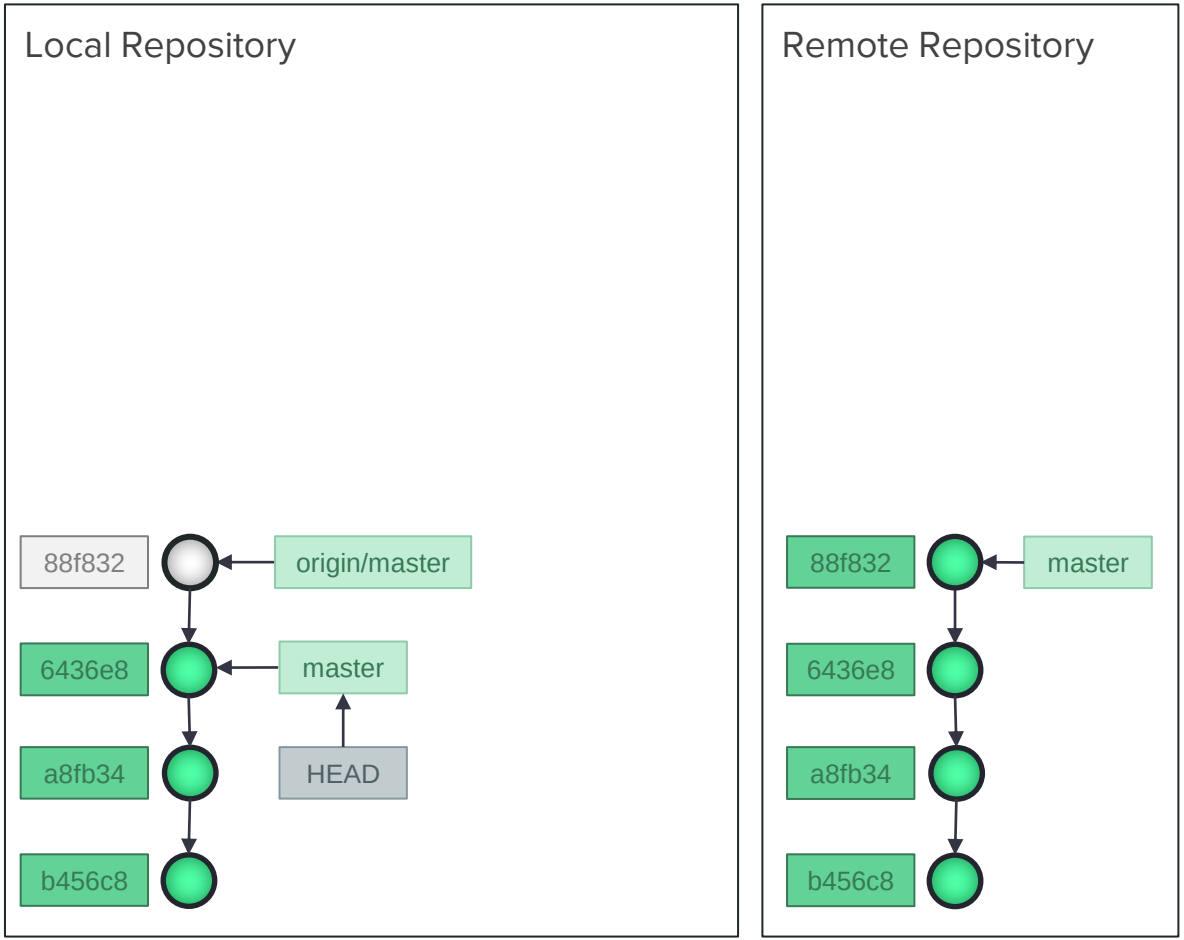
But Why?

```
$ git reset

$ |
```



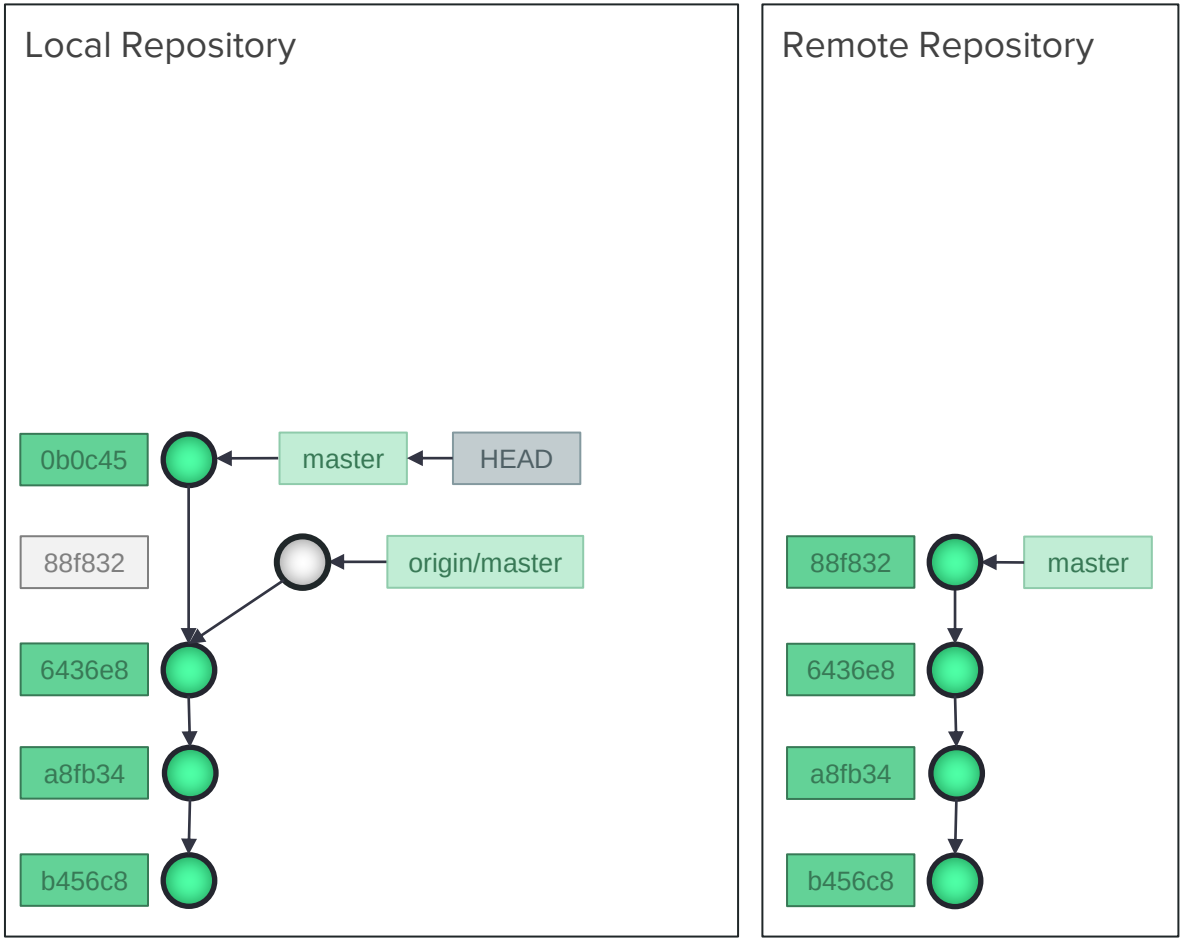
But Why?

```
$ git reset

$ |
```



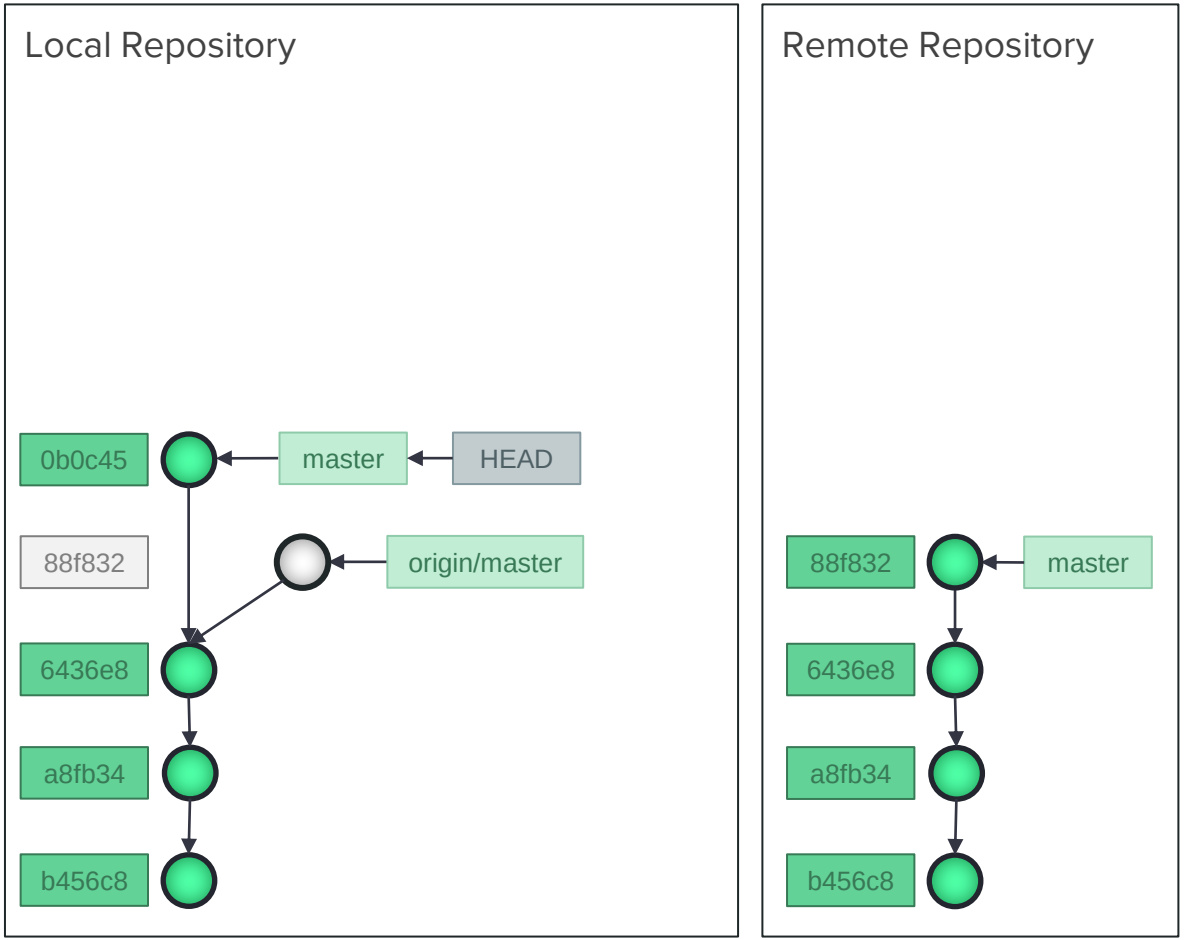
But Why?

```
$ git reset

$ git commit

$ |
```



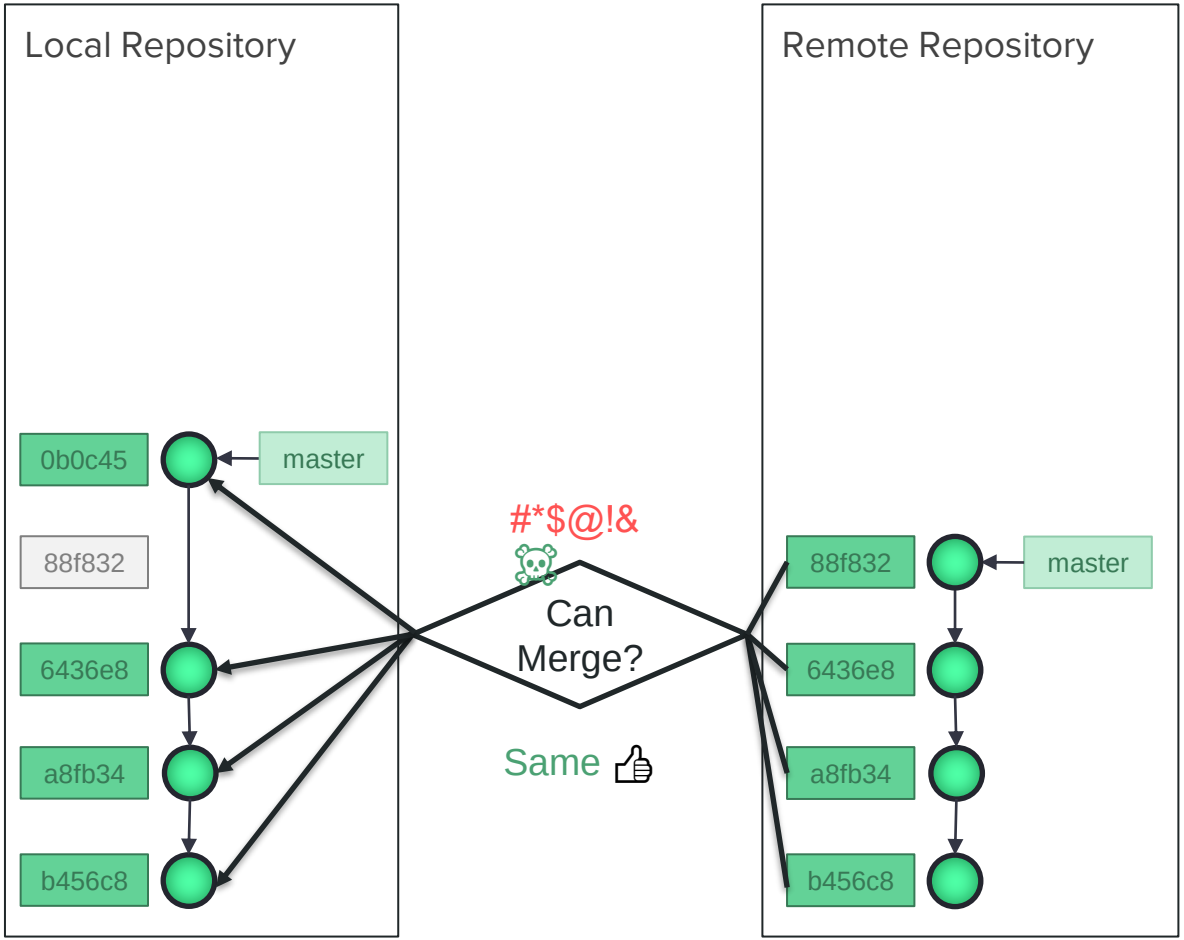
But Why?

```
$ git reset

$ git commit

$ git push

$ |
```



But Why?

```
$ git reset

$ git commit

$ git push

$ |
```

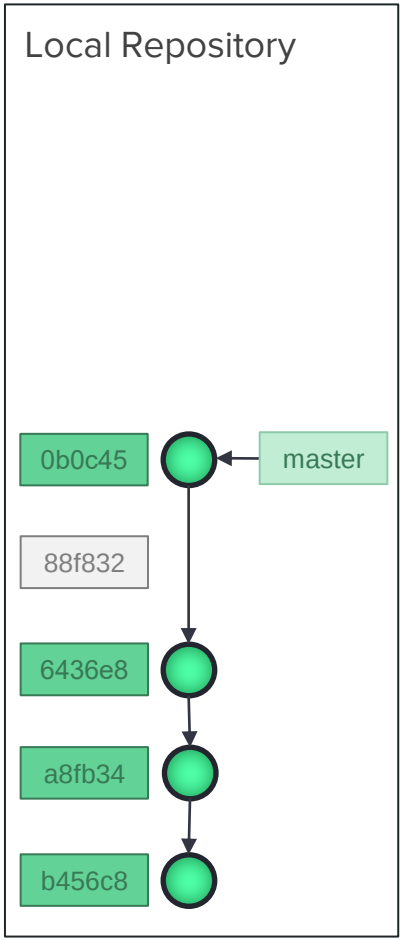


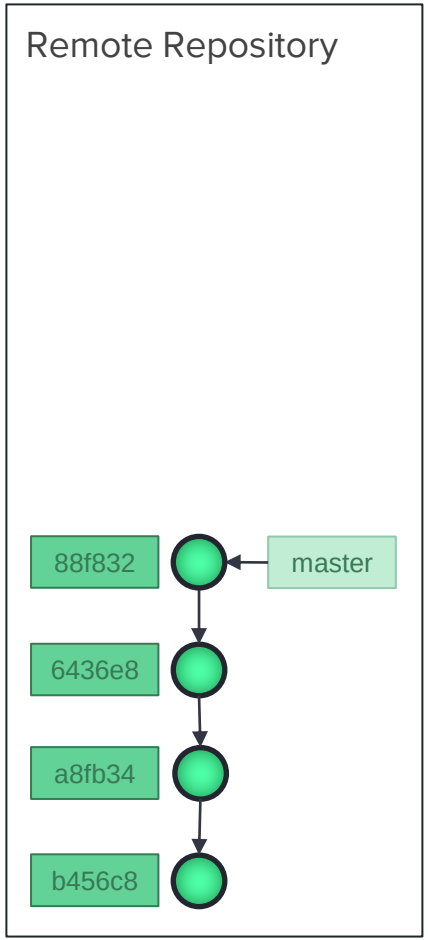
But Why?

```
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:PaulNegoescu/gittest.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.
```

But Why?

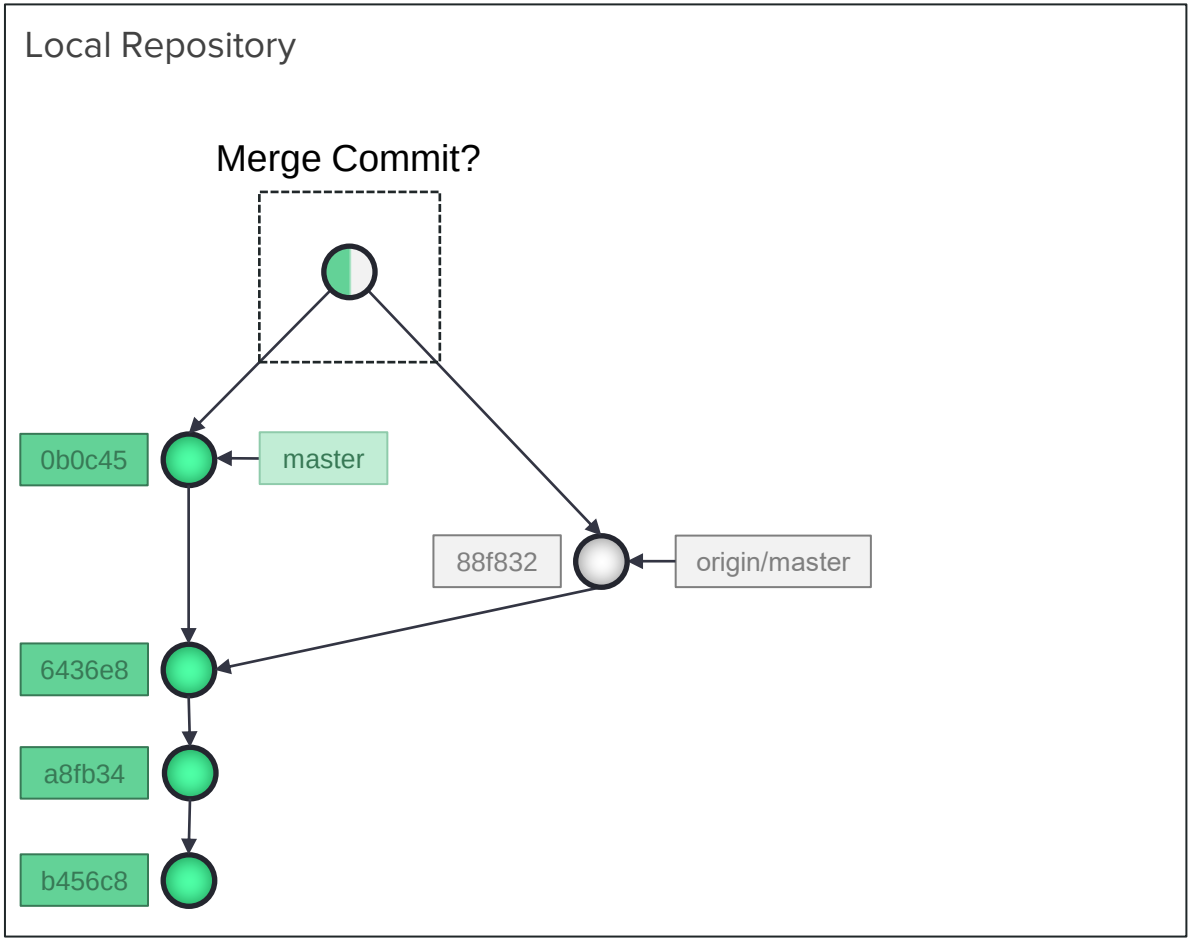
```
$ git reset

$ git commit

$ git push

$ git pull

$ |
```





But Why?

```
$ git reset

$ git commit

$ git push

$ git pull

$ |
```



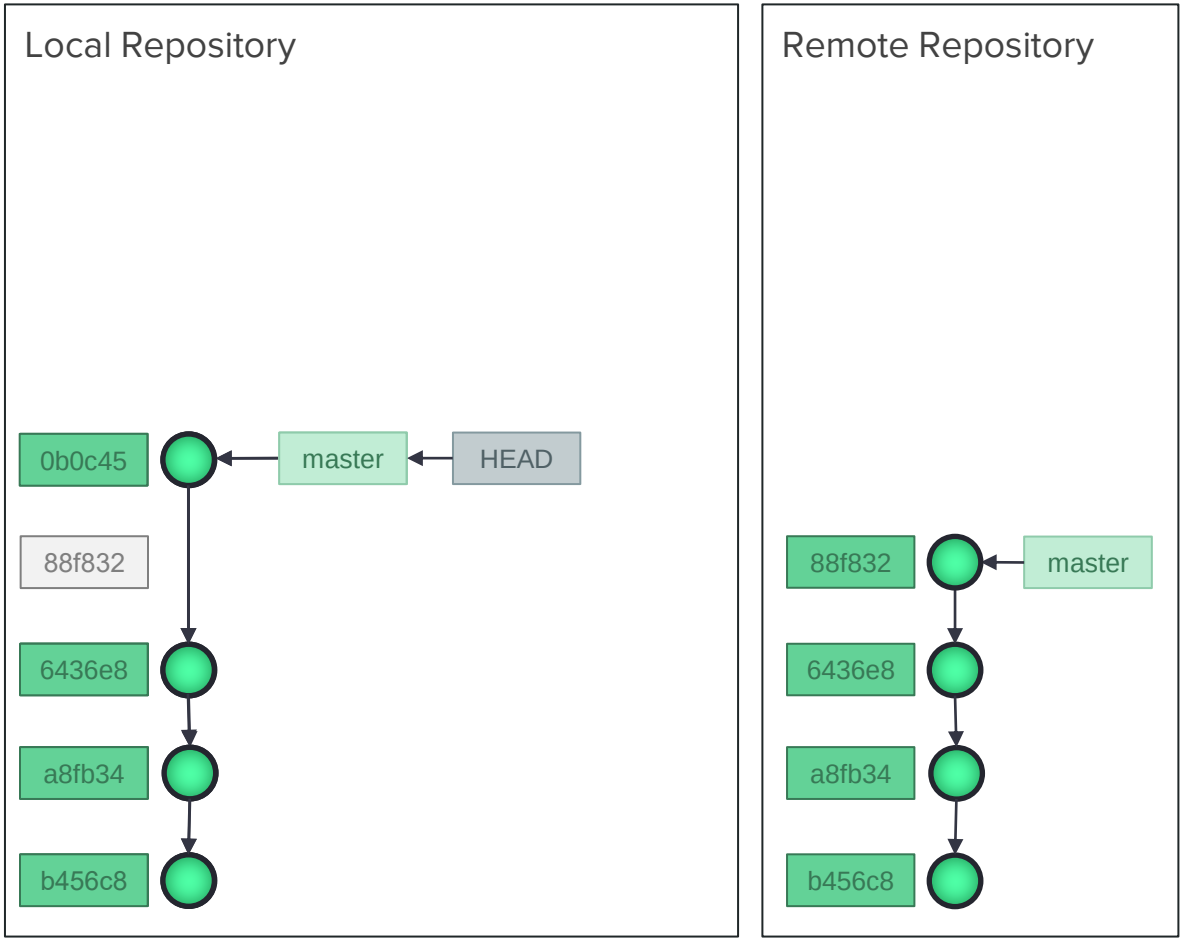
Nope!

- You either get:
  - A merge conflict (in case of soft or mixed reset and minor changes to the affected files)
  - Or an automatic merge (in case of hard reset and different changes)

This brings the changes you reset back into the repo!

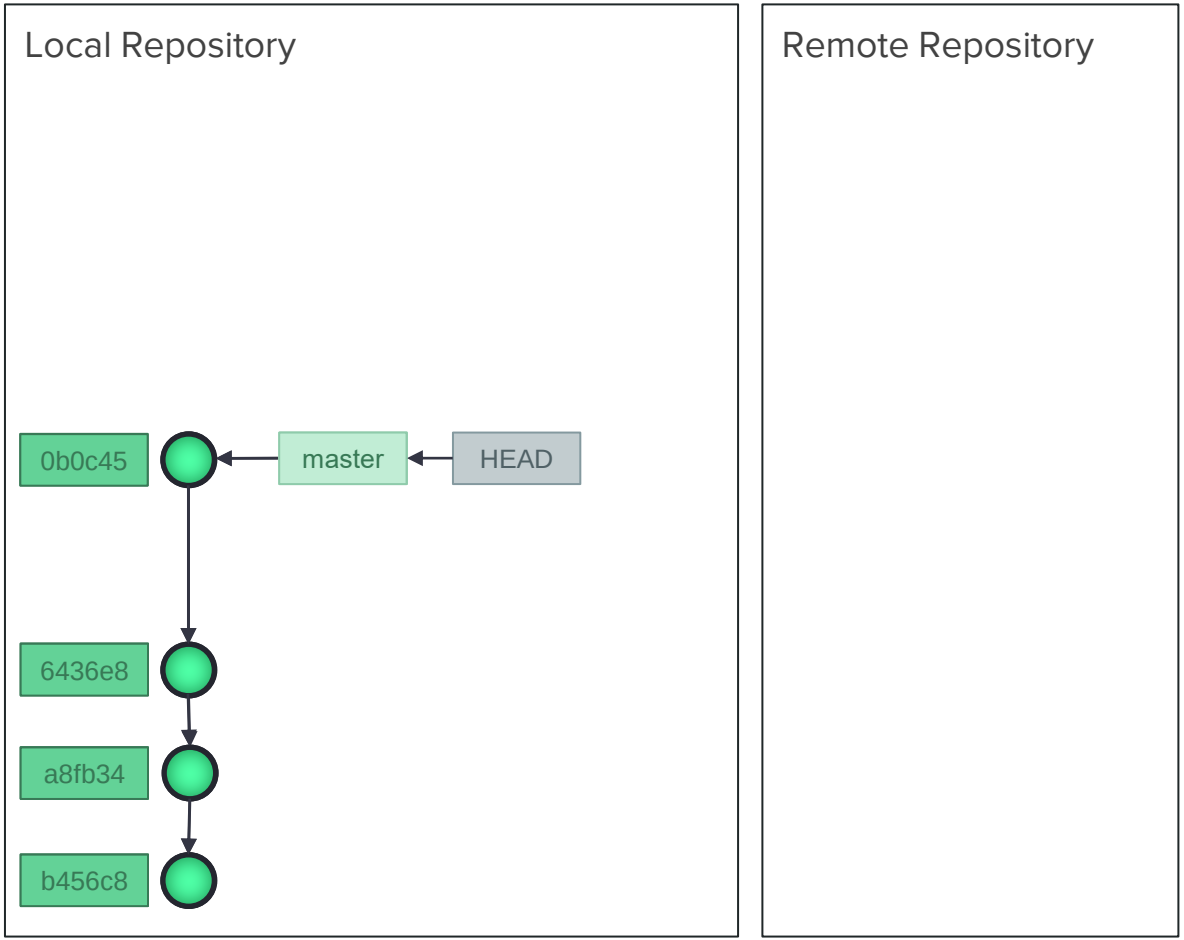
Worst Solution

```
$ git push --force

$ |
```



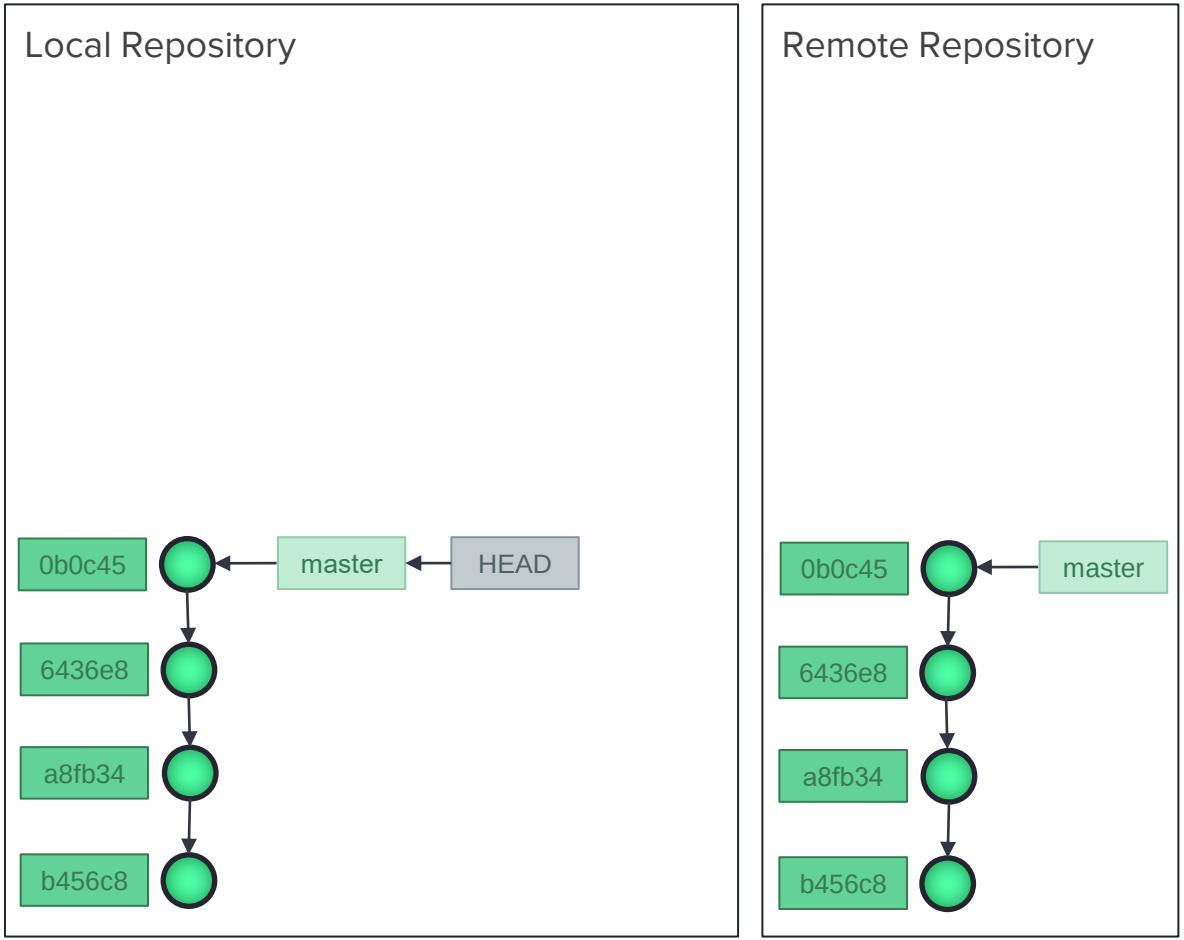
Worst Solution

```
$ git push --force

$ |
```



Worst Solution

```
$ git push --force

$ |
```



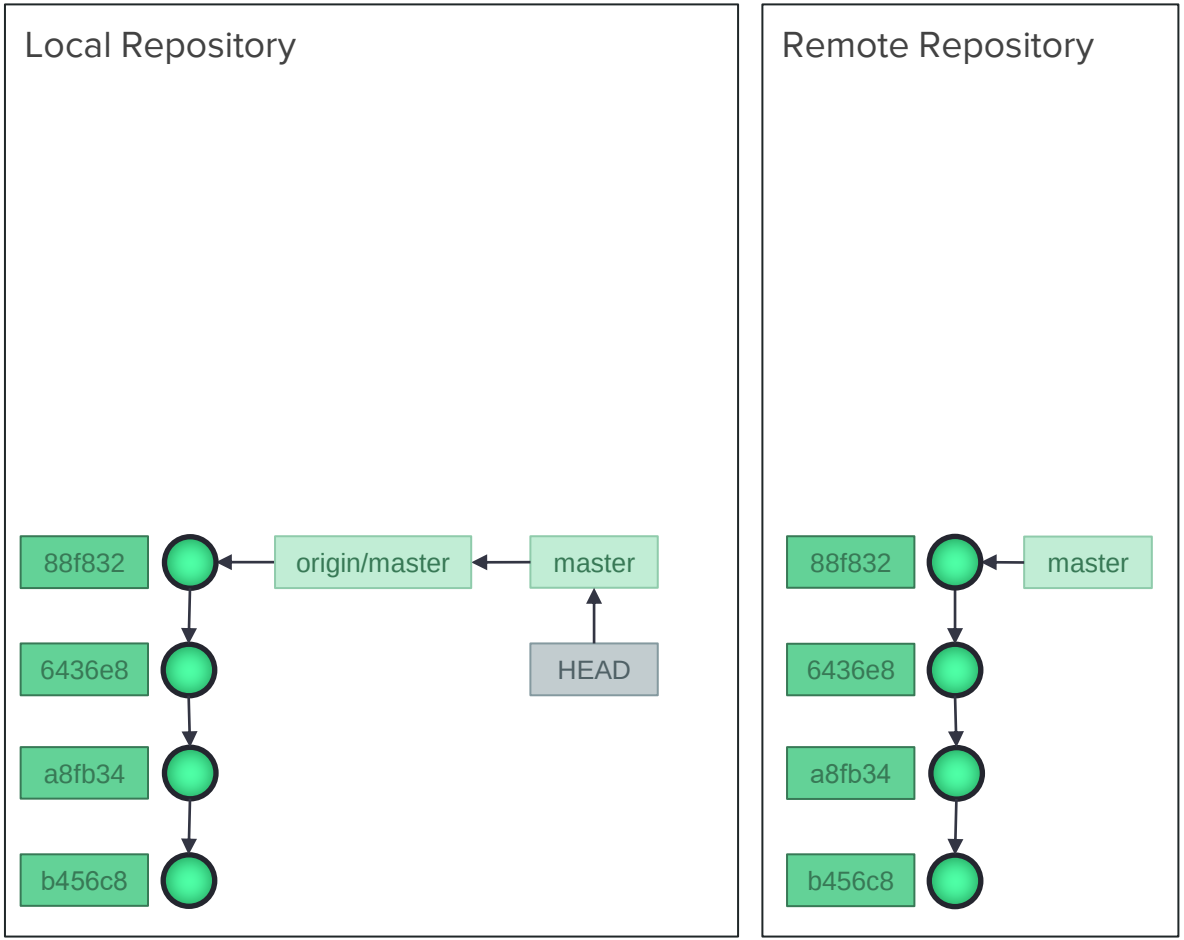
Developer: git push origin master --force
Developer: Sorry, wrong window
Every other developer in the chat channel:

Force Pushing

- Force pushing **overwrites** the remote history
- It changes the history for **every contributor**
- Everyone who now pulls from the remote will have confusing conflicts to fix, which, if done wrong, **can potentially reintroduce the eliminated changes** into the codebase

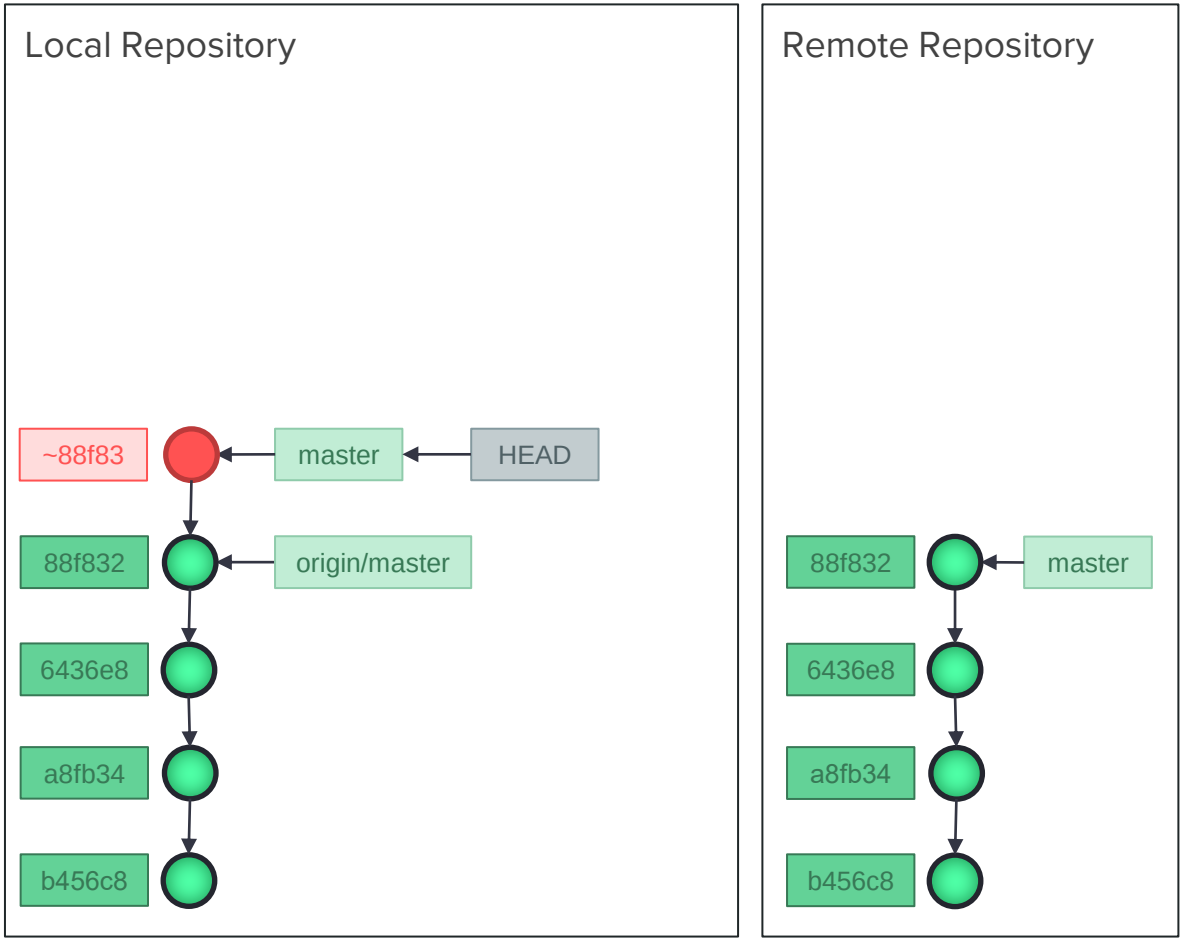
Best Solution

```
$ git revert 88f832

$ |
```



Best Solution

```
$ git revert 88f832

$ |
```



Reverting

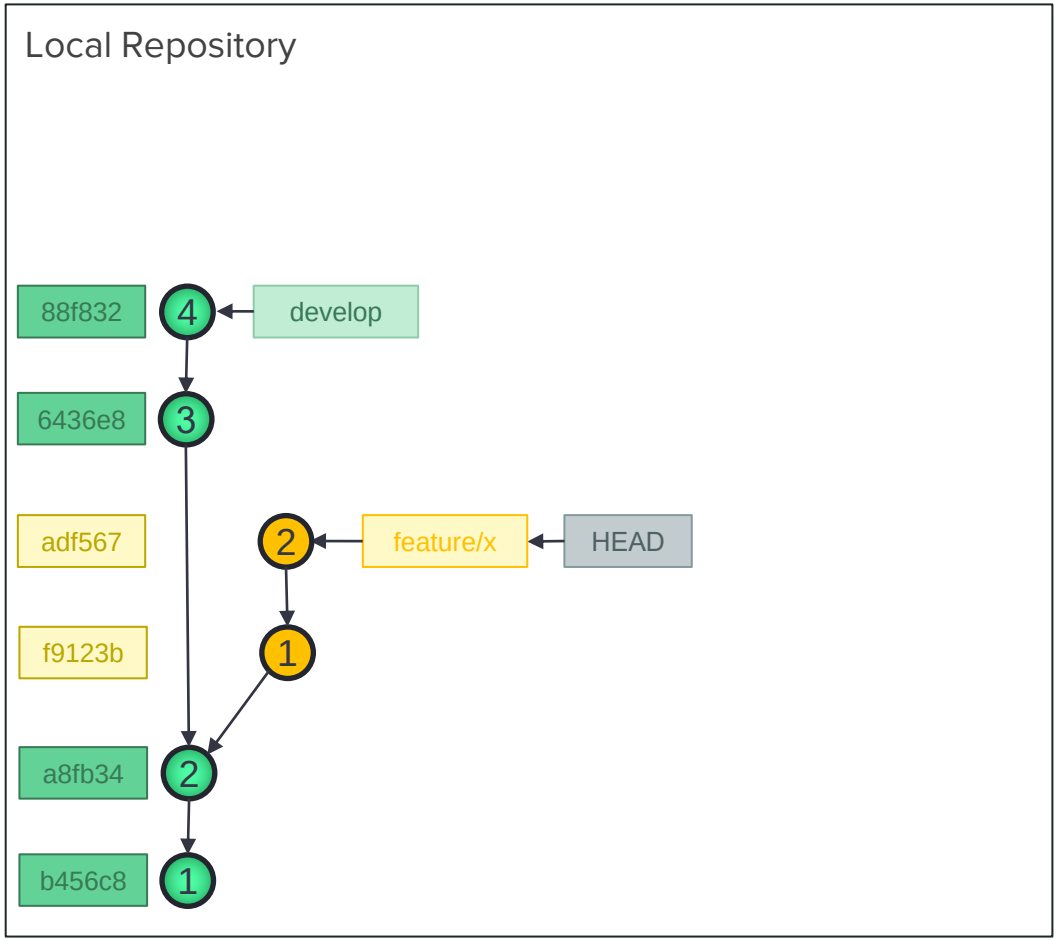
- Reverting creates a new commit that holds the exact **inverse of all changes** in the commit you revert
- Does not change the history
- Allows for problemless push
- Does not hurt your colleagues
- You can still access the discarded changes later if the need arises again

History Manipulation

Rebasing

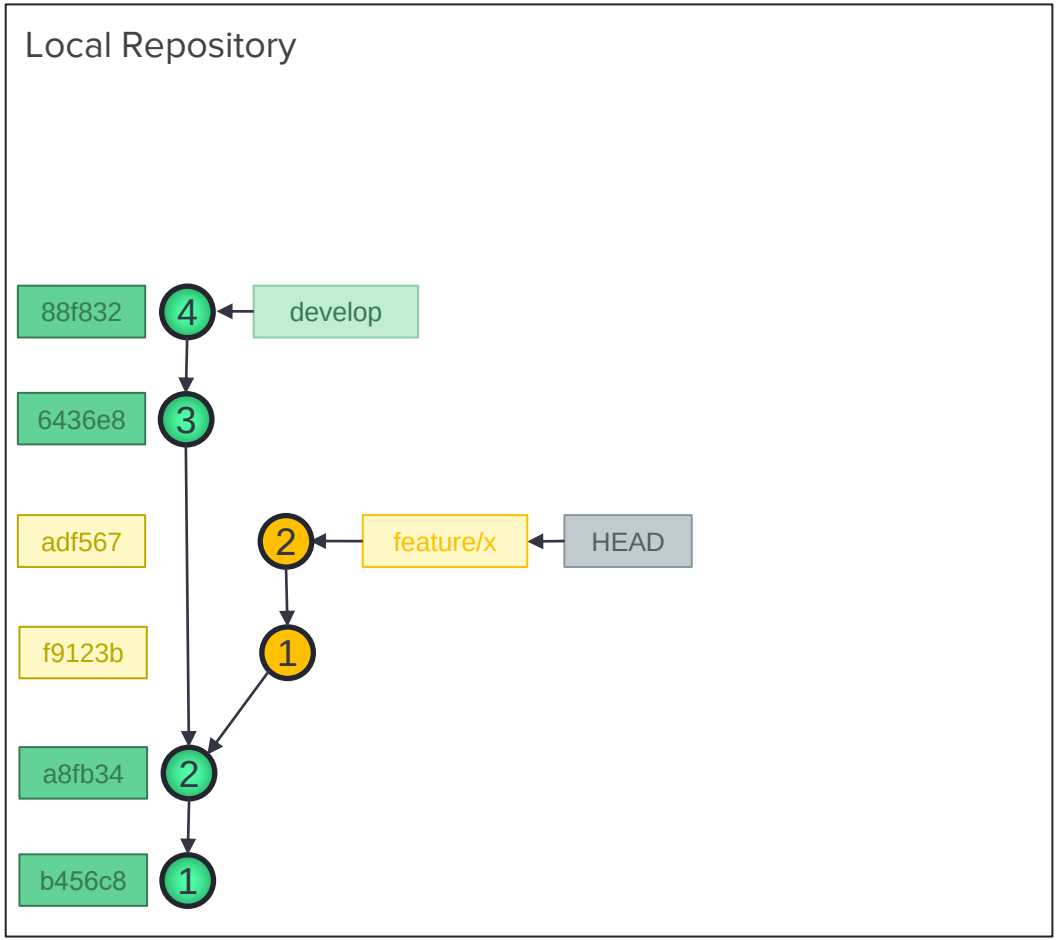
Rebasing

```
$ |
```


Local Repository
88f832
4
develop
6436e8
3
adf567
2
feature/x
HEAD
f9123b
1
a8fb34
2
b456c8
1

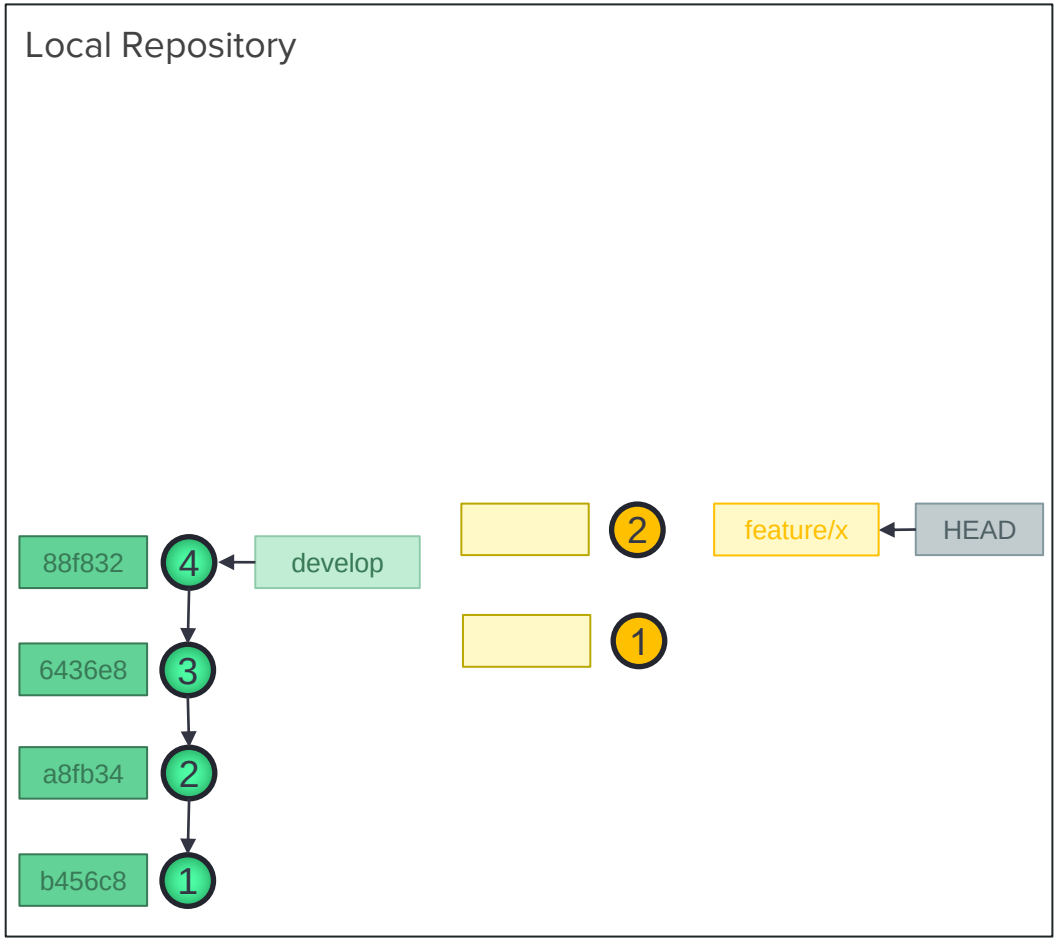
Rebasing

```
$ git rebase develop

$ |
```



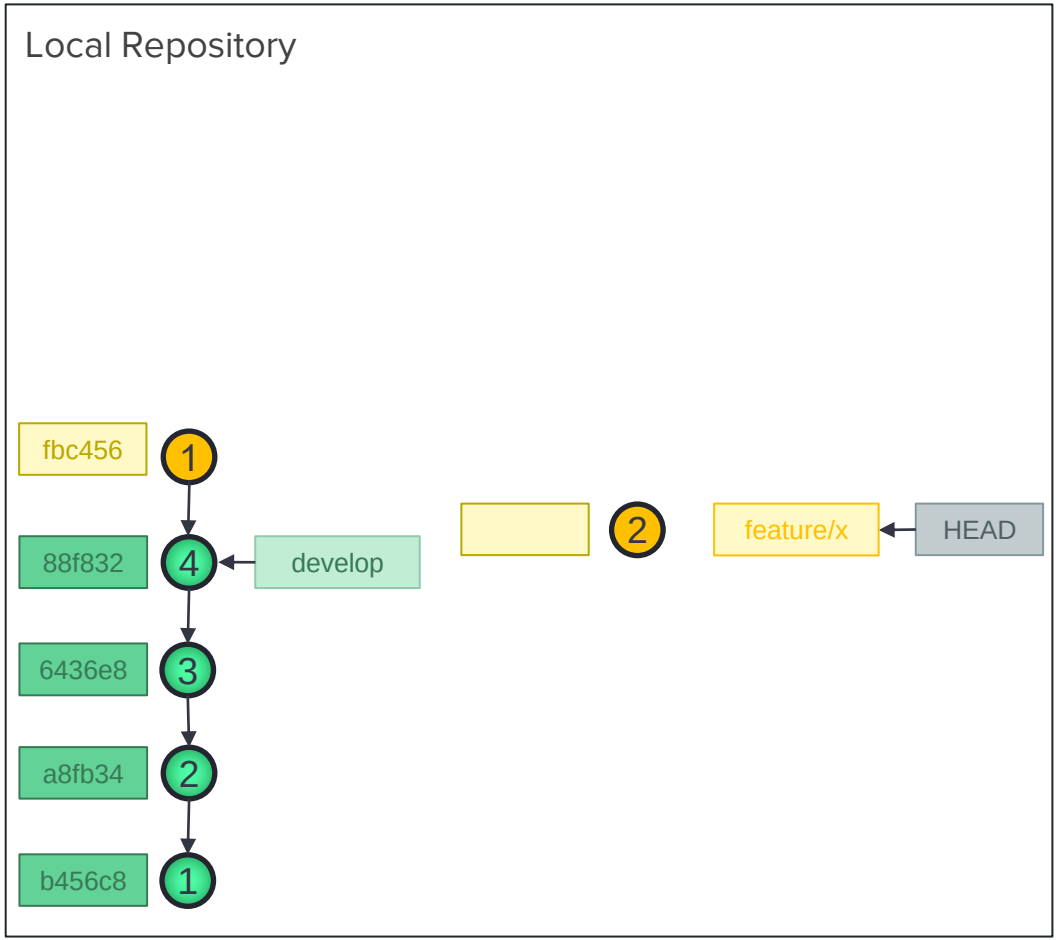
Rebasing

```
$ git rebase develop

$ |
```



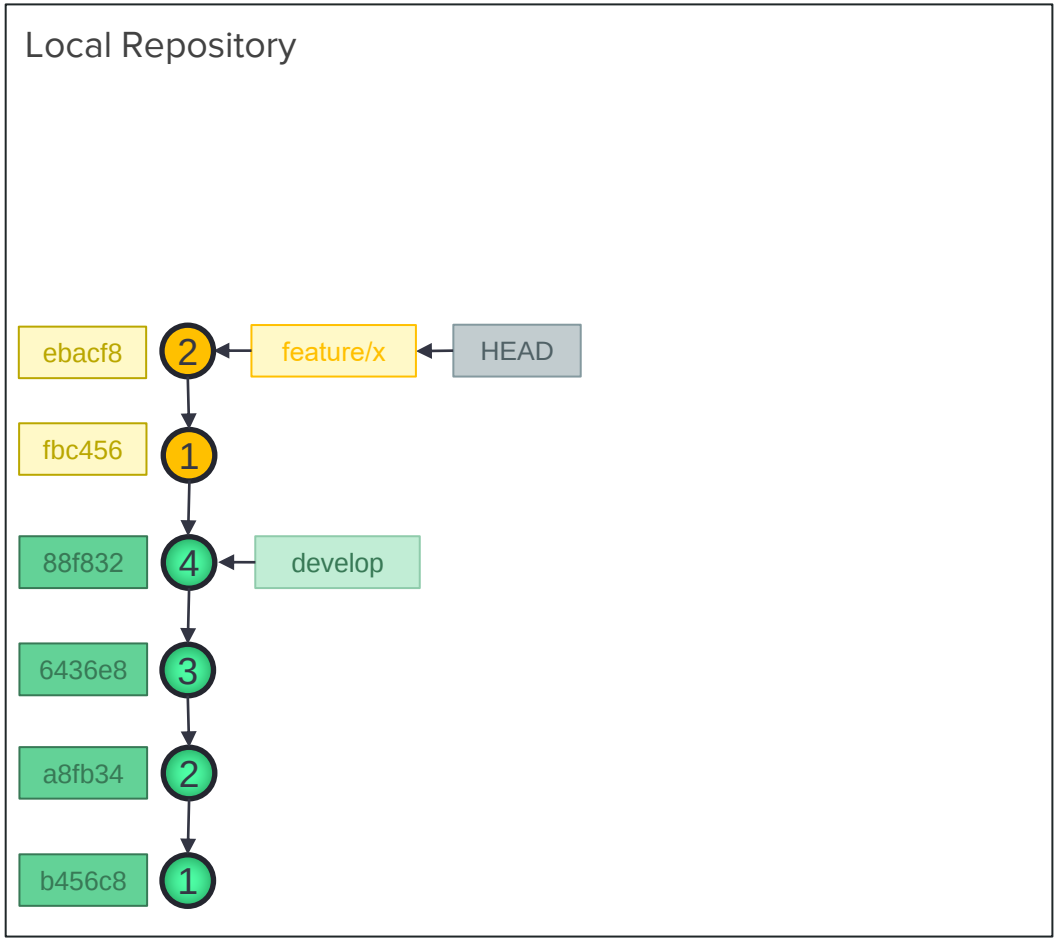
Rebasing

```
$ git rebase develop

$ |
```



Rebasing

```
$ git rebase develop

$ |
```



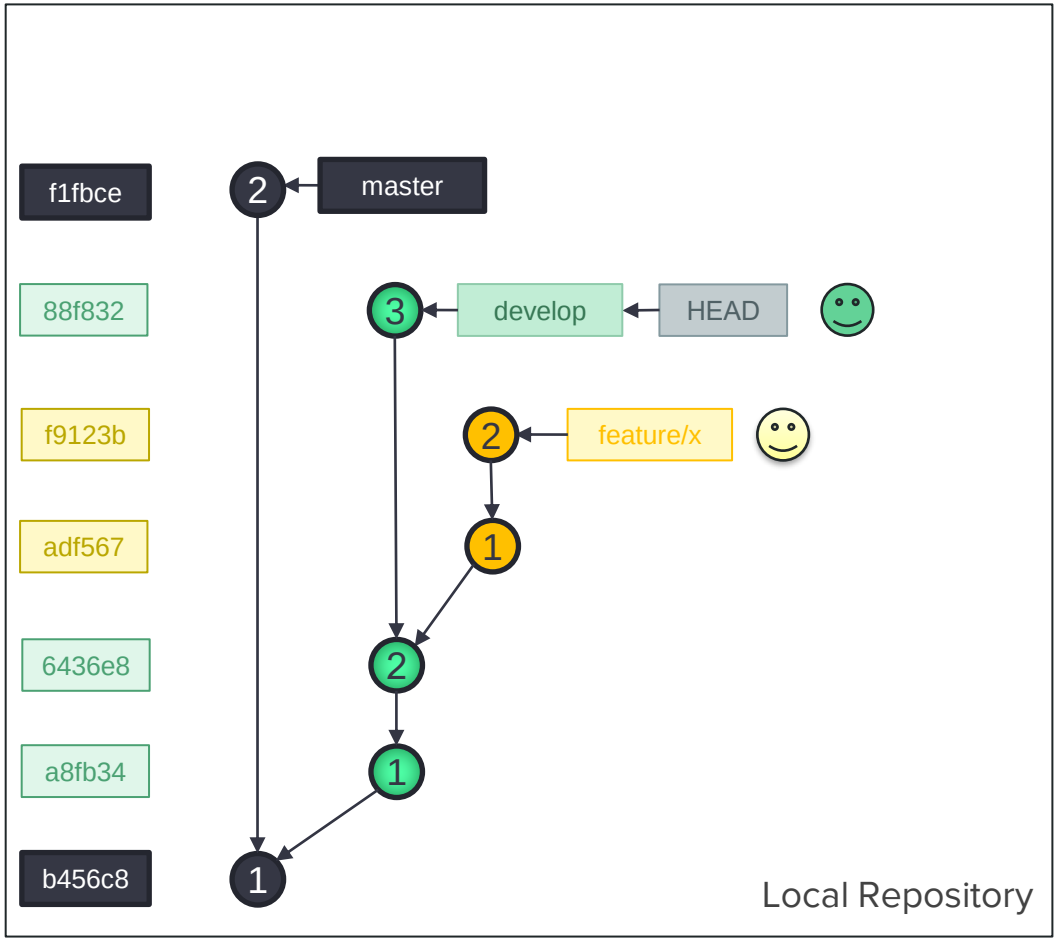
Rebasing

- The biggest benefit is a much cleaner history
  - Eliminates merge commits
  - Results in linear history

- The tradeoffs are:
  - Safety: re-writing project history can be potentially catastrophic for your collaboration workflow
  - Traceability: you can’t see when upstream changes were incorporated into the feature
  - Increased difficulty when there are conflicts

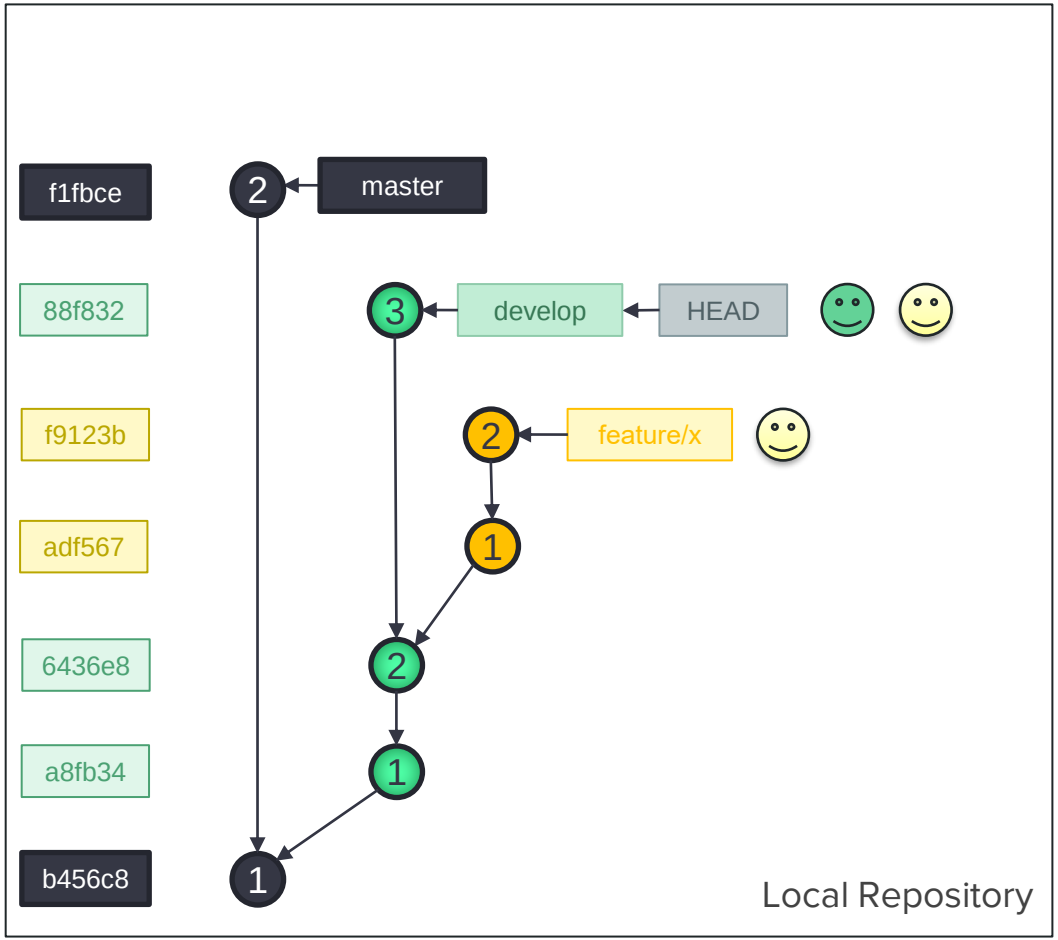
Who’s Looking?

```
$ |
```



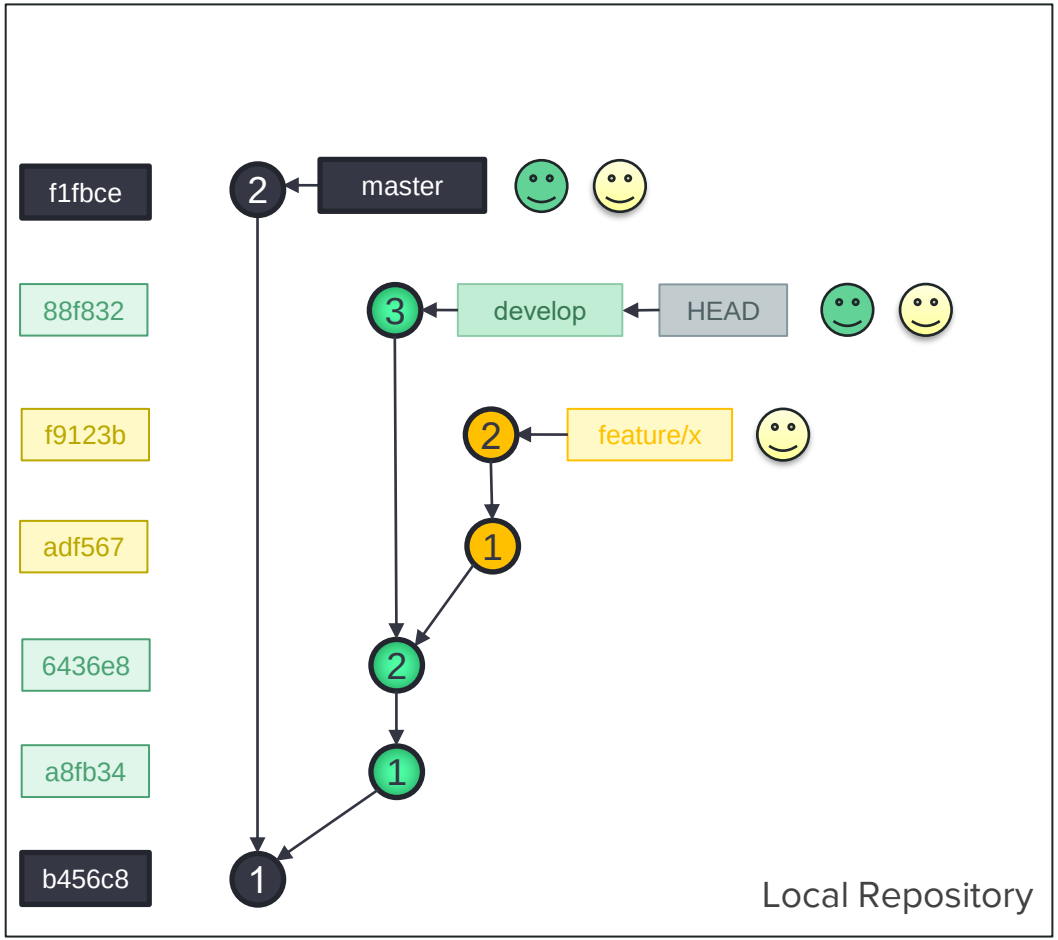
Who’s Looking?

```
$ |
```



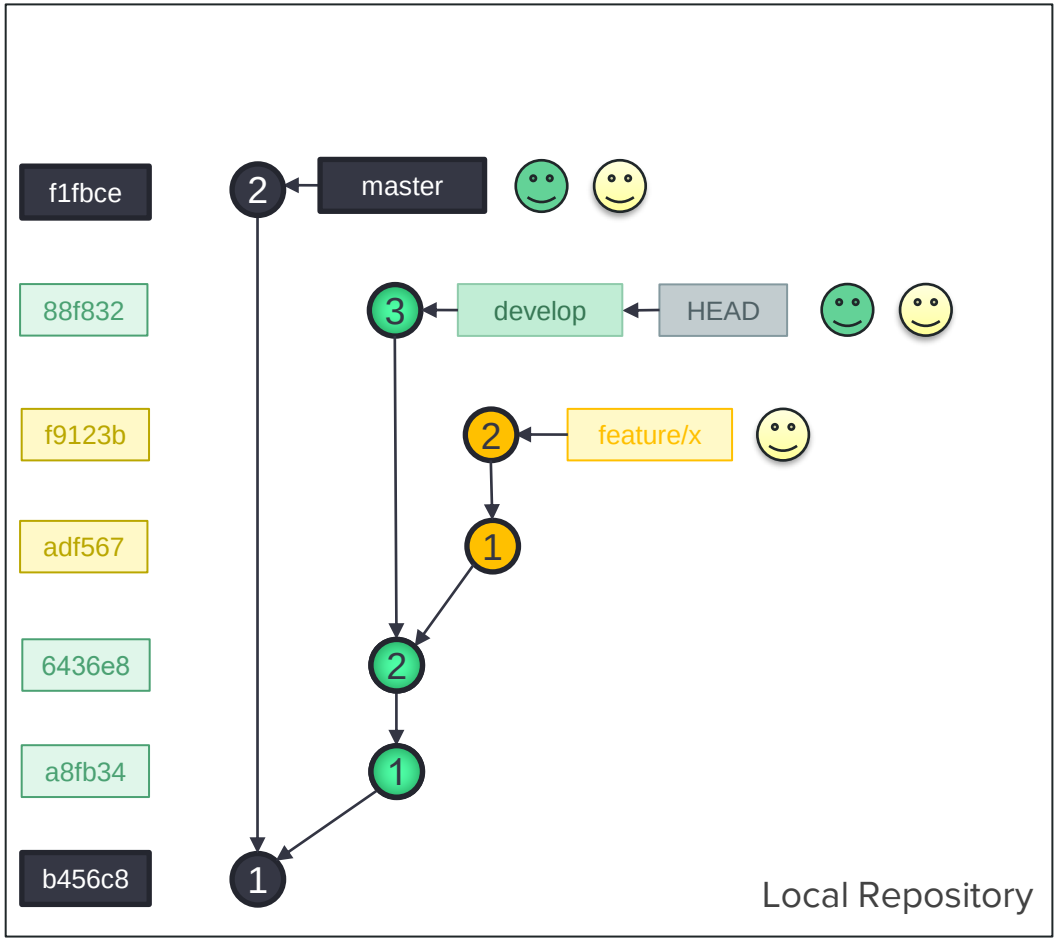
Who’s Looking?

```
$ |
```



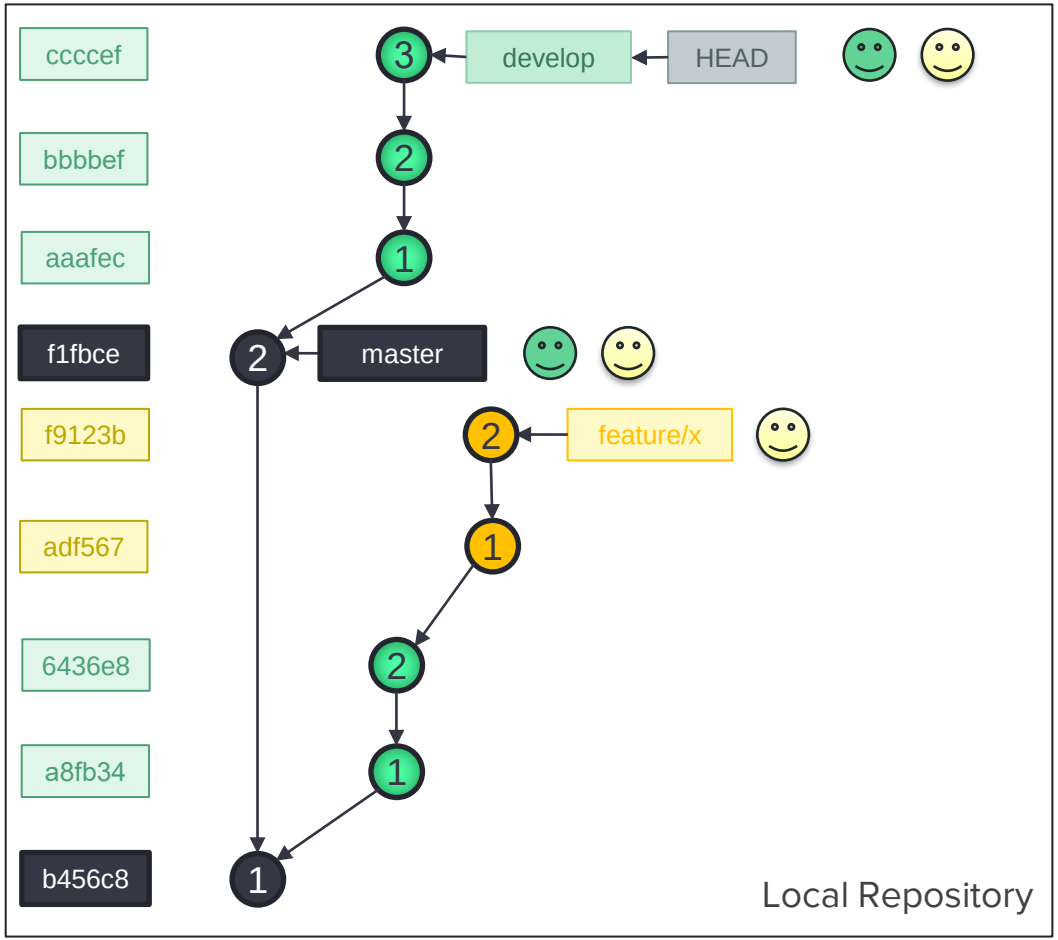
What can go wrong?

```
$ git rebase master

$ |
```



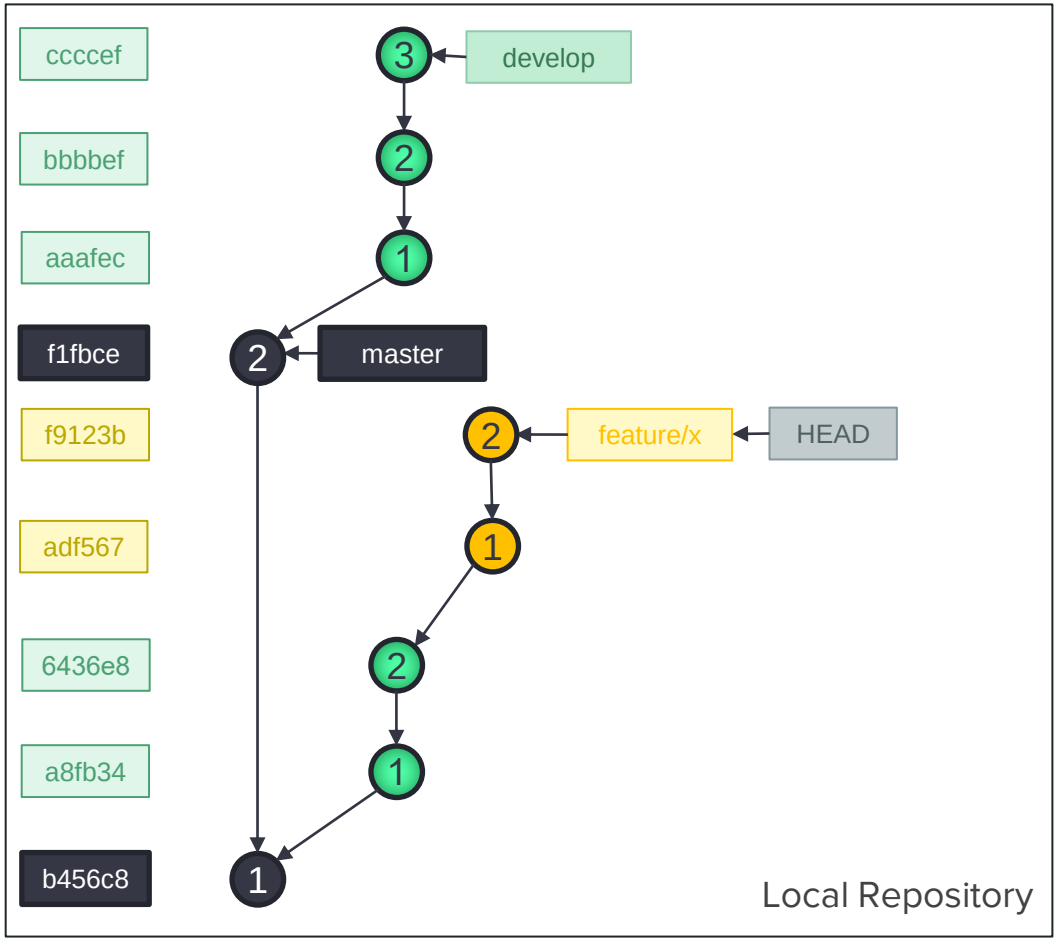
What can go wrong?

```
$ git rebase master

$ |
```



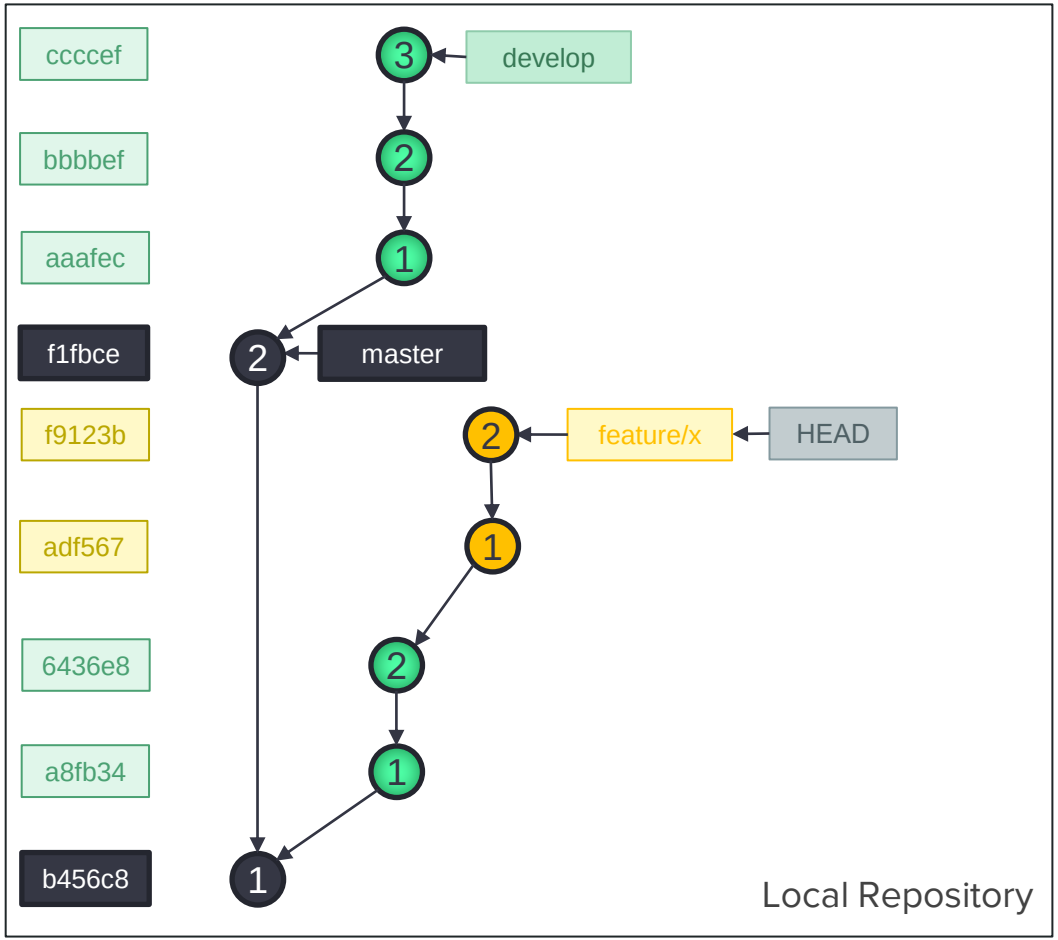
Can I merge?

```
$ git rebase master

$ git checkout feature

$ |
```



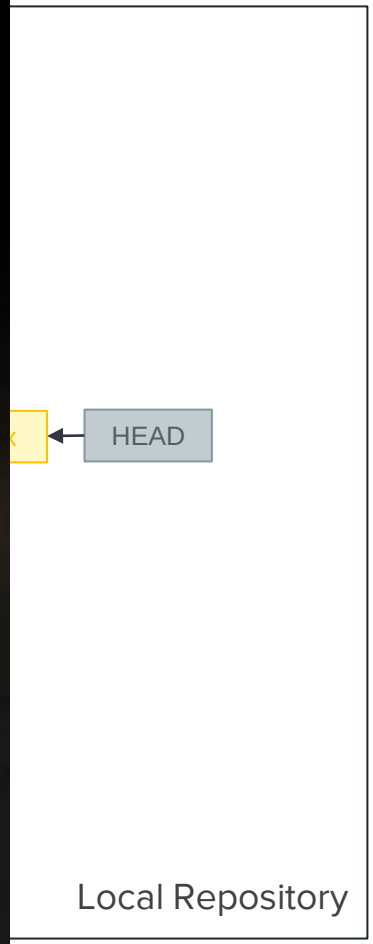
Can I merge?

```
$ git rebase master

$ git checkout feature

$ git merge develop

$ |
```



Can I merg

```
$ git rebase

$ git checkou

$ git merge

$ |
```




noooooooo!
nooo!
quickmeme.com

What can go wrong?

- Rebasing a shared branch results in duplicate commits
  - Duplicate commits are commits with the same content but different hash (different parent)
  - One user sees that content on one branch (develop), the other user sees the same commits with different hashes on their own branch (feature)
- Trying to merge these commits results in **guaranteed merge conflicts** and a **guaranteed messed up history**
- Rebasing the feature branch also yields conflicts only **it will do so every time it applies one of the commits** so it's **even worse**

Remember

- While this example was done completely locally it is **completely valid for working with remotes**!
- Remote branches are just like normal branches they just have to be **fetched first** but the behavior is the same after fetching
- **Never rebase and push --force!** That makes developers cry.

History Manipulation

Squashing
(Jedi Master-level Rebasing)

Squashing

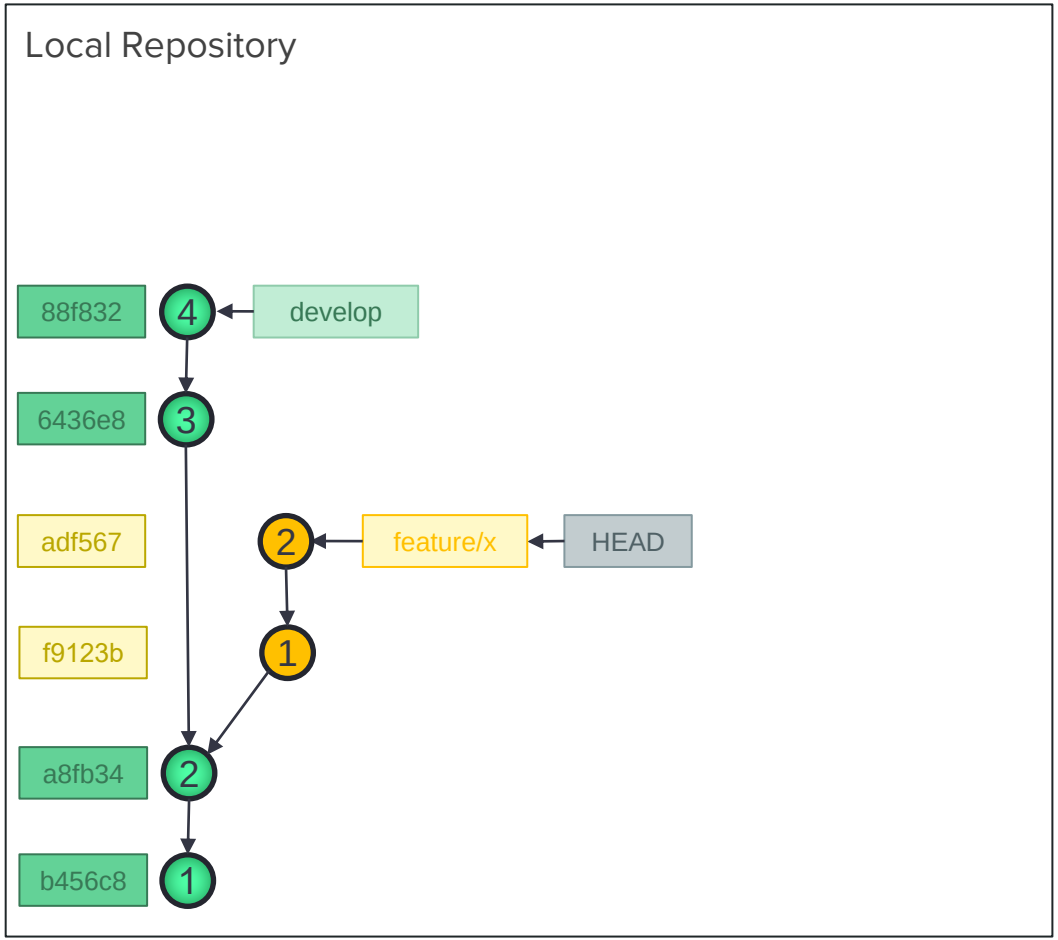
- Is essentially part of a rebase operation, specifically an interactive rebase

- **Standard:** git rebase <base>
- **Interactive:** git rebase --interactive <base>

- Being a rebase it has **all the same pitfalls and gotchas**, but also has one important advantage: **Complete control over how the rebase is performed, every step of the process.**

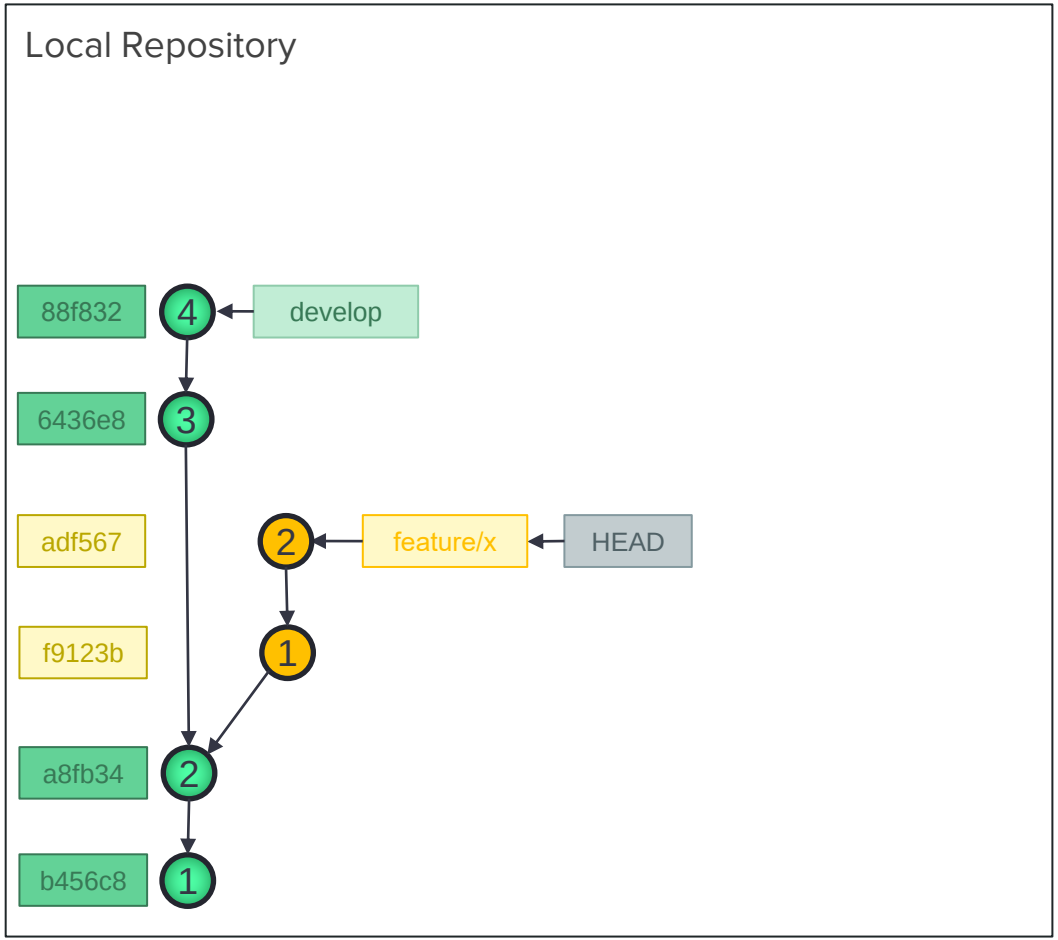
Squashing

```
$ |
```


Local Repository
88f832
4
develop
6436e8
3
adf567
2
feature/x
HEAD
f9123b
1
a8fb34
2
b456c8
1

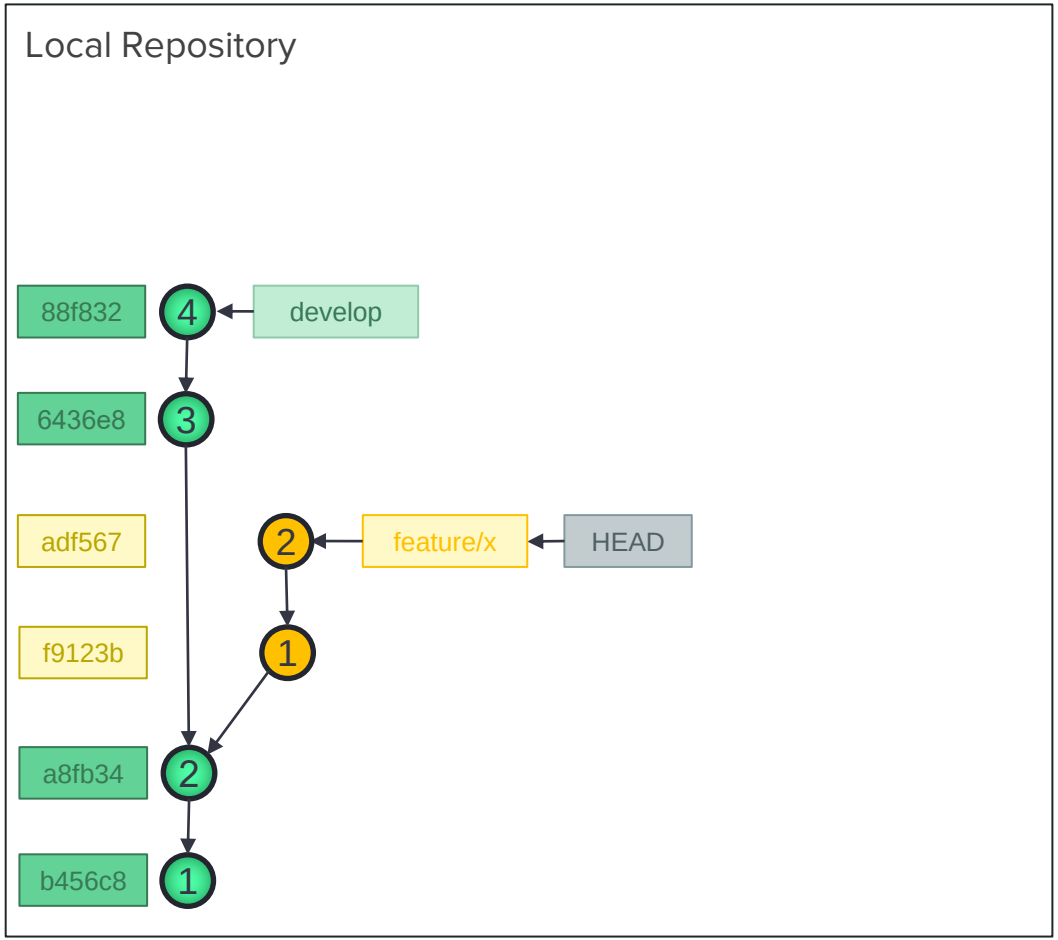
Squashing

```
$ git rebase -i HEAD~2

$ |
```



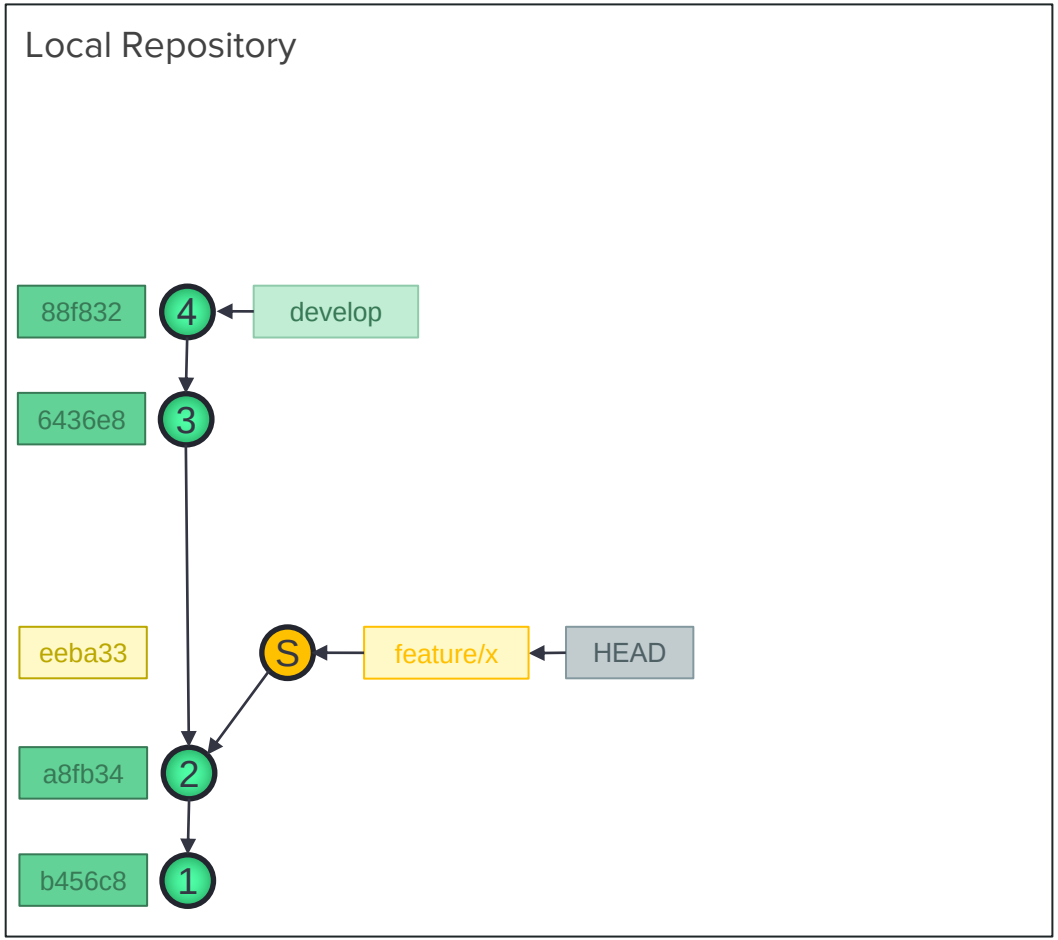
Squashing

```
pick f9123b Commit 1
squash adf567 Commit 2

# Rebase f9123b..adf567
onto a8fb34 (2 commands)
```



Squashing

```
$ git rebase -i HEAD~2

$ |
```



Note

- You can also use „**reword”** instead of „pick” and „**fixup”** instead of „squash” to cotrol the final commit message of the squashed commit

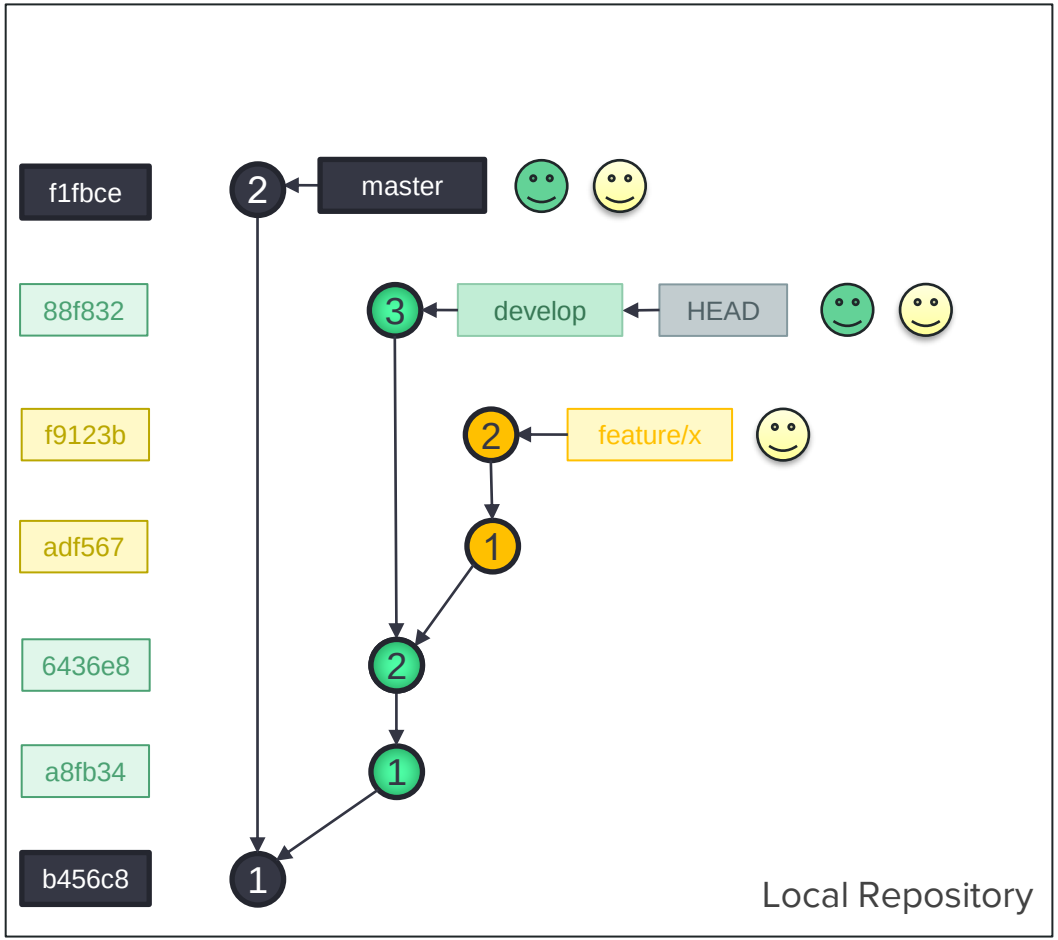
Squashing

- Benefits
  - Atomic features (the whole feature in one single commit)
  - Much simpler history

- The tradeoffs are:
  - Safety: re-writing project history can be potentially catastrophic for your collaboration workflow
  - Traceability: you can’t see when upstream changes were incorporated into the feature
  - History: It disables undoing of part of the feature, it’s all or nothing

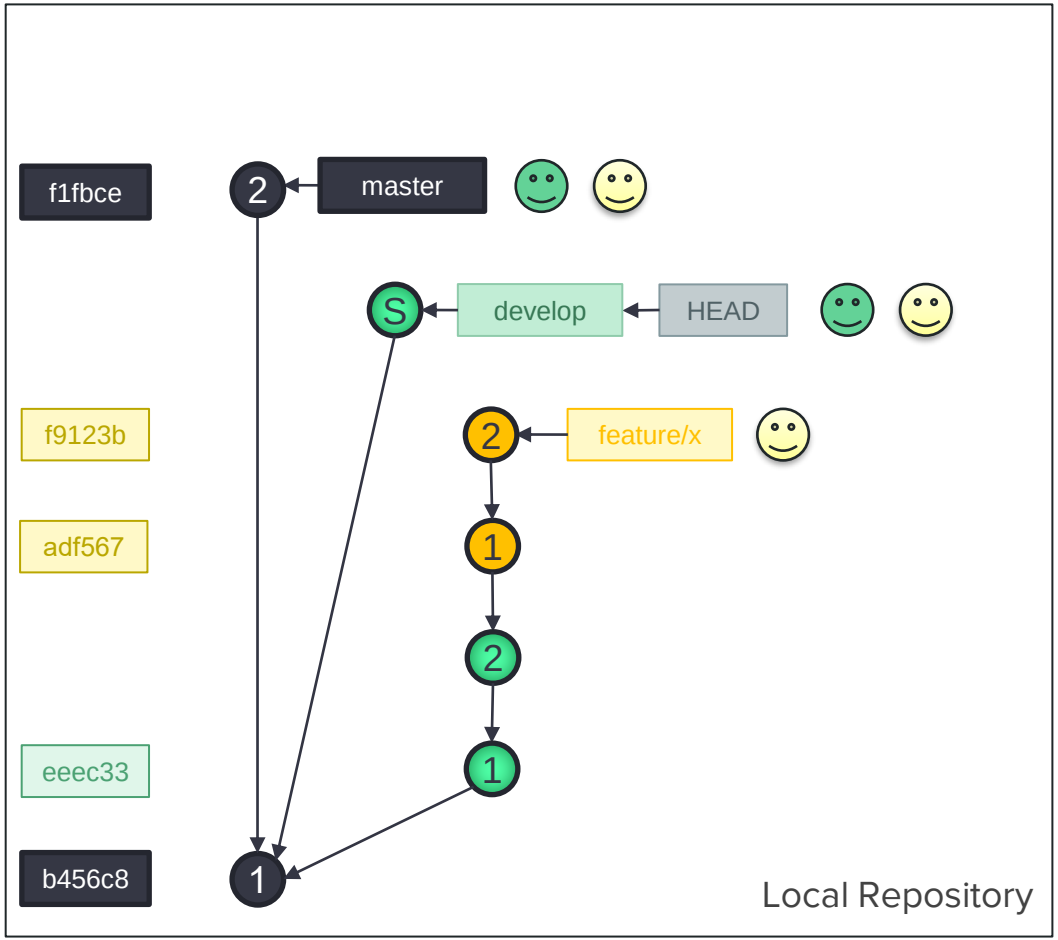
What can go wrong?

```
$ git rebase -i HEAD~3

$ |
```



What can go wrong?

```
$ git rebase -i HEAD~3

$ |
```



What can go wrong?

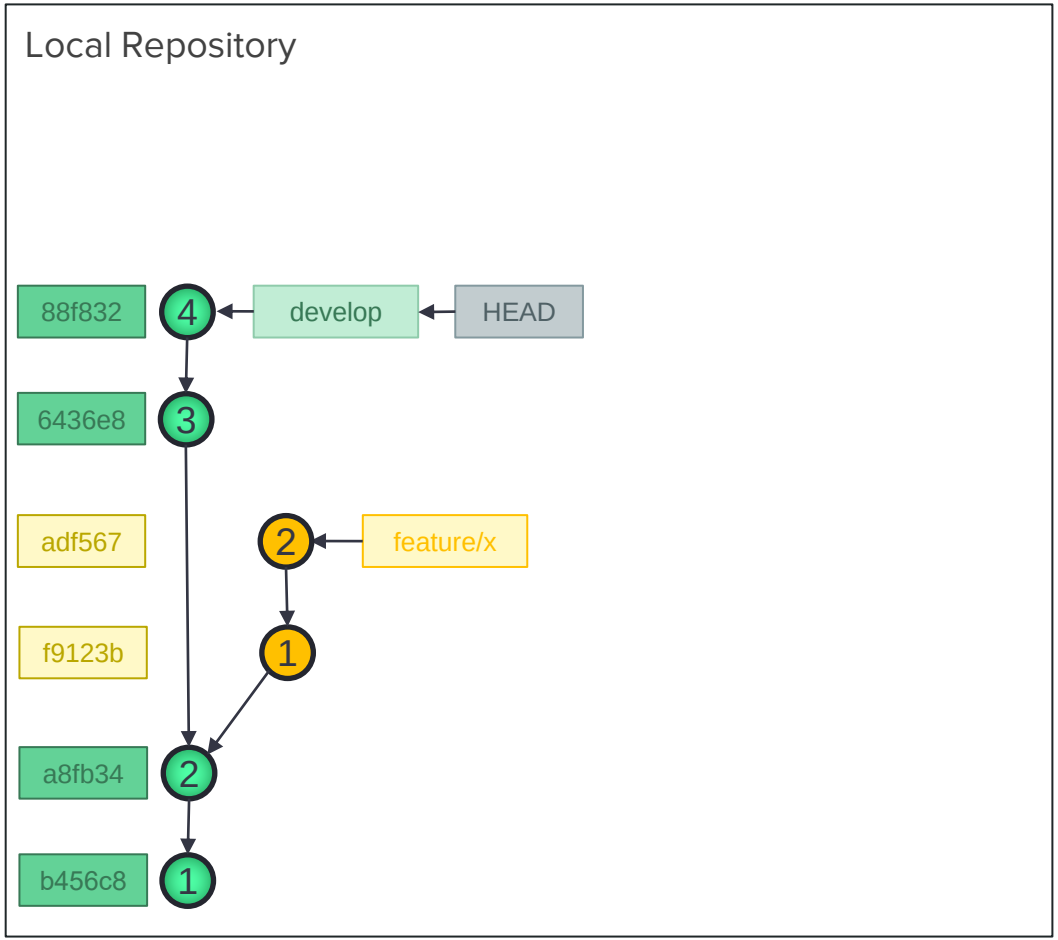
- The squash commit now contains the changes from both the 1 and 2 commit
- Merging will yield **conflicts** again, as will rebasing the feature commit onto the develop branch
- Developers on feature branches will see the same changes twice in different commits which is confusing

History Manipulation (not quite but still)

Cherry Picking

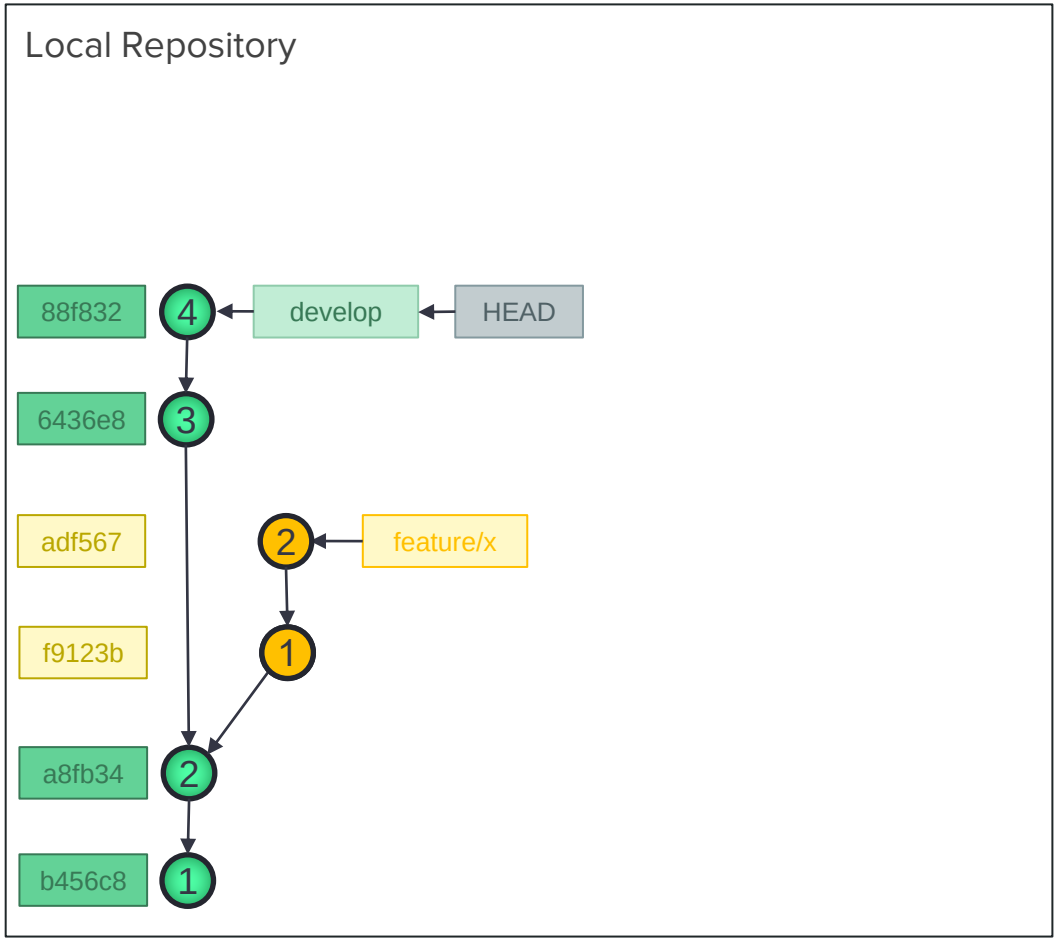
Cherry Picking

```
$ |
```



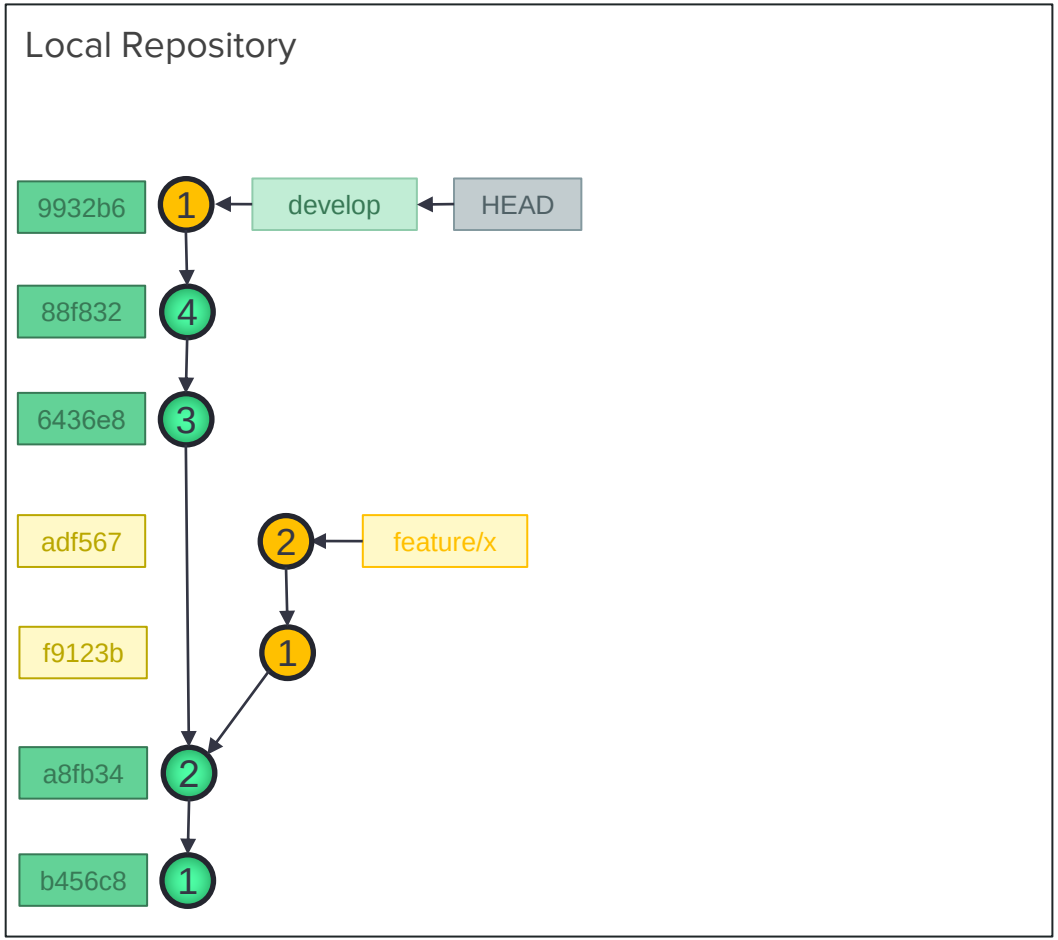
Cherry Picking

```
$ git cherry-pick
  f9123b

$ |
```



Cherry Picking

```
$ git cherry-pick
  f9123b

$ |
```



Cherry Picking

- Benefits
  - Most useful for hot-fixing or pulling developed modules from other sources without merging
  - Does not require a full merge with unfinished features or breaking changes

- The tradeoffs are:
  - Creates a new commit with the exact same changes on a new branch which creates a duplicate of the changes
  - Cherry picking multiple commits is a tempting and confusing alternative to a much simpler merge

What can go wrong?

- The cherry picked commit contains the same changes already on the feature branch

- Merging with the feature branch can yield bizarre **conflicts**, as will rebasing the feature

The Golden Rule of History Manipulation

The Golden Rule

The Golden Rule

- Before doing any history manipulation ask yourself:

Is anyone else looking at this branch?

Yes

No

The Golden Rule

Is anyone else looking at this branch?

- Is this branch pushed, is it a public branch?
- Is this branch a source branch for other branches?

YES!

Somebody else is also looking at this branch!


STOP

Tips and Tricks

Tips and Tricks

Branching Models

Branching Models

- Benefits
  - Create a rule framework to follow when working in a team
  - Help avoid merge issues
  - Ensure pseudo-stable development branches and stable release branches

- The tradeoffs are:
  - Rules to learn and follow (can be managed with tools)
  - Higher complexity git logs (can be managed with a proper workflow)

Git Flow

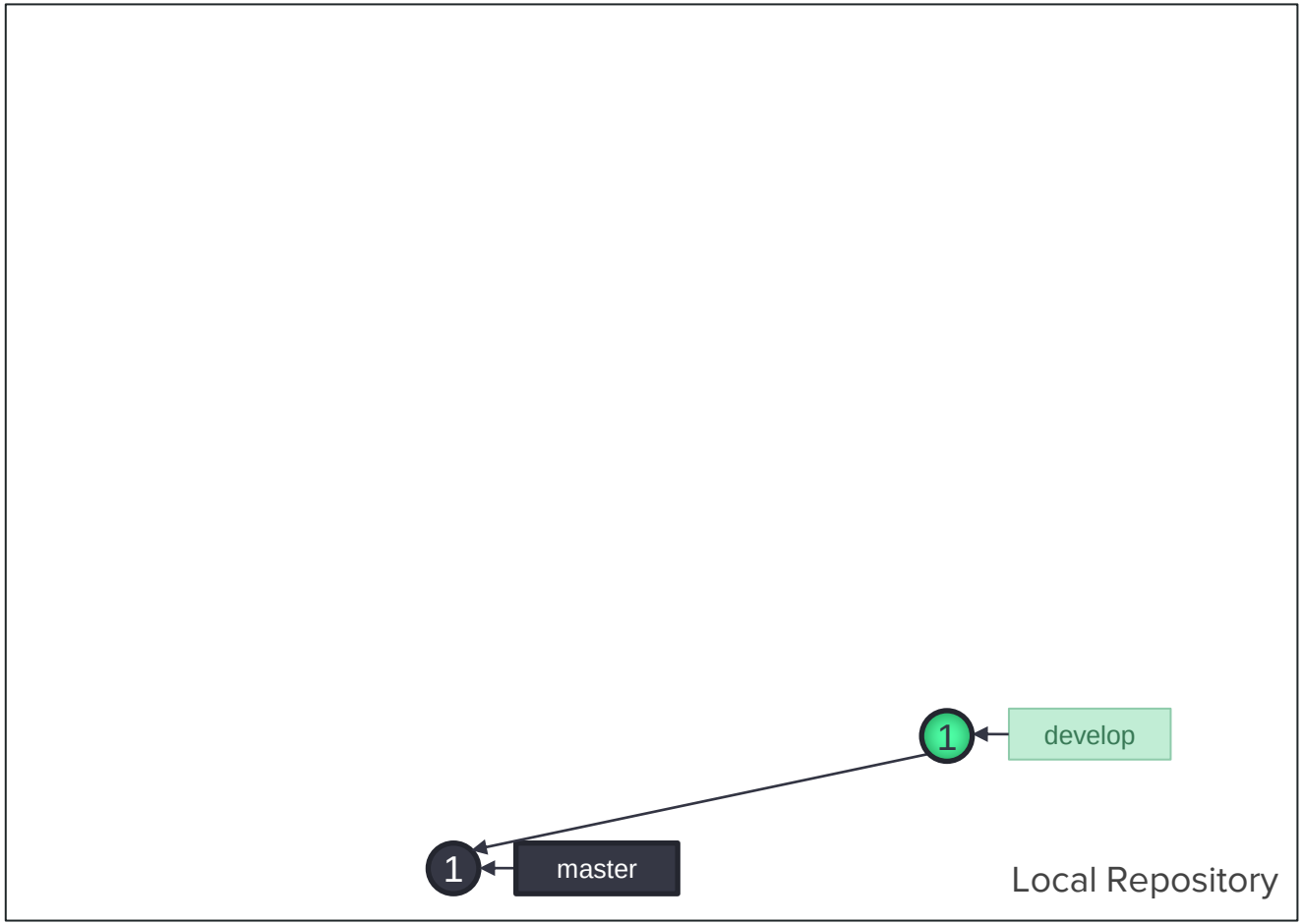
master == production

Git Flow

develop == staging

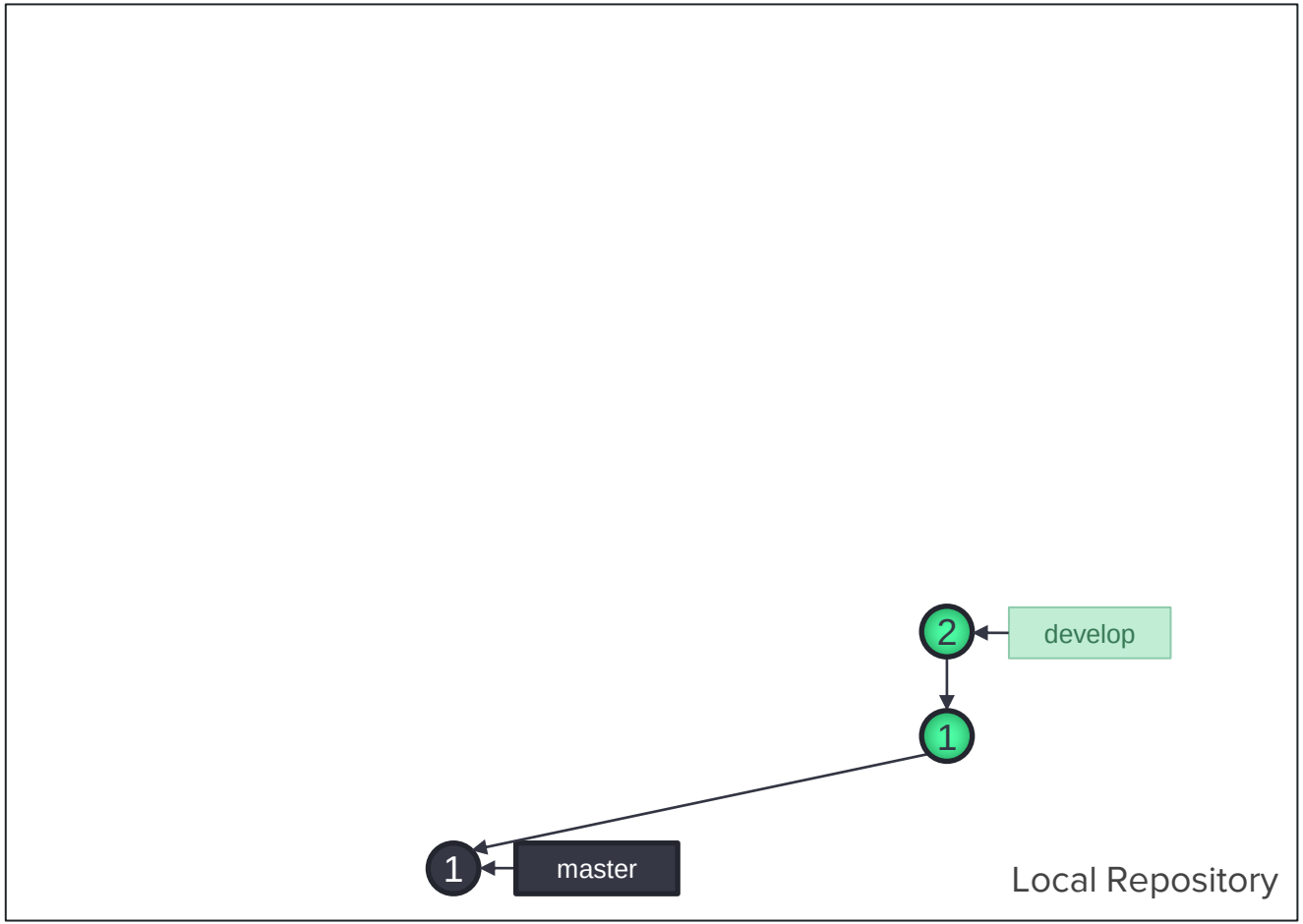
Git Flow

develop == staging



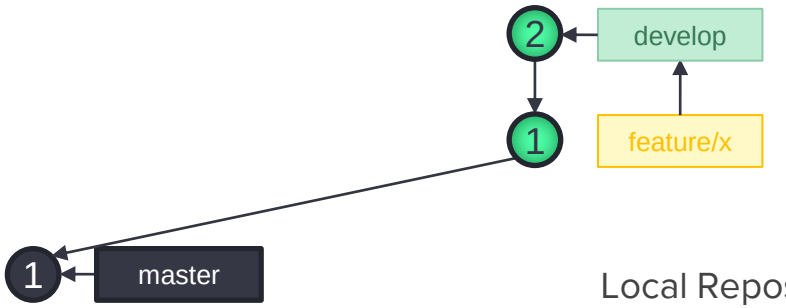
Git Flow

develop == staging



Git Flow

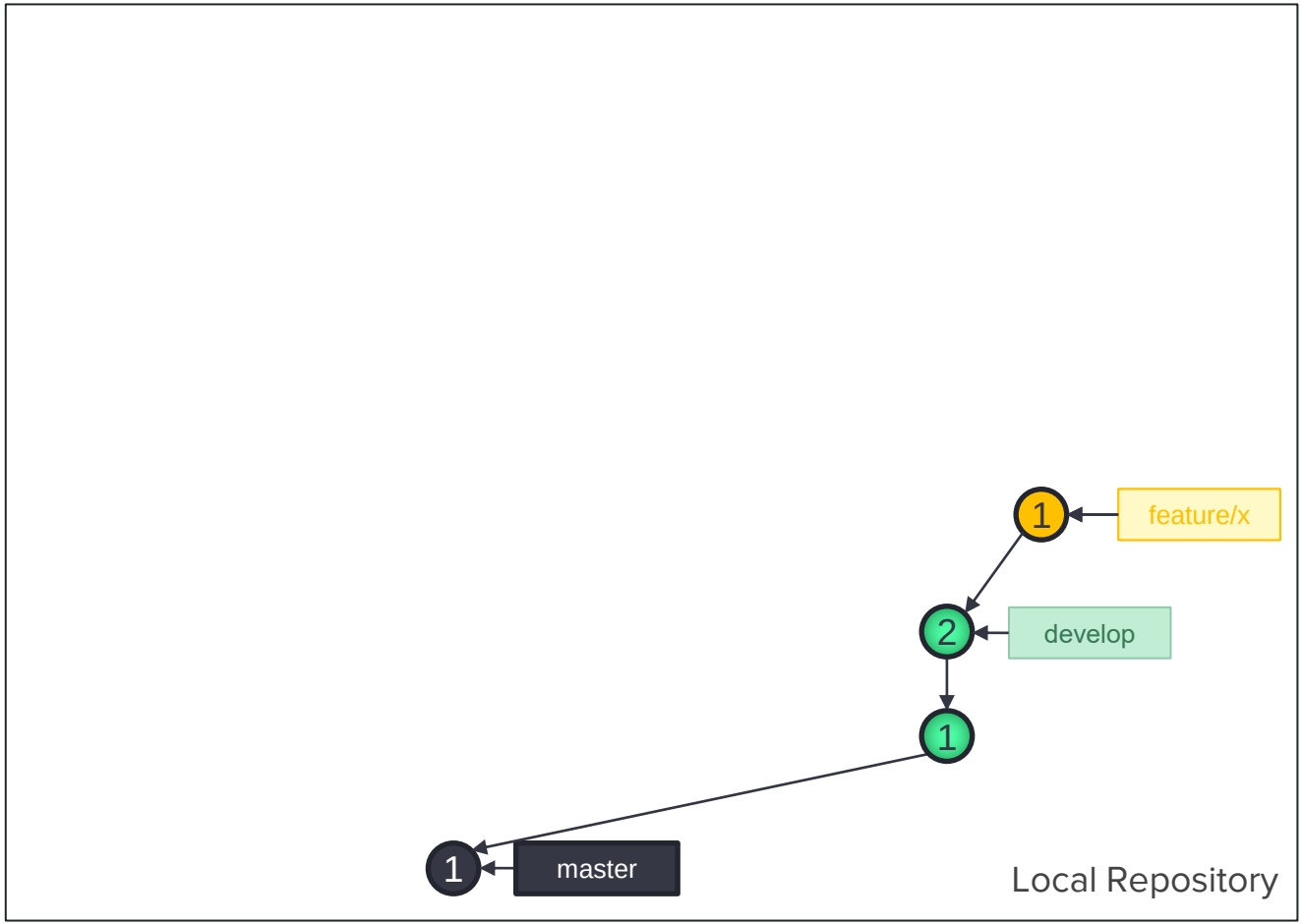
feature/x == developer



Local Repository

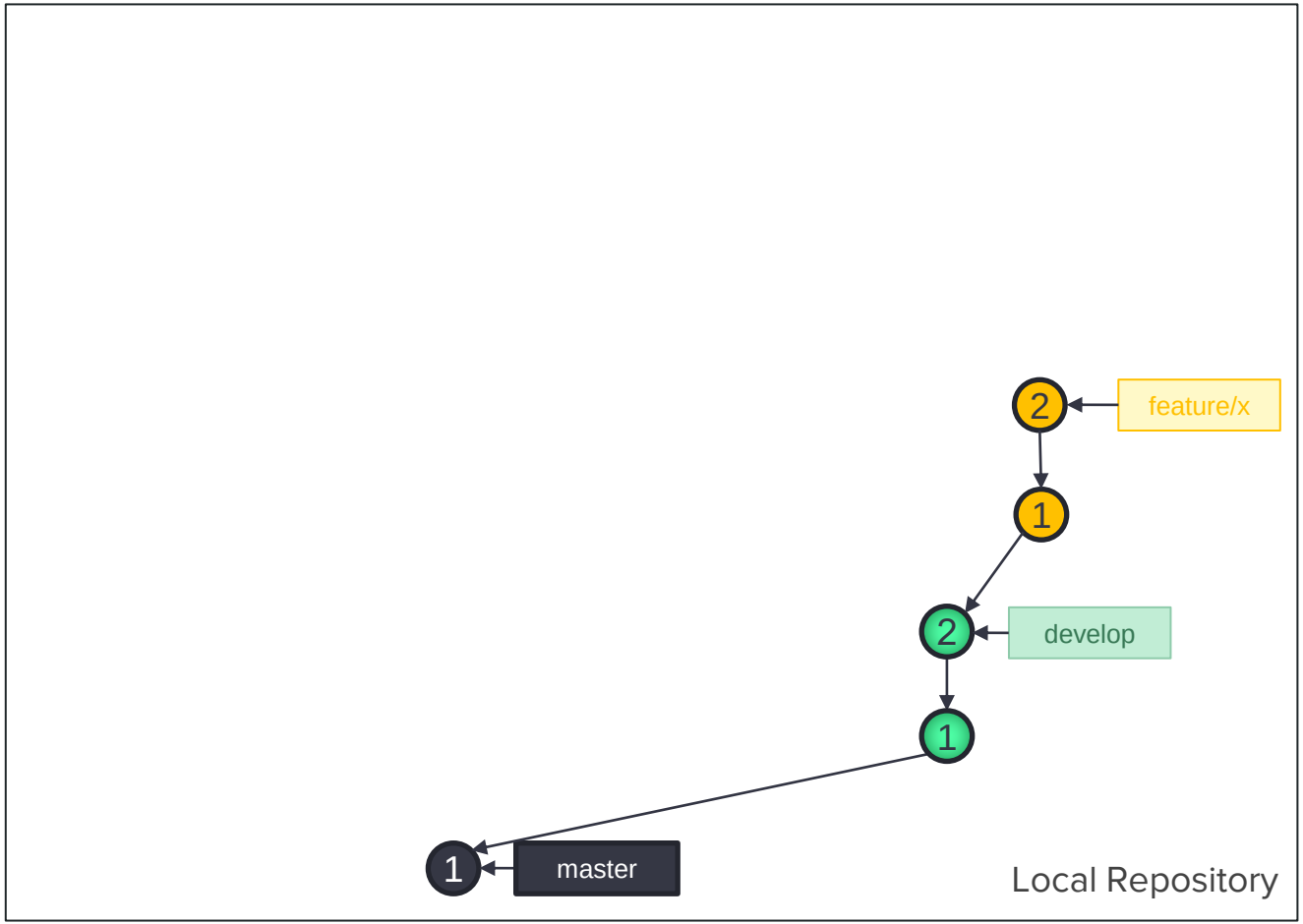
Git Flow

feature/x == developer



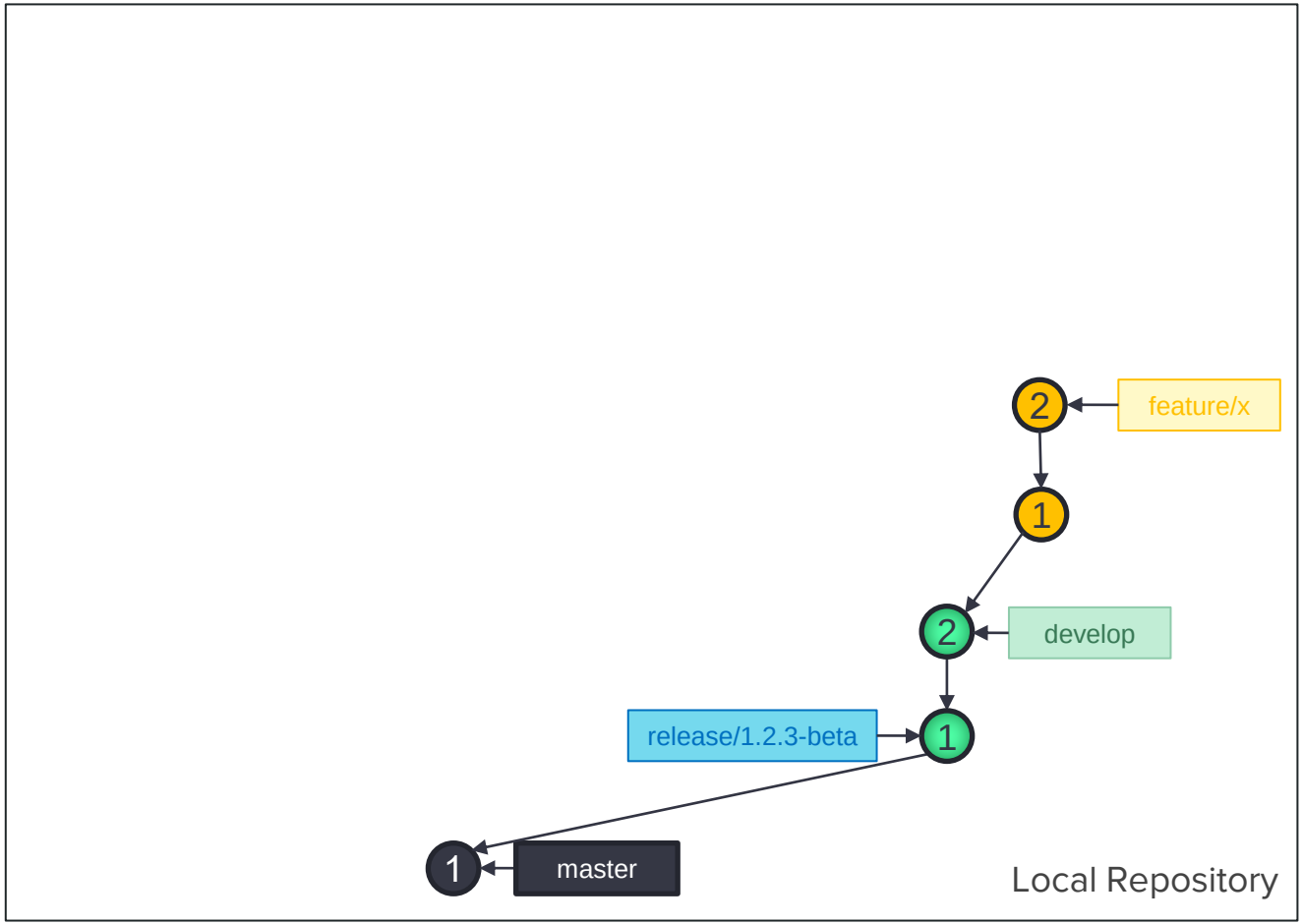
Git Flow

feature/x == developer



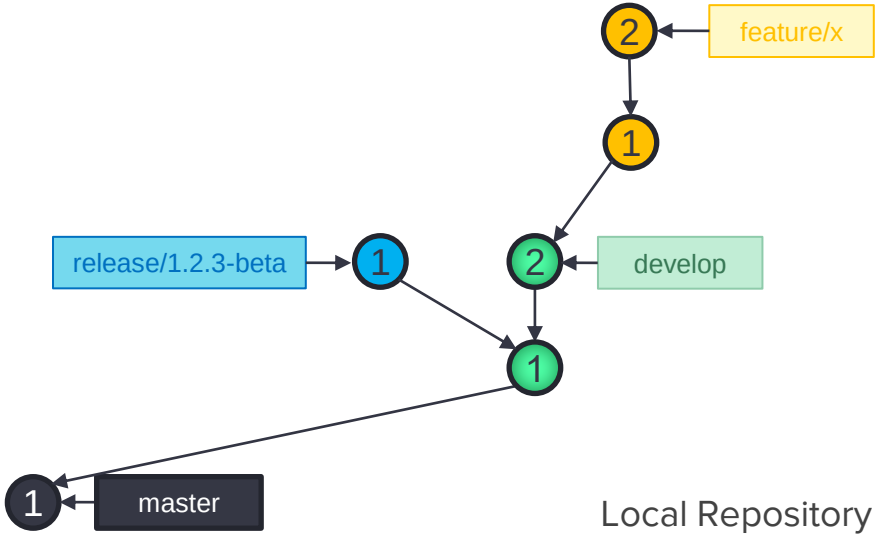
Git Flow

`release/x == QA`



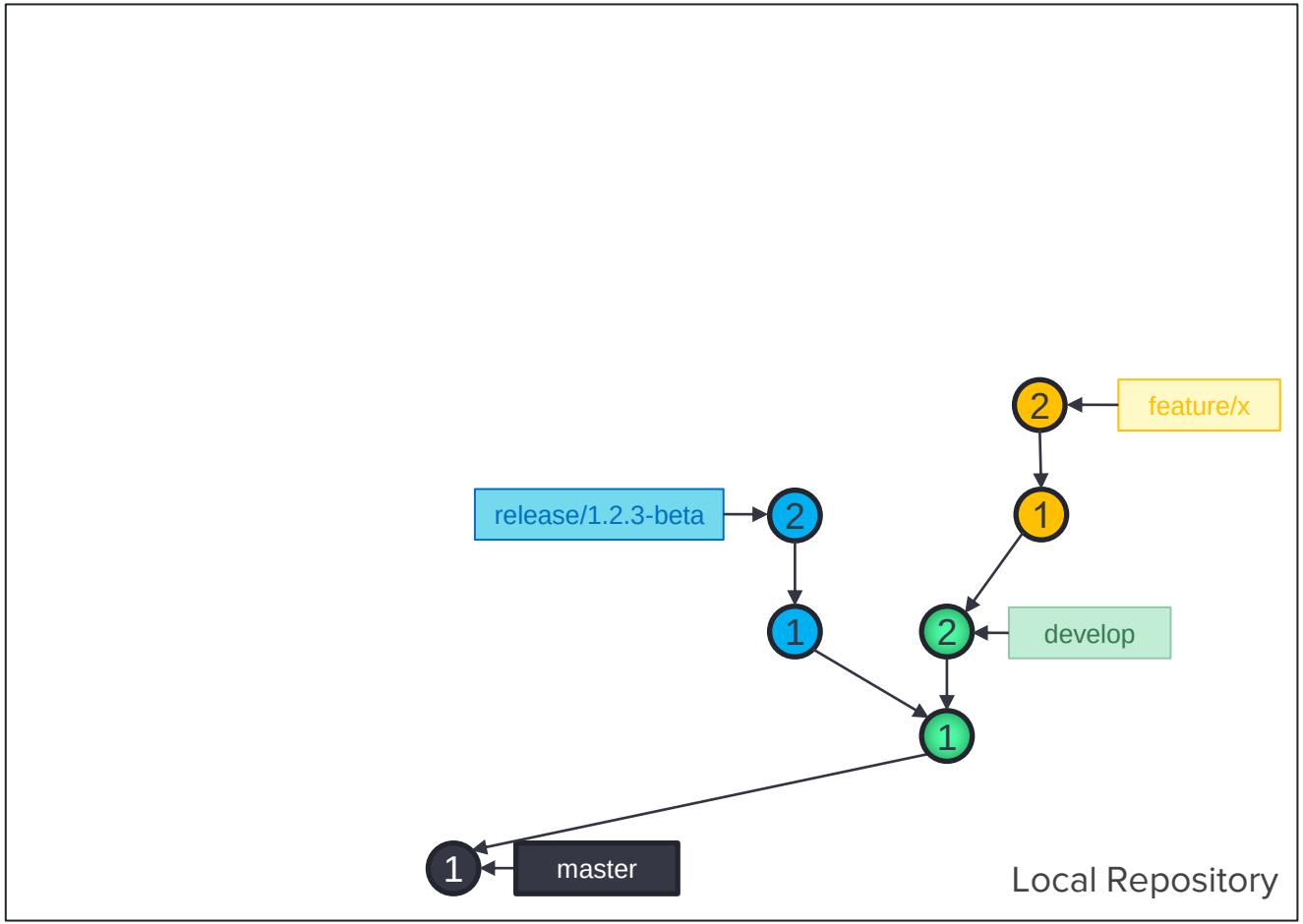
Git Flow

`release/x == QA`



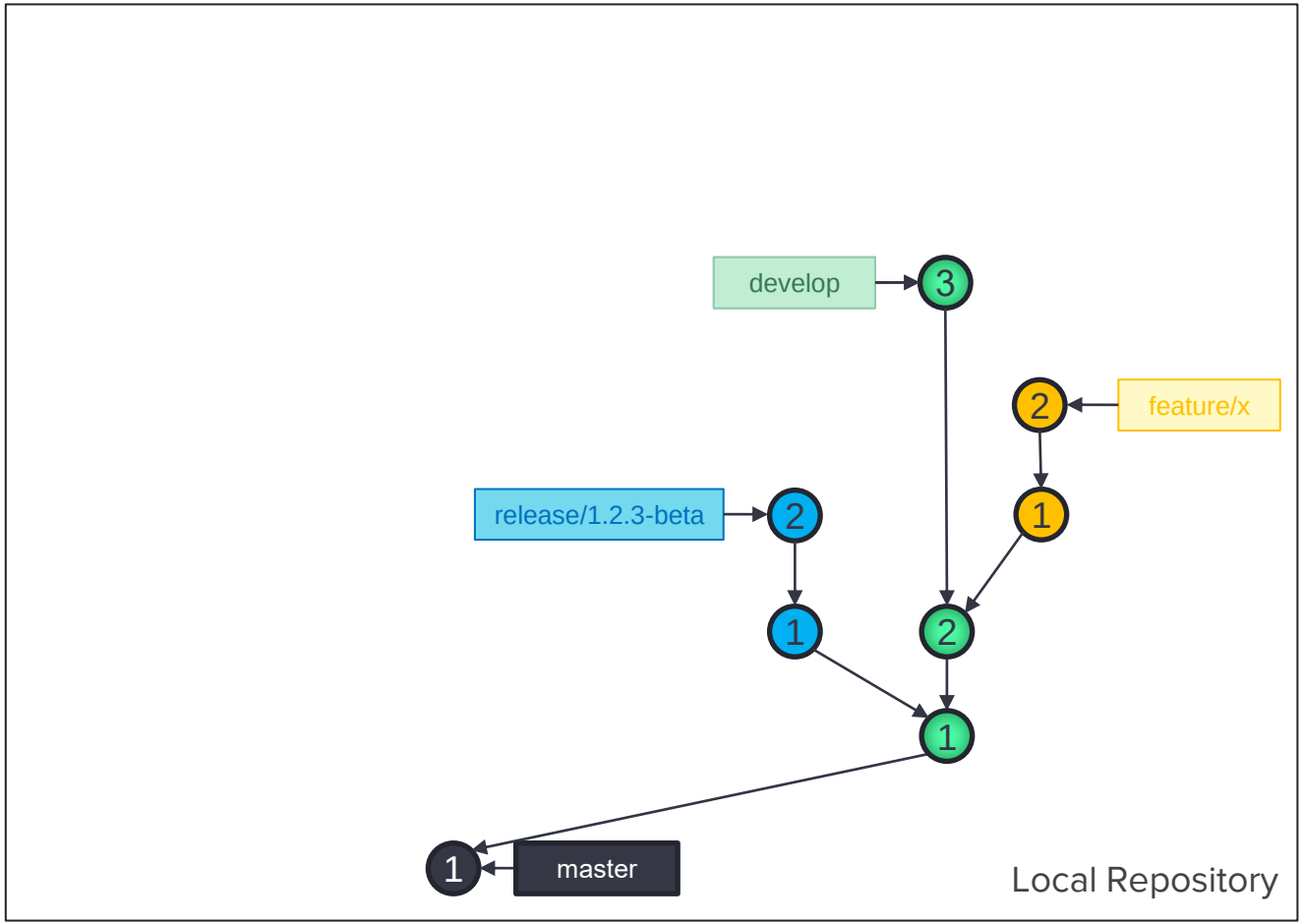
Git Flow

`release/x == QA`



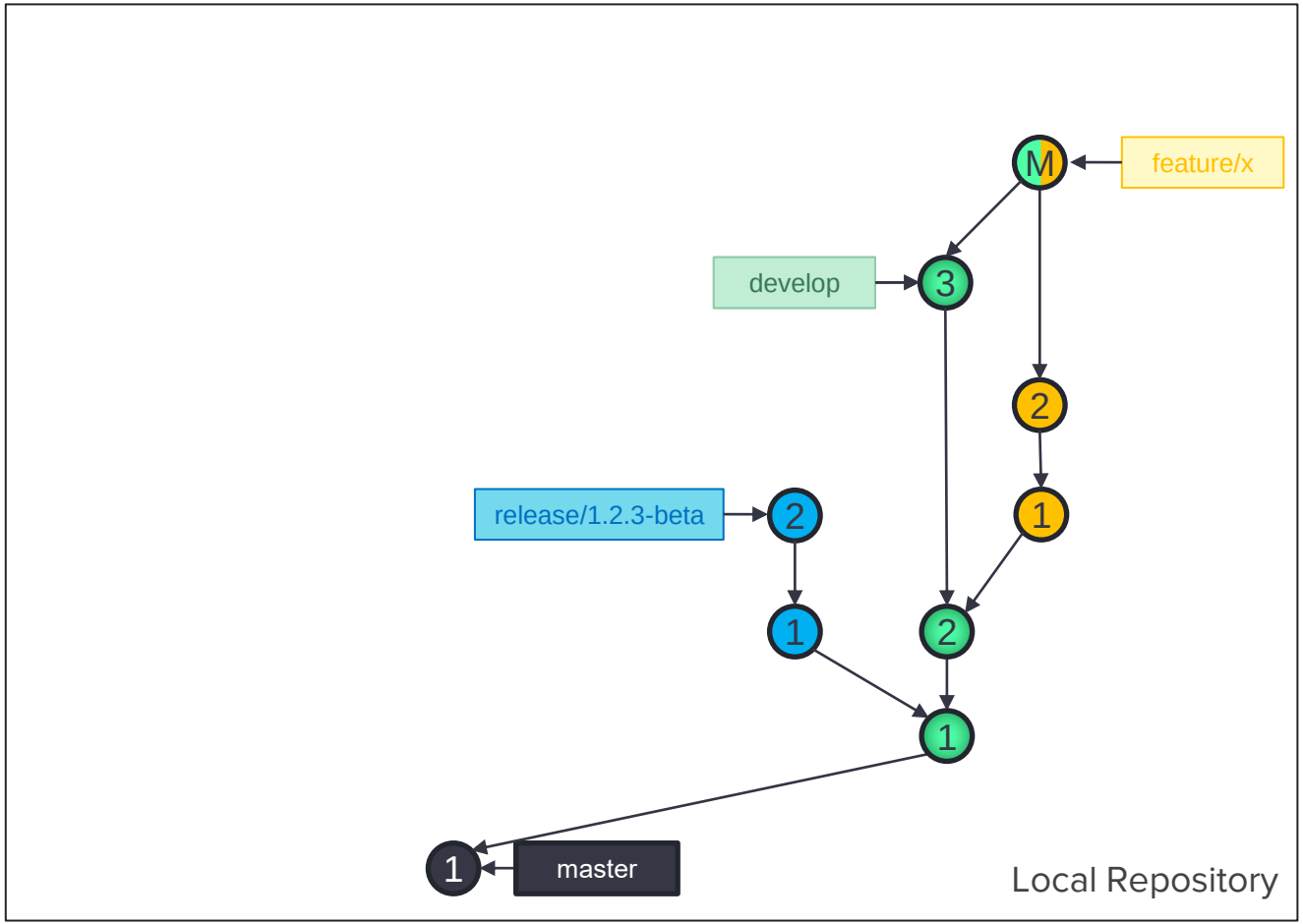
Git Flow

```
Merge develop into feature
```



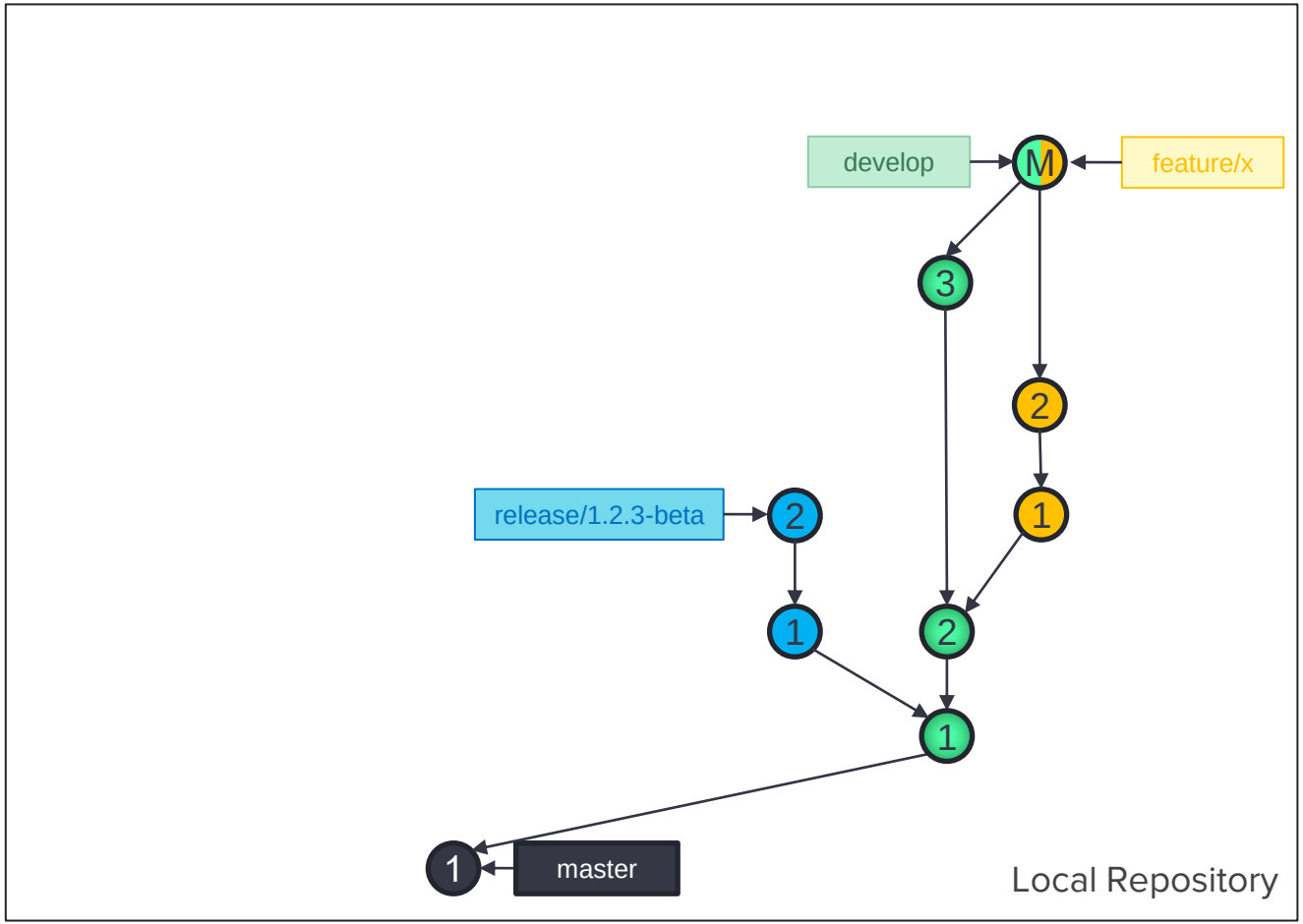
Git Flow

```
Merge develop into feature
```



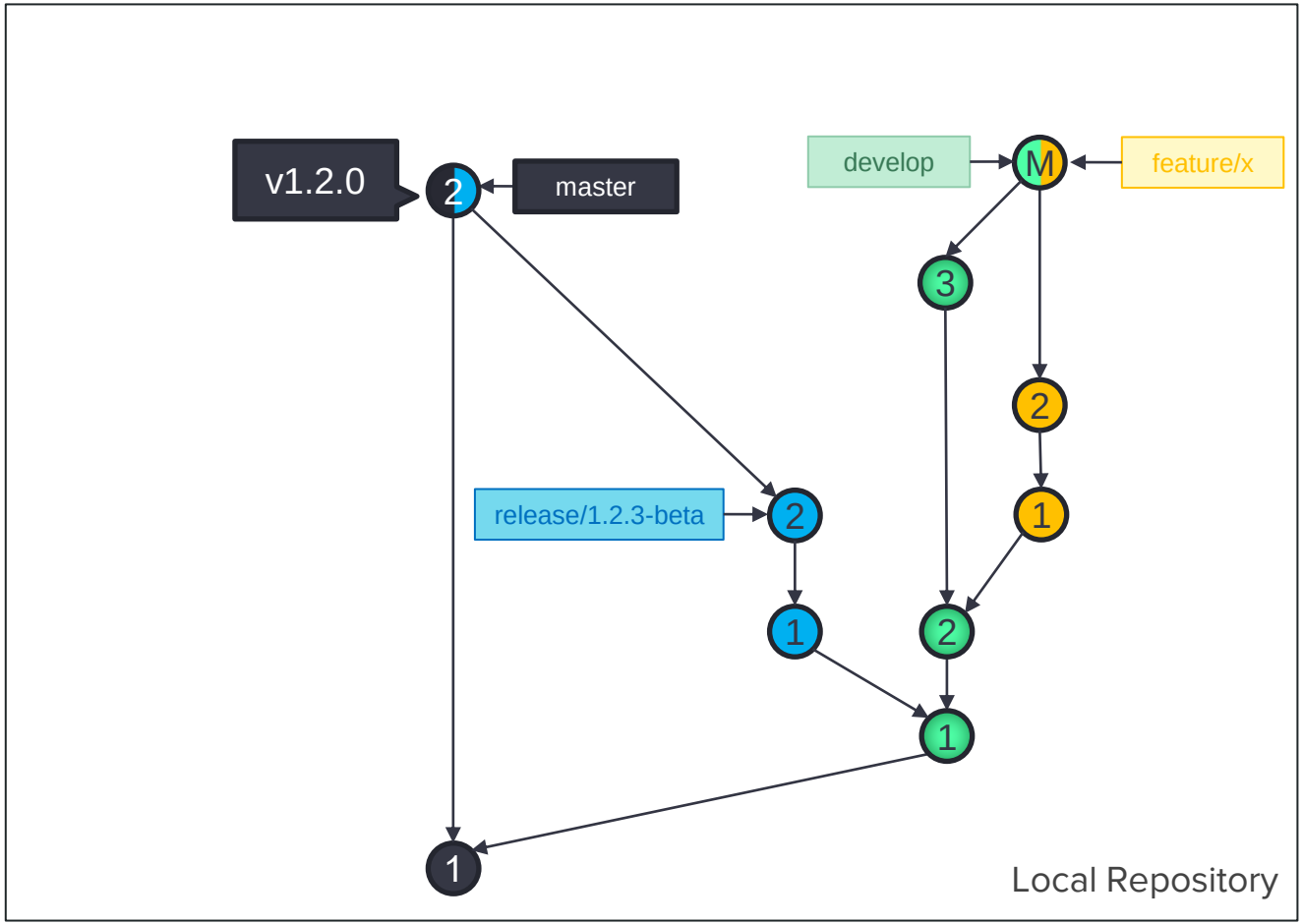
Git Flow

```
Fast-forward develop
```



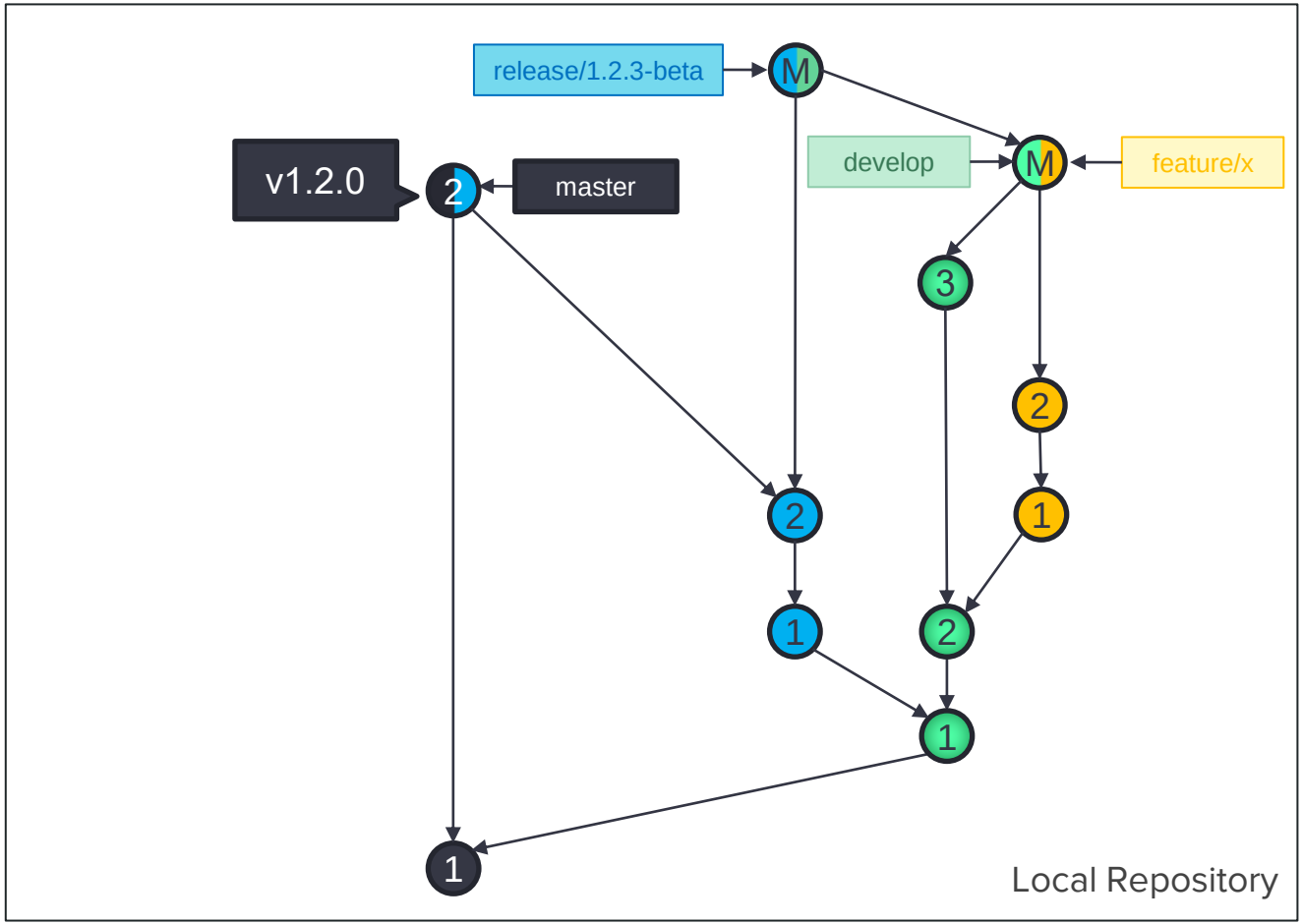
Git Flow

```
Merge release into master
```



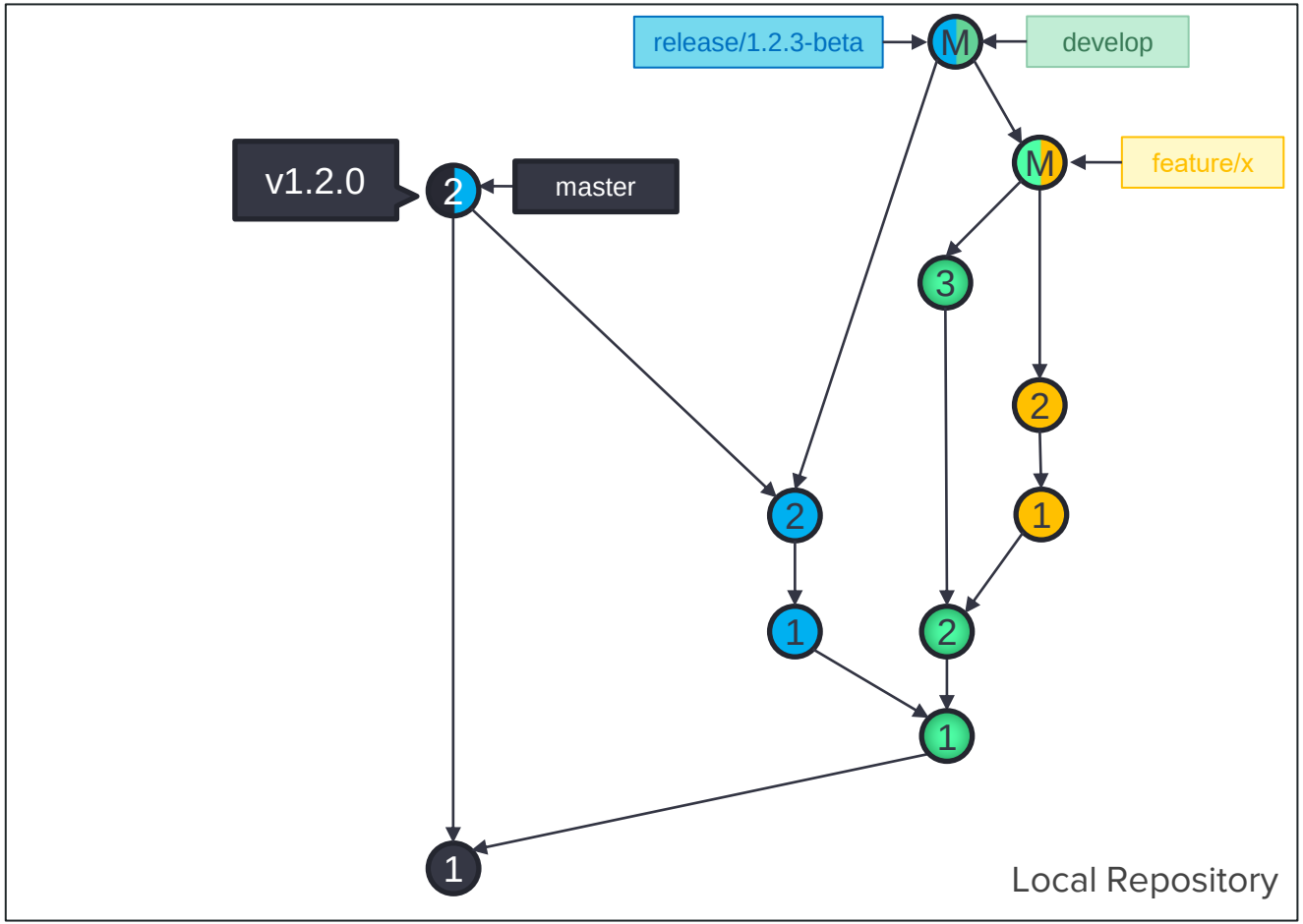
Git Flow

Merge develop into release



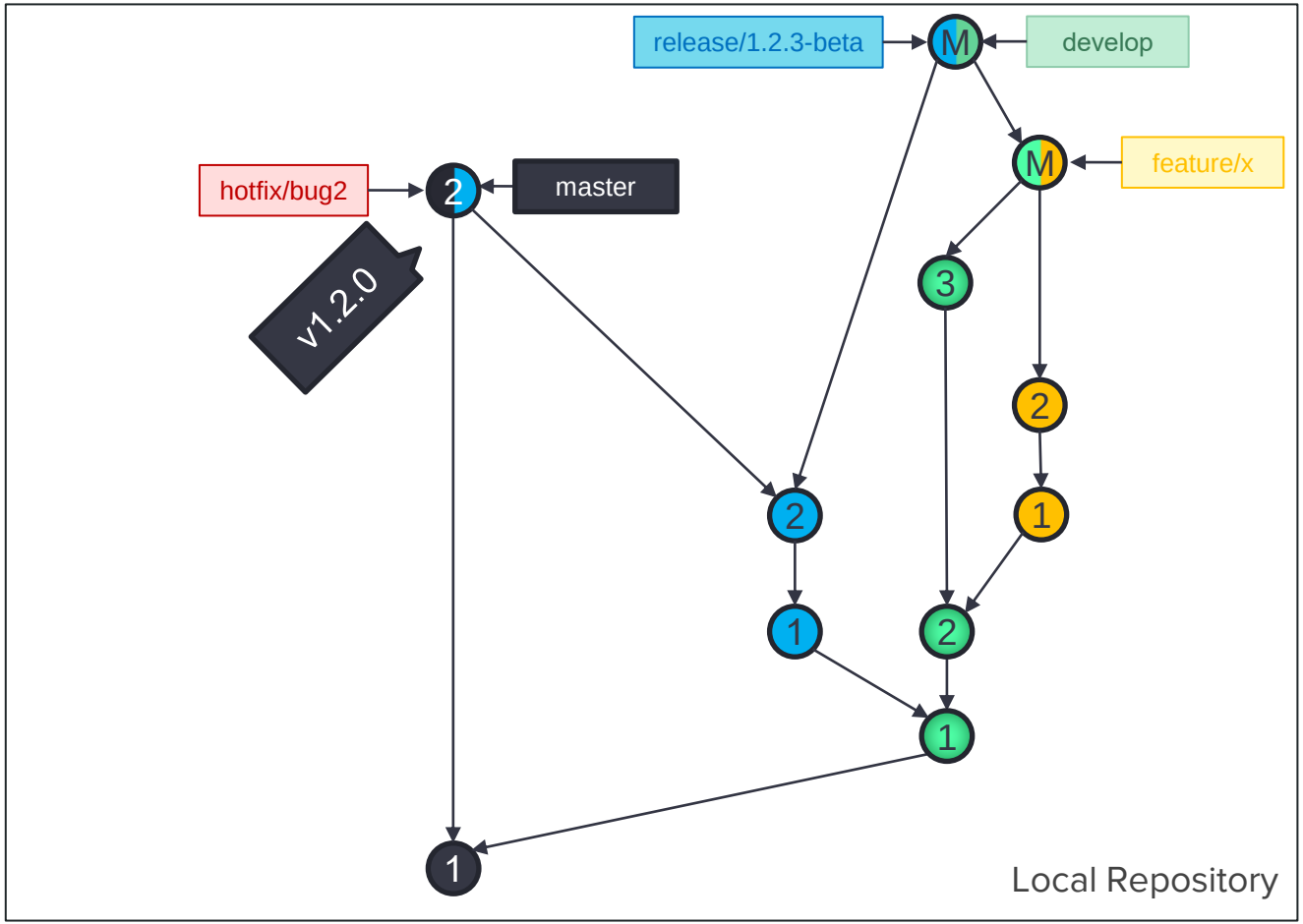
Git Flow

Fast-forward develop



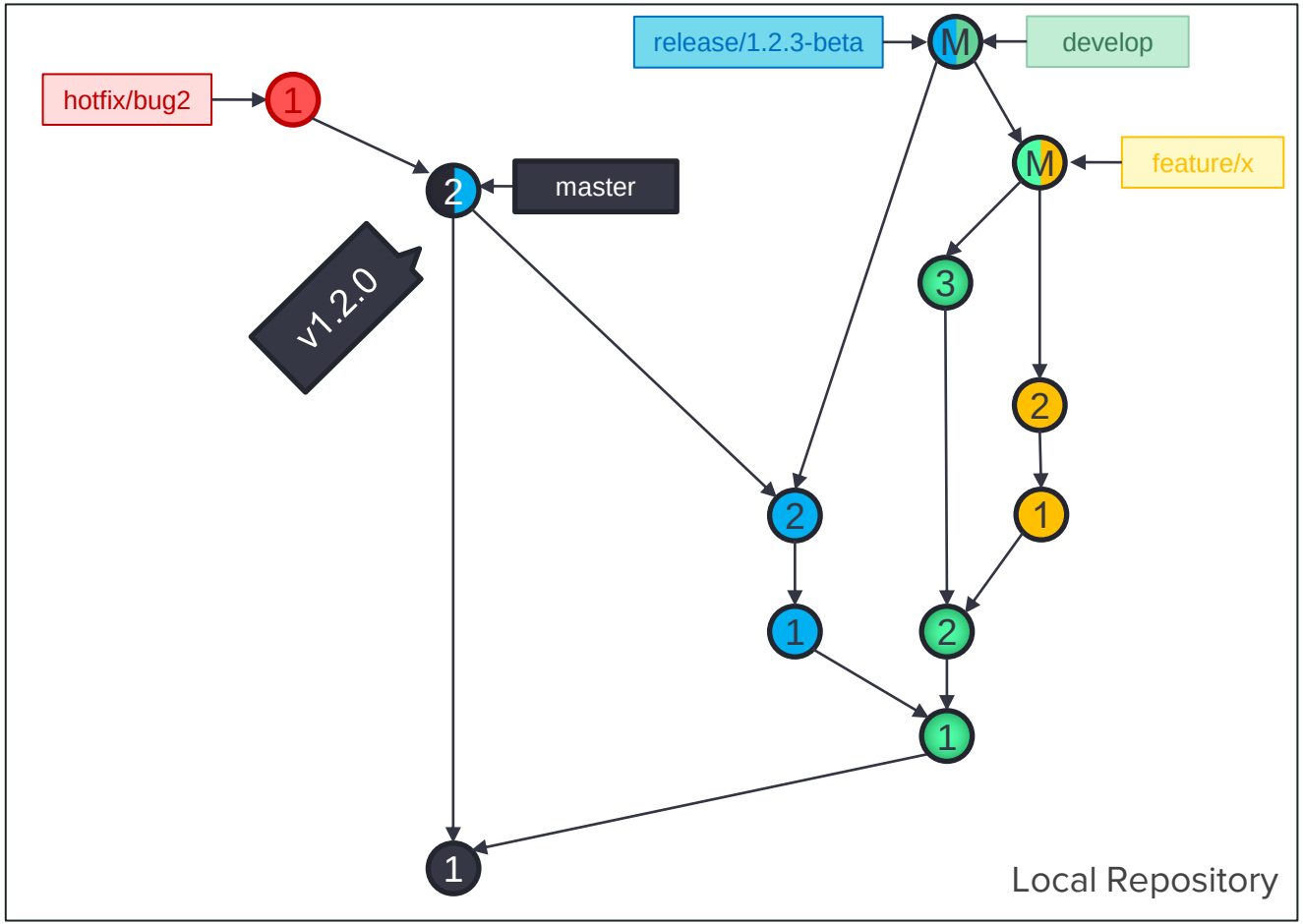
Git Flow

hotfix/x == developer



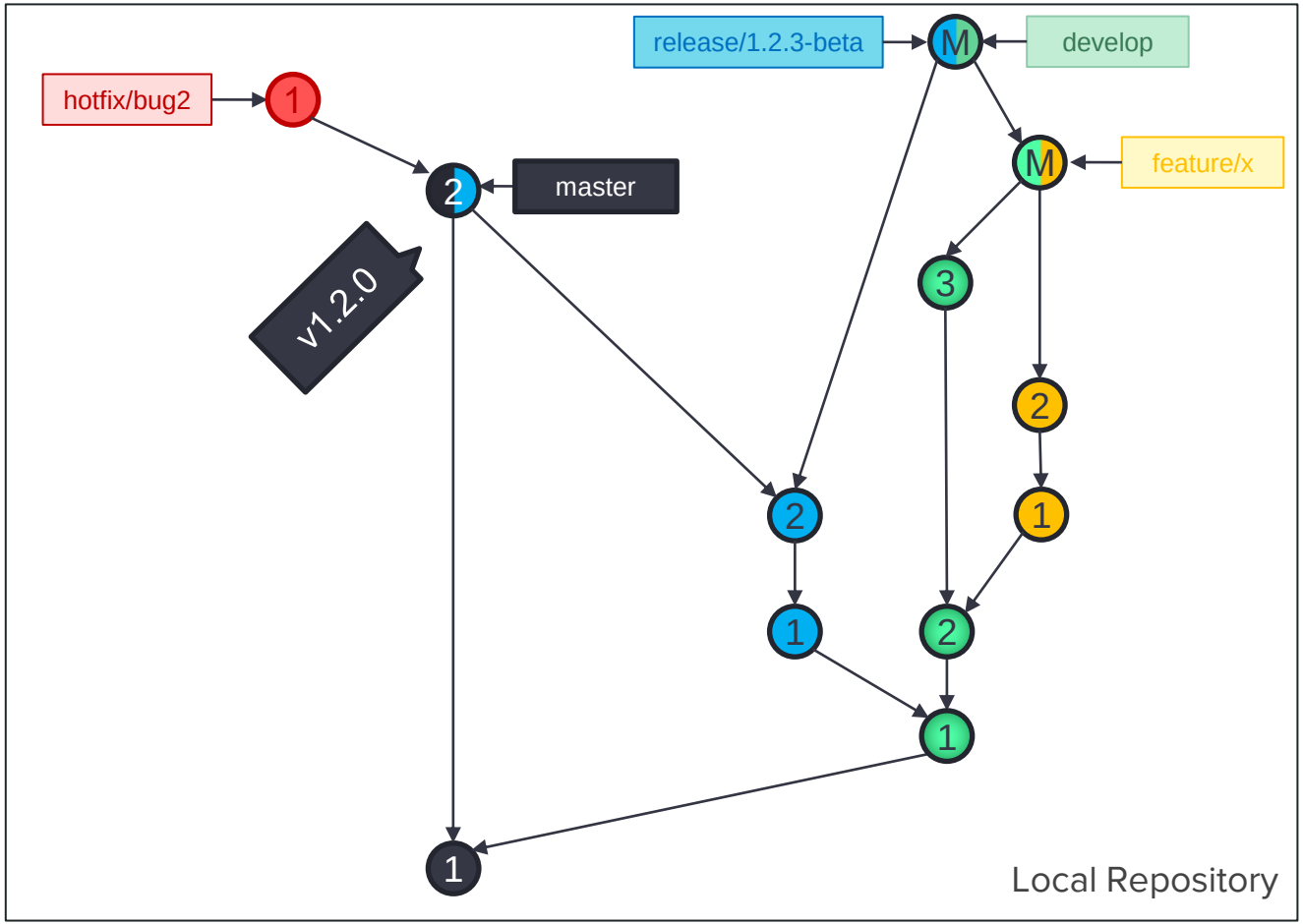
Git Flow

hotfix/x == developer



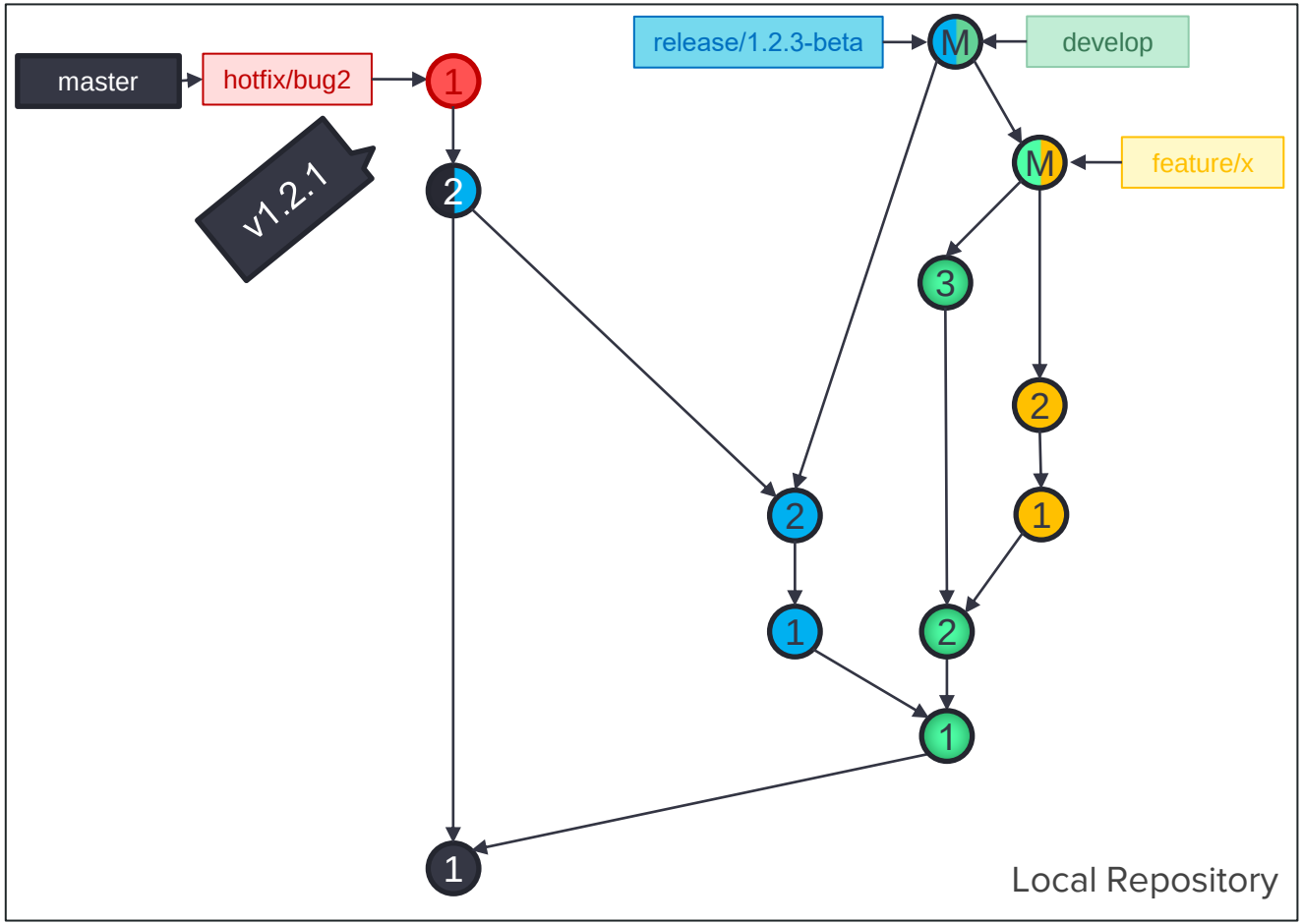
Git Flow

Fast-forward master



Git Flow

Fast-forward master



Git Flow

```
Merge hotfix into develop
```

Git Flow

```
Merge hotfix into develop
```

```
Delete hotfix branch
```

Git Flow

```
Merge hotfix into develop
```

```
Delete hotfix branch
```

```
Delete release branch
```

Git Flow

```
Merge hotfix into develop
```

```
Delete hotfix branch
```

```
Delete release branch
```

```
Delete feature branch
```

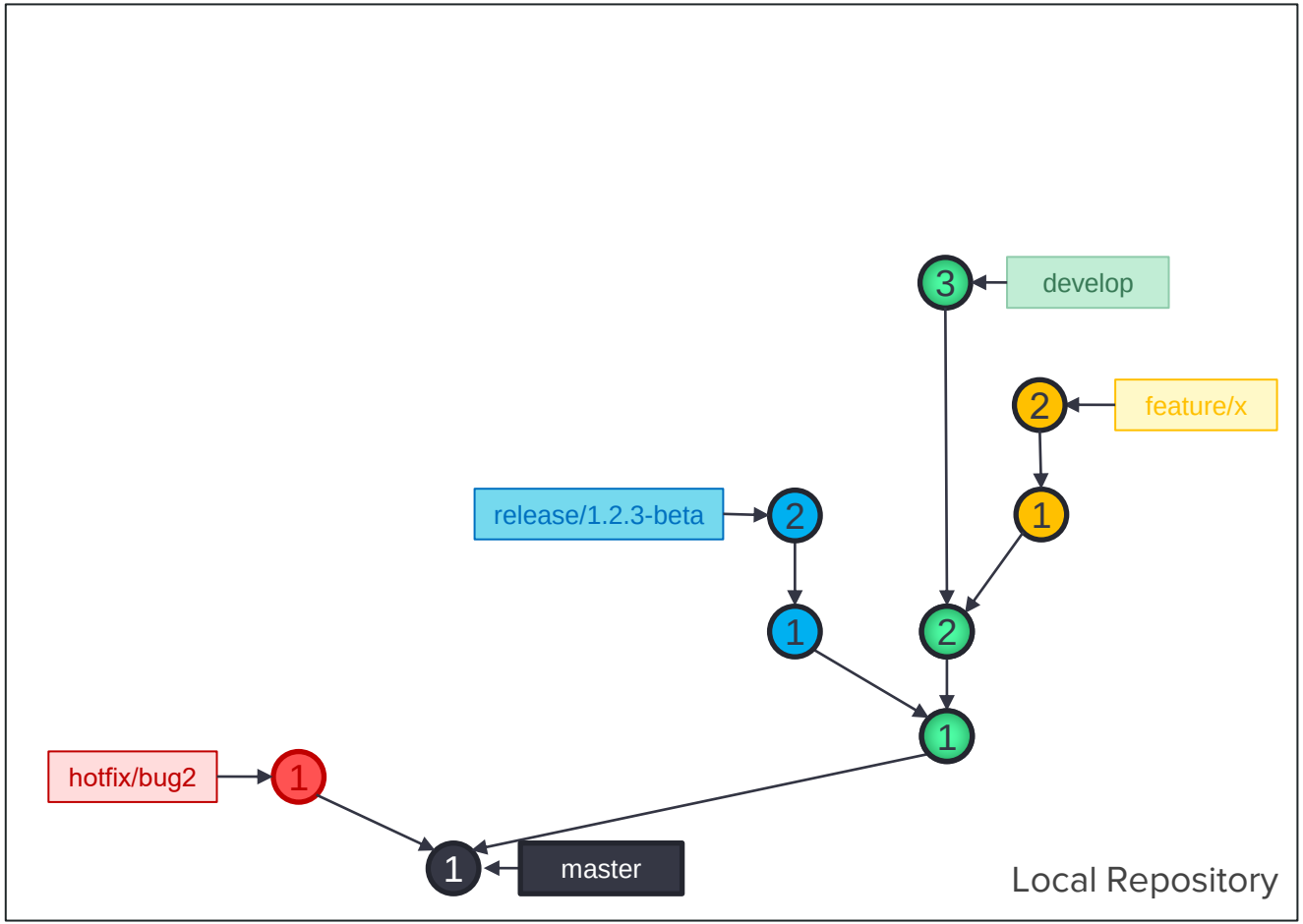
Tips and Tricks

Branching Models
Keeping Branches Linear

Linear Branches

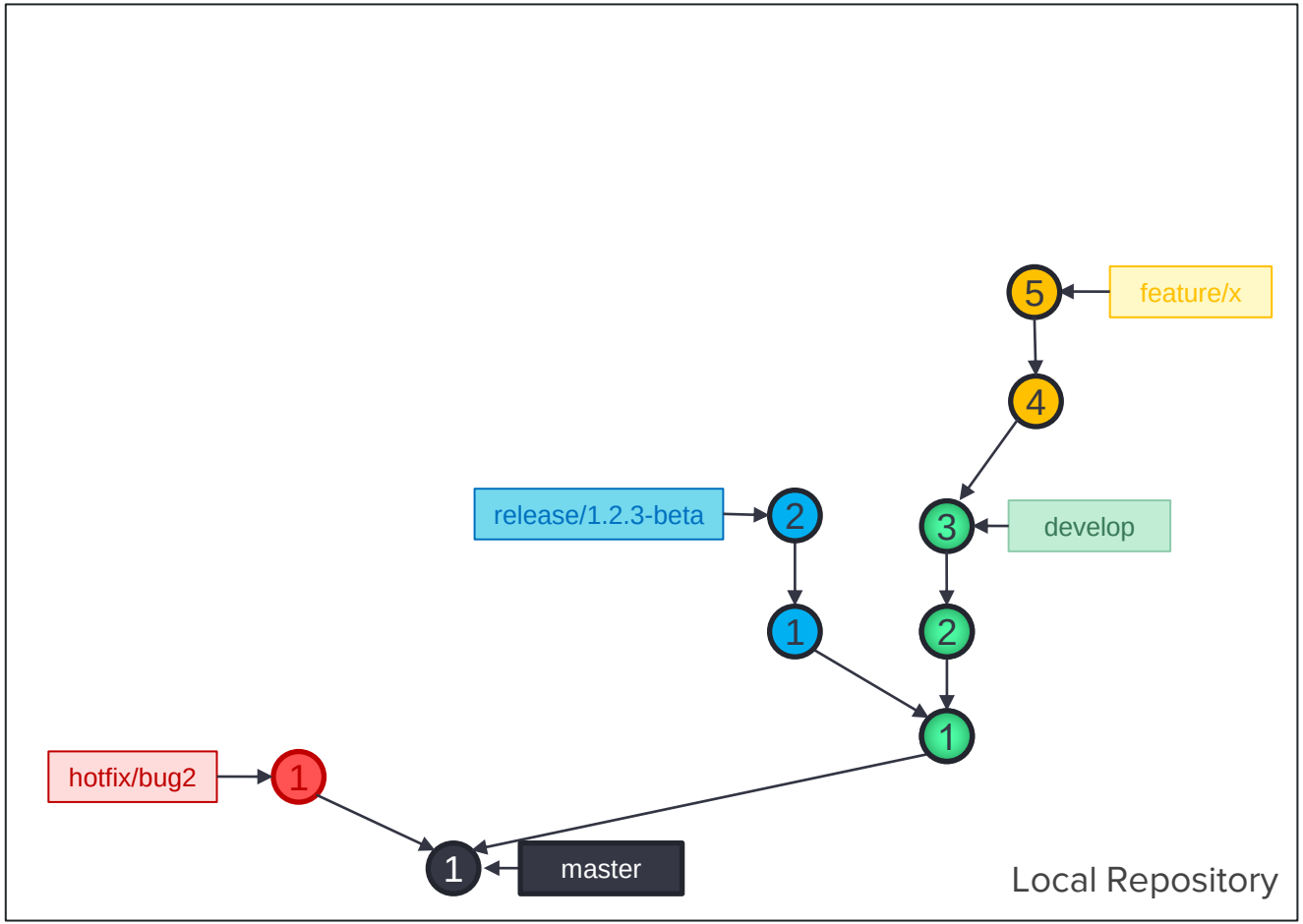
Linear Branches

```
Rebase feature onto develop
```



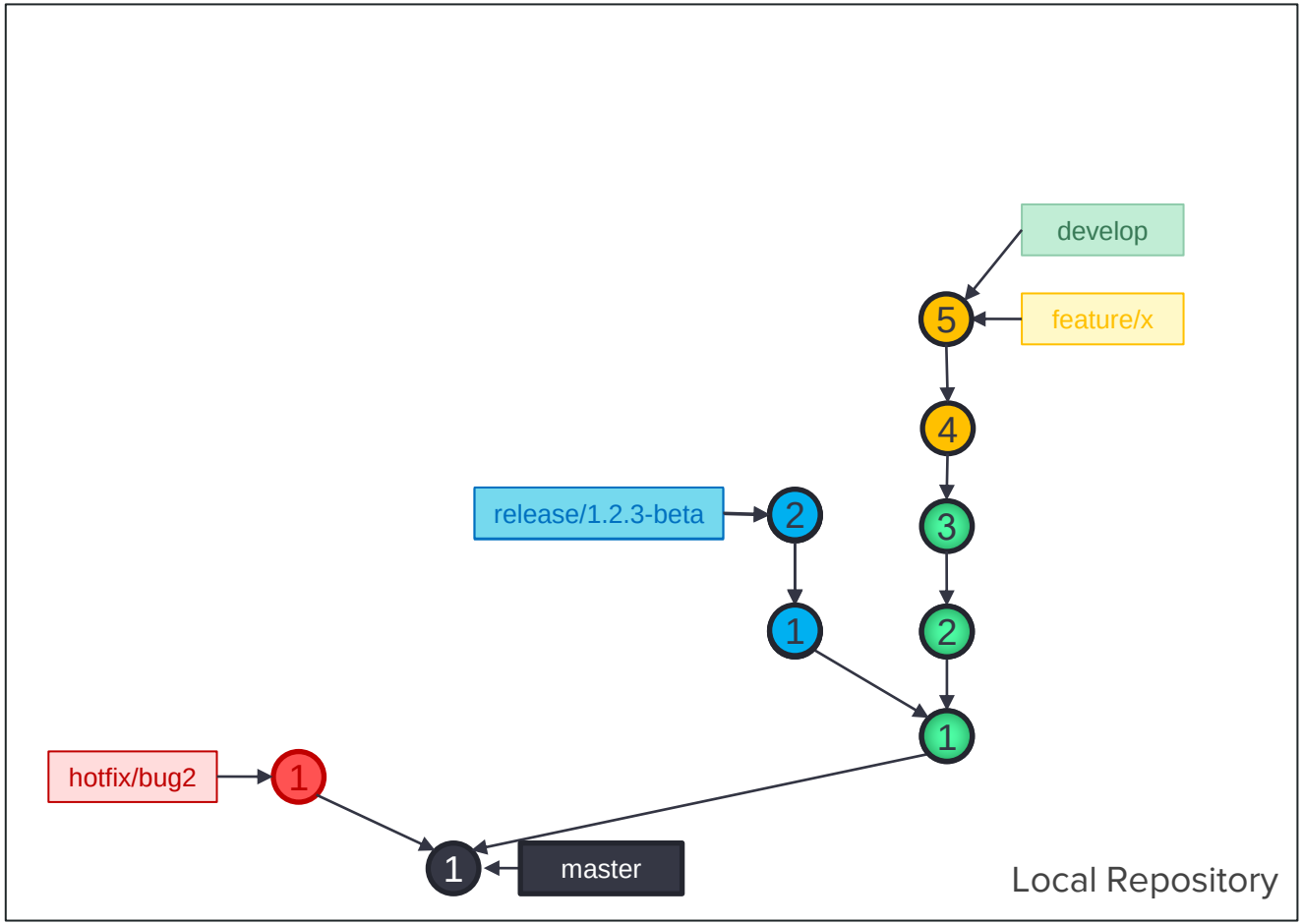
Linear Branches

```
Rebase feature onto develop
```



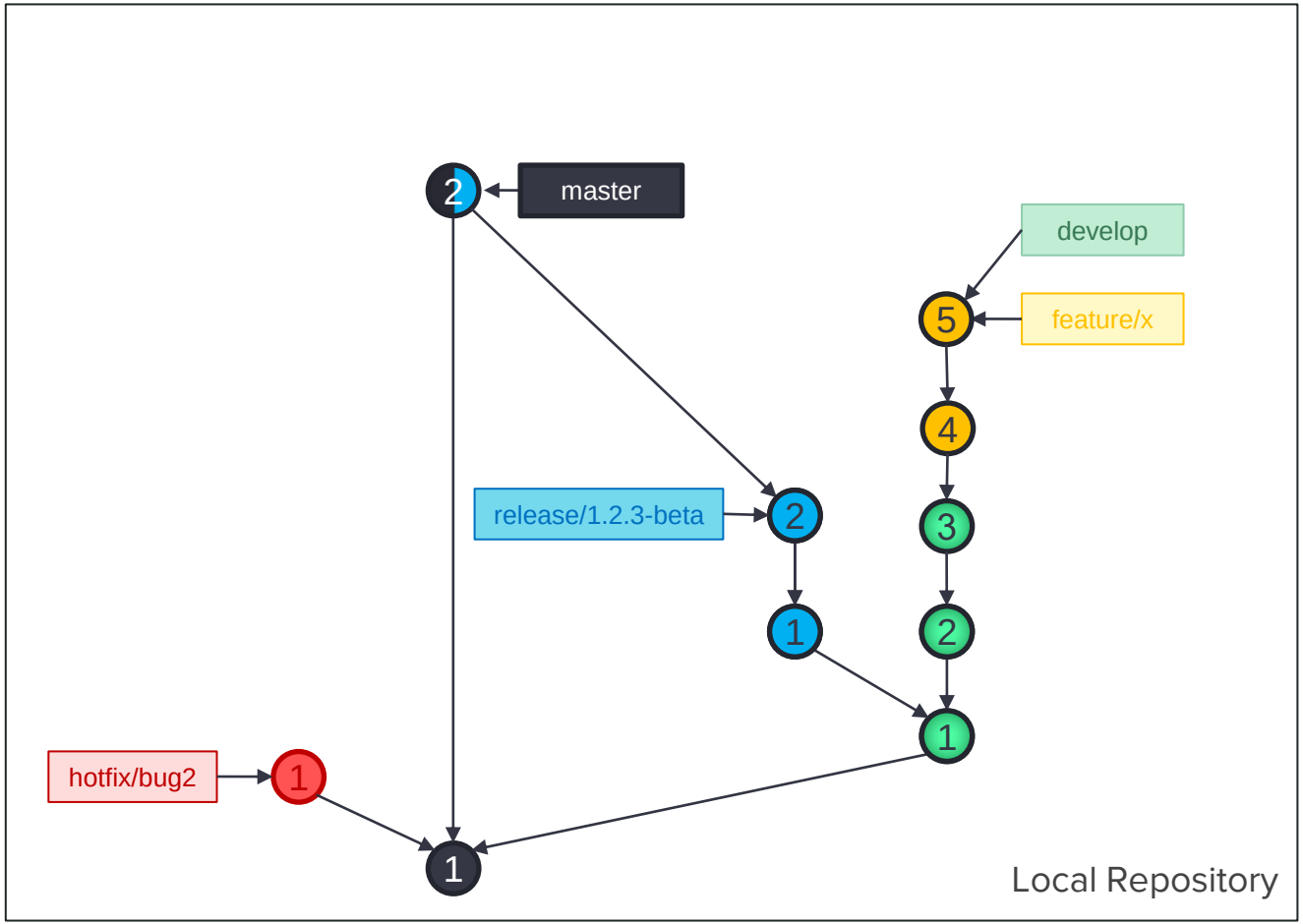
Linear Branches

Fast-forward develop



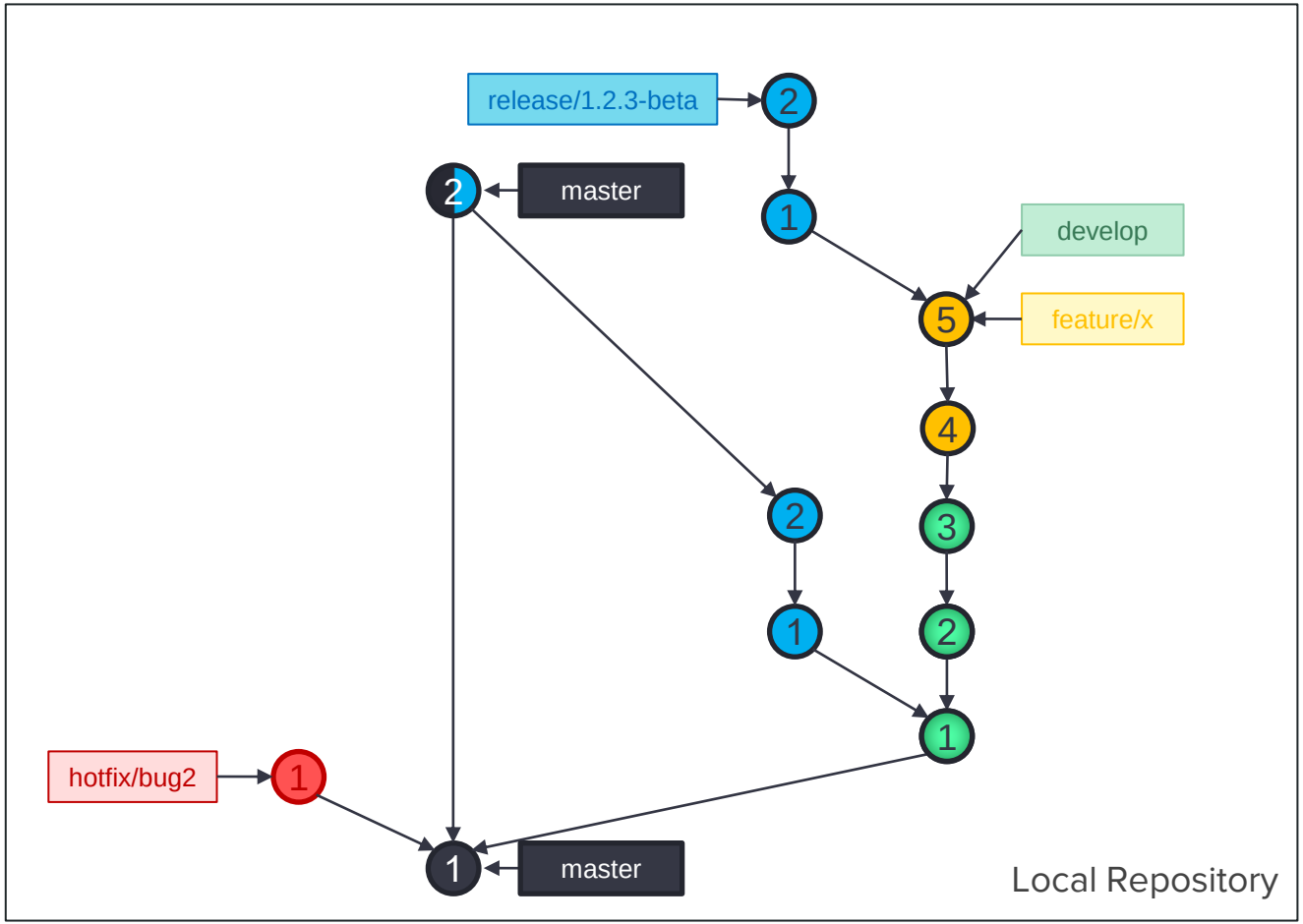
Linear Branches

```
Merge release into
master
```



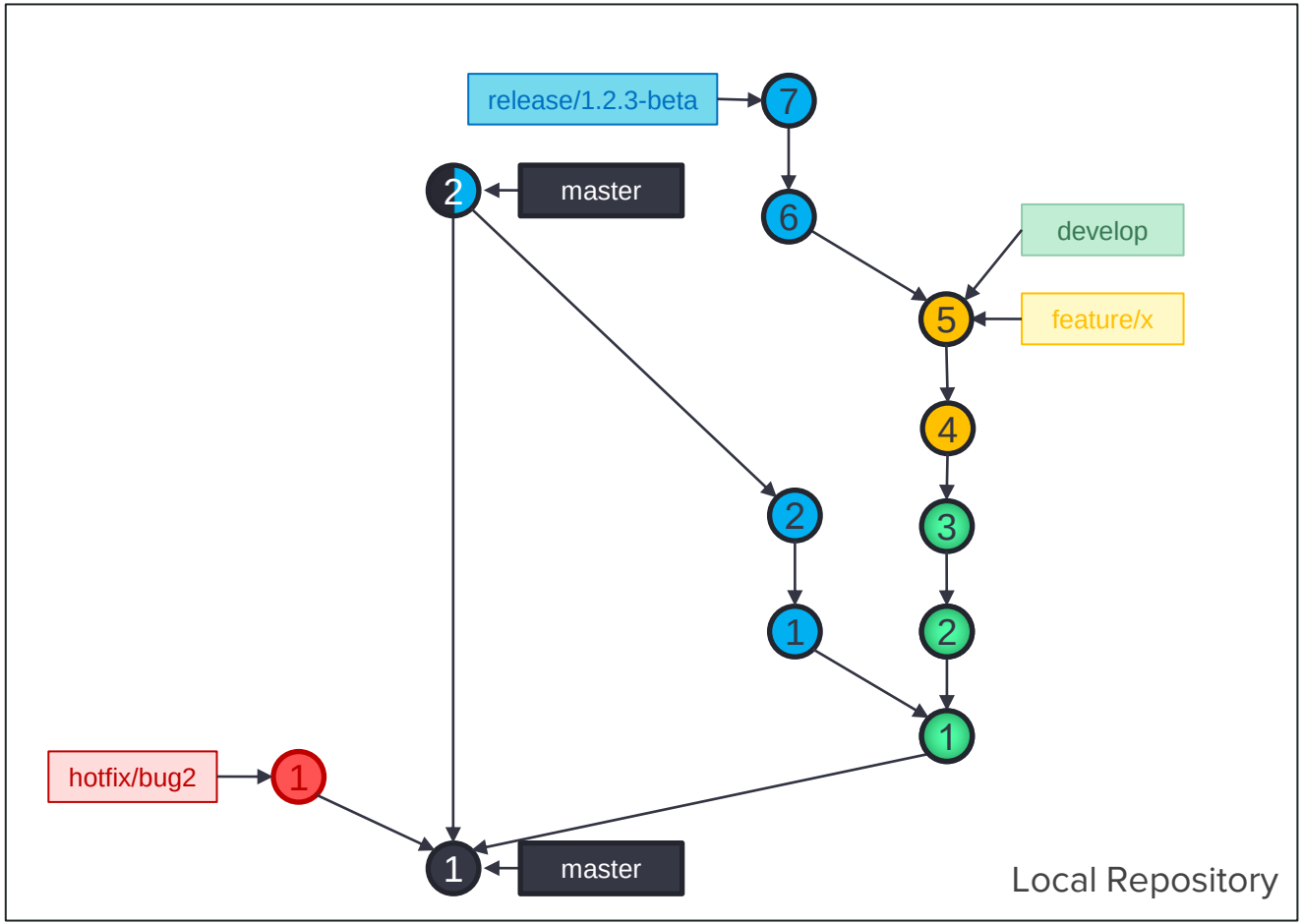
Linear Branches

Rebase release onto develop



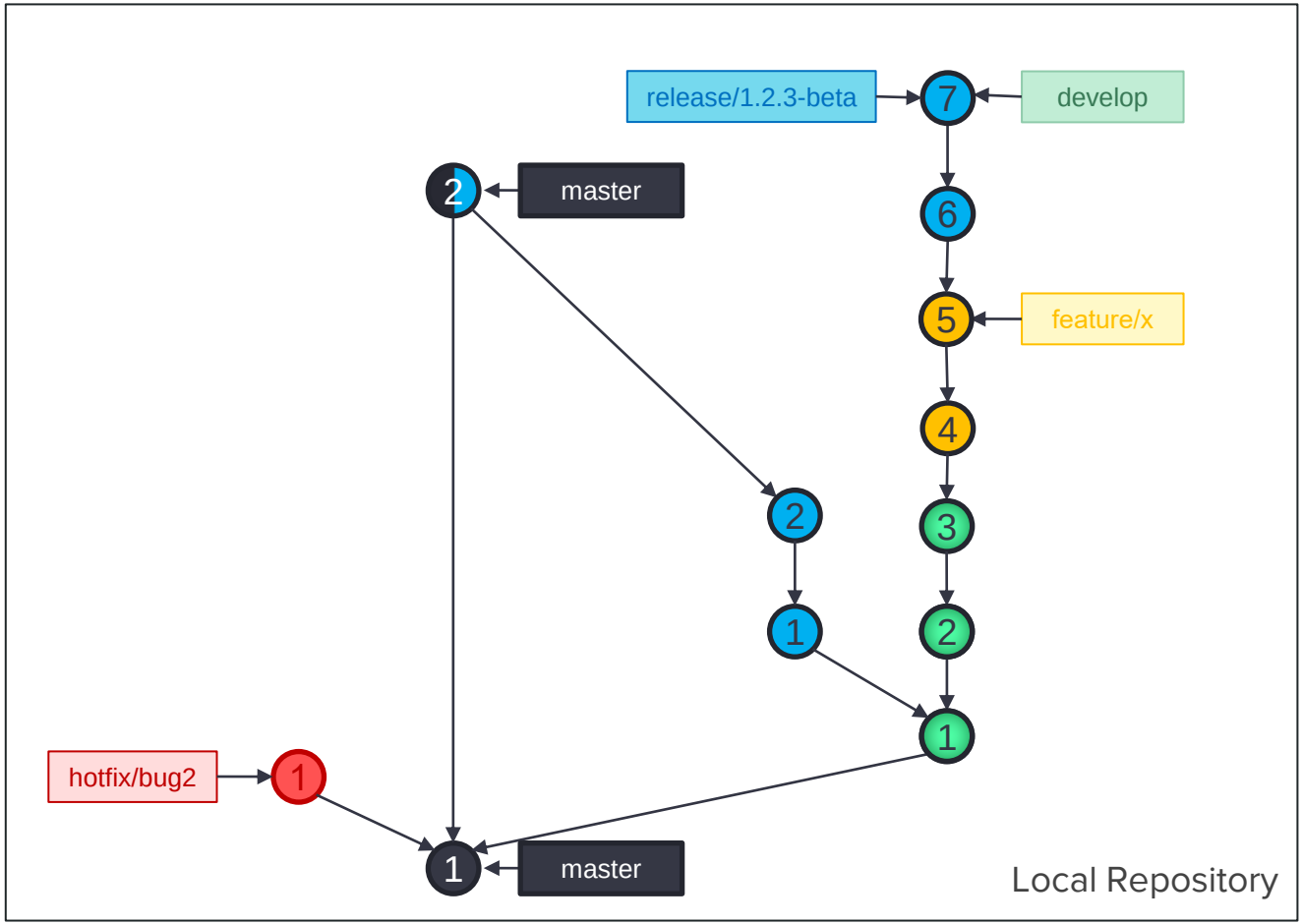
Linear Branches

Rebase release onto develop



Linear Branches

Fast-forward develop



Linear Branches - Rebasing

- Benefits
  - Develop is very clear and linear, no distracting branches

- The tradeoffs are:
  - No more overview of when a feature was introduced into the timeline
  - No easy way of removing a feature (have to hunt down all commits pertaining to feature and remove them one by one)

You thought I wanted to turn you away from the dark side?

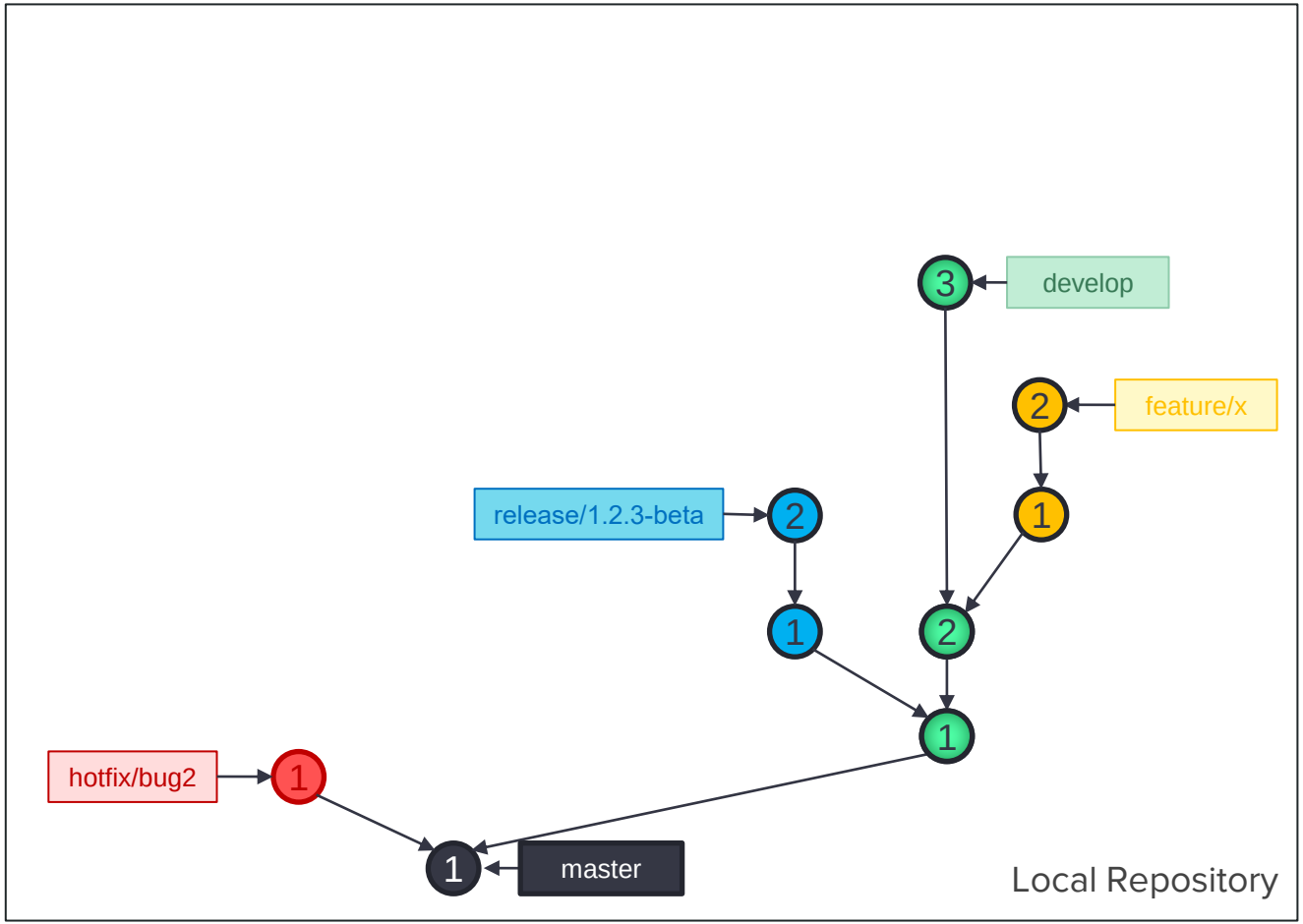
Use it to your advantage … embrace the dark side …

Tips and Tricks

Branching Models
Improving Linear Branches

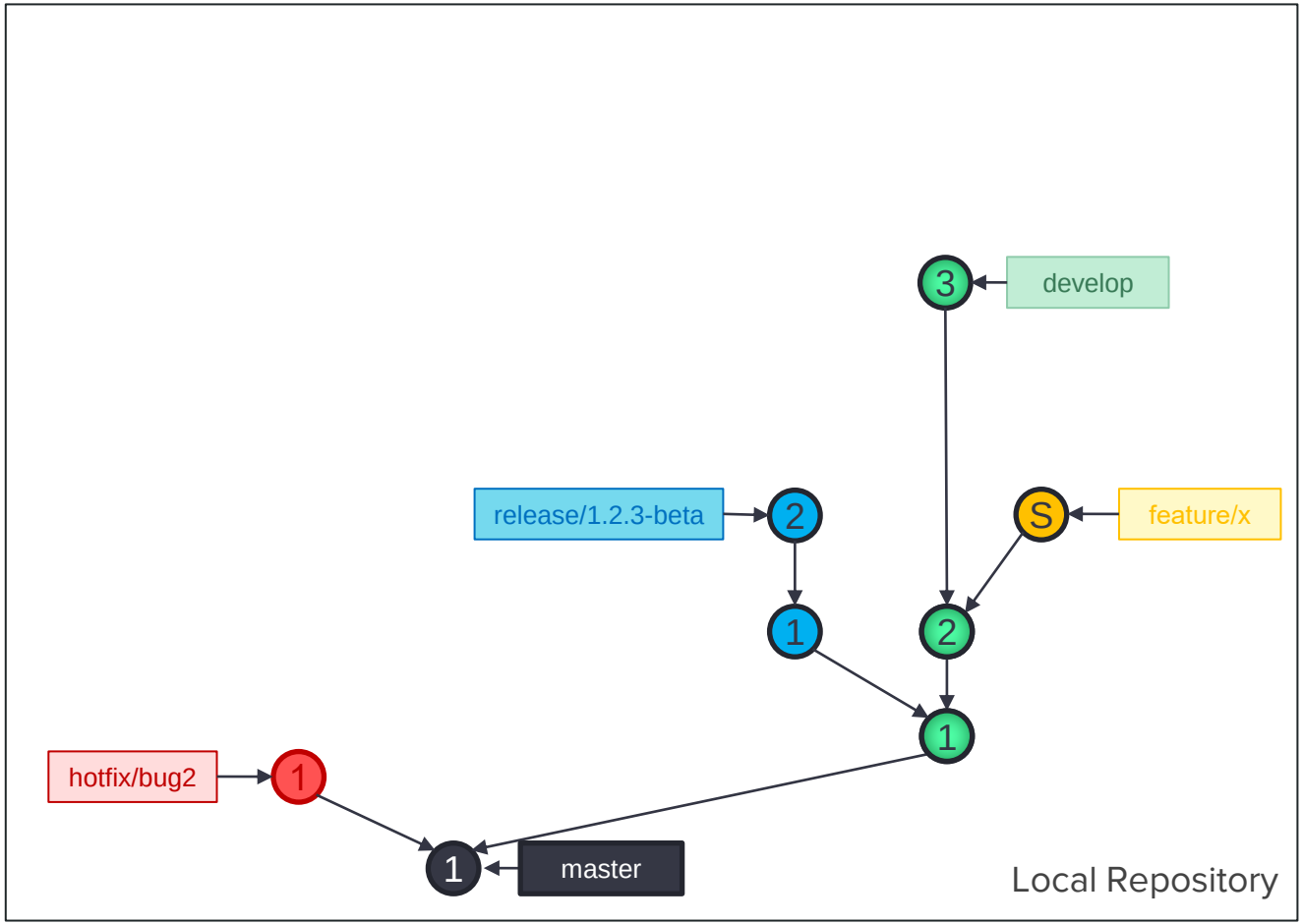
Linear Improvement

Squash & Rebase feature onto develop



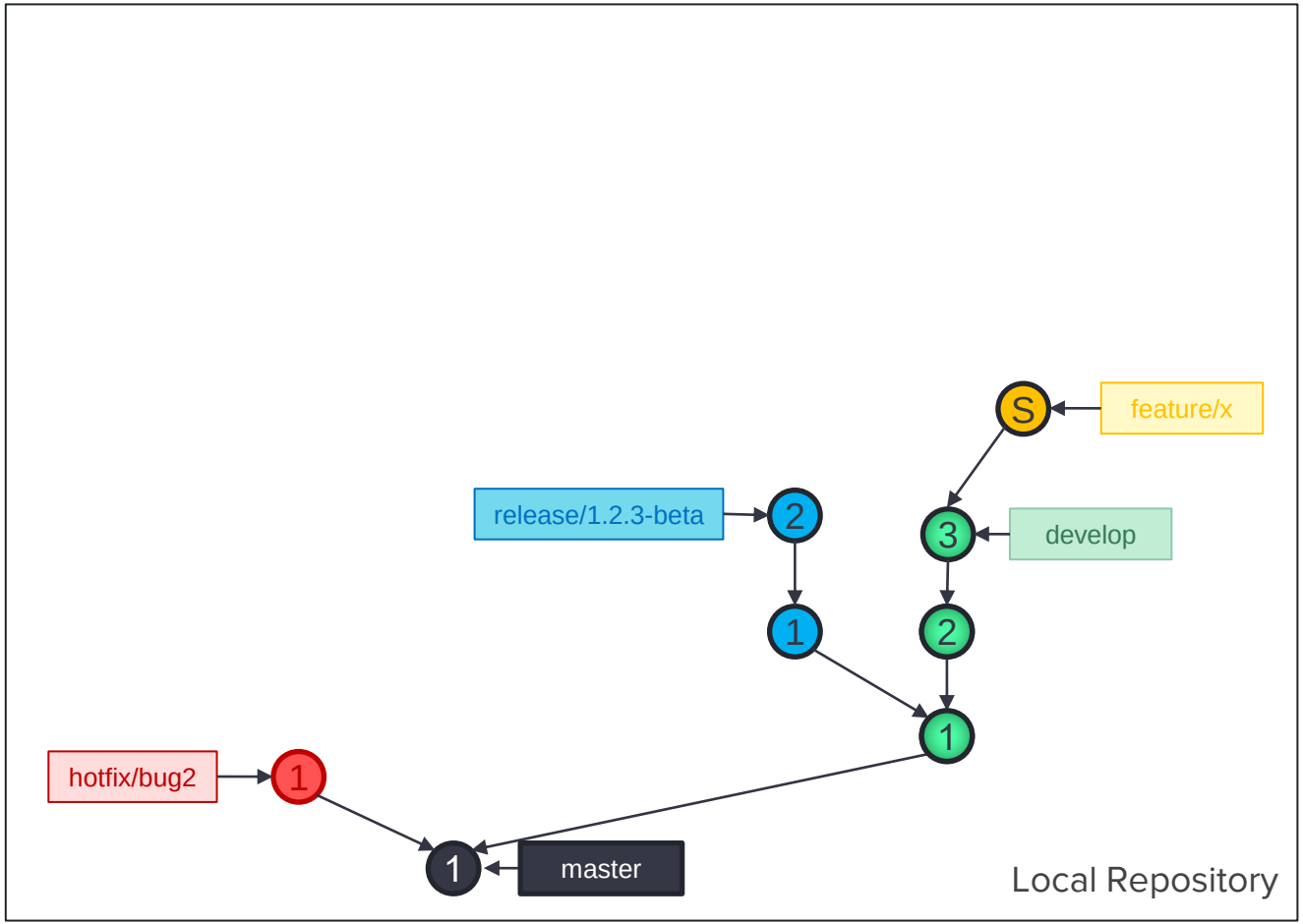
Linear Improvement

Squash & Rebase **feature** onto **develop**



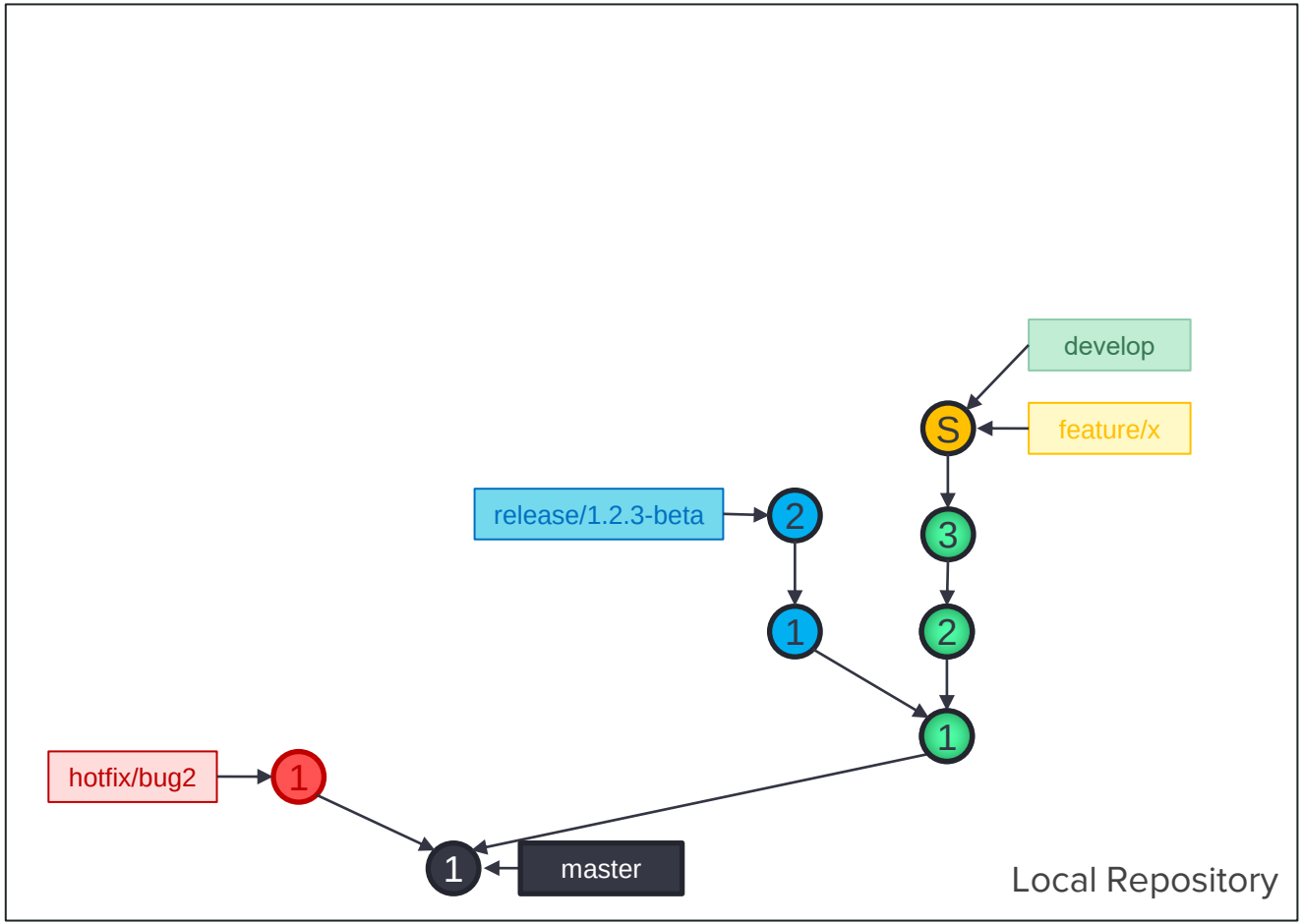
Linear Improvement

Squash & Rebase **feature** onto **develop**



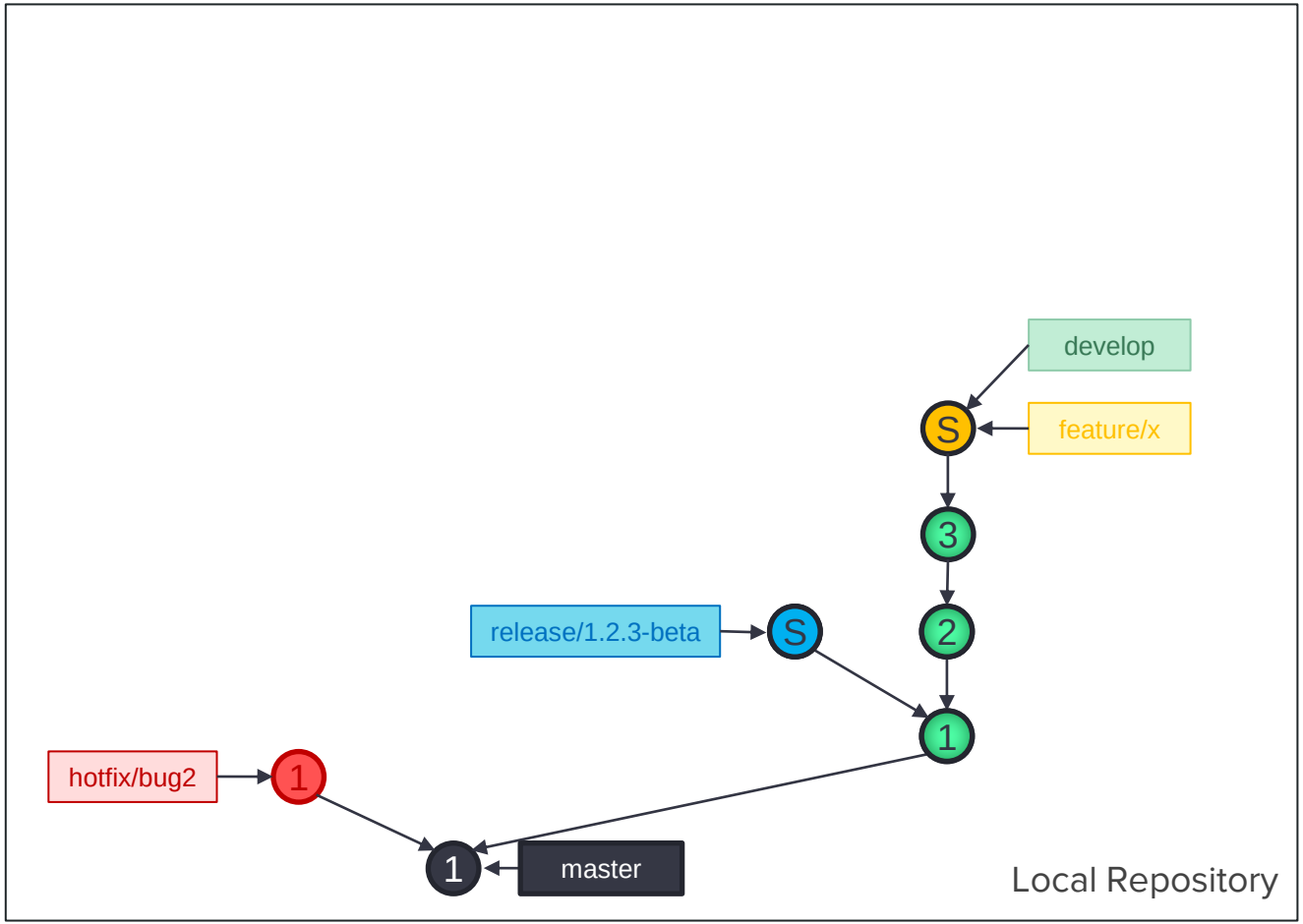
Linear Improvement

```
Fast-forward develop
```



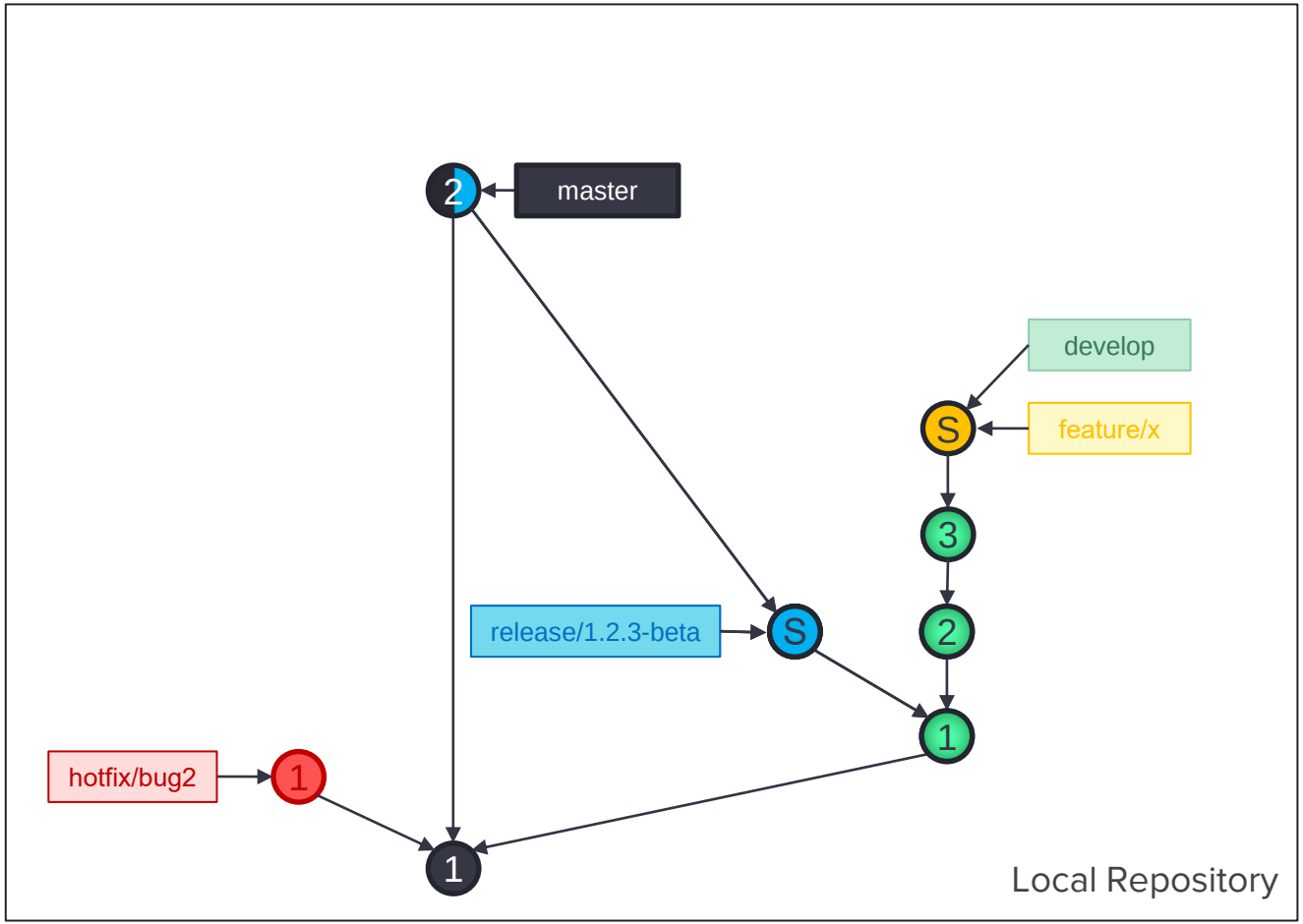
Linear Improvement

Squash release



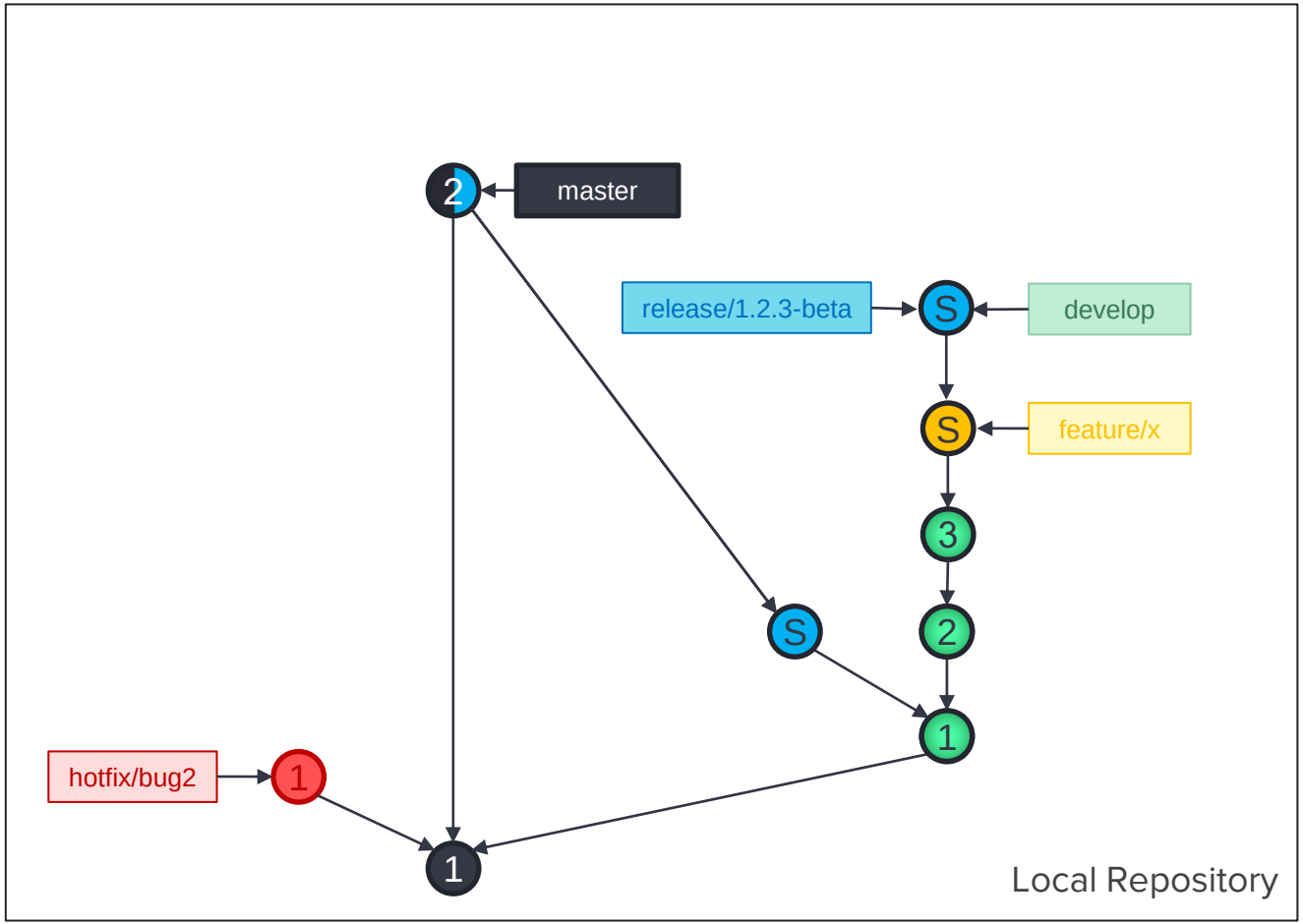
Linear Improvement

```
Merge release into master
```



Linear Improvement

Squash and rebase
release onto develop



Linear Improvement

- Benefits
  - Develop is very clear and linear, no distracting branches
  - Each feature has own commit (atomic features)
    - There is traceability of when a feature was introduced
    - Features are easy to remove
- The tradeoffs are:
  - No more history of individual feature development

Recommendations

- Always merge **the branch you diverged from** into your branch first **and test**, not the other way around!
- Name your feature commits (the squashes) in a relevant way (Feature Jira ID and Feature Name for example)
- Release Branches should **only have fixes**, all other code should be frozen
- **Never merge master back into develop**

Recommendations - Continued

- Always **merge** releases into master, have that merge commit there!

- Don’t squash develop, **ever**
- Don’t rebase develop, **ever**

This also means: never squash releases and features beyond their first divergent commit

- Always **merge** hotfixes into master and develop

Recommendations - Continued

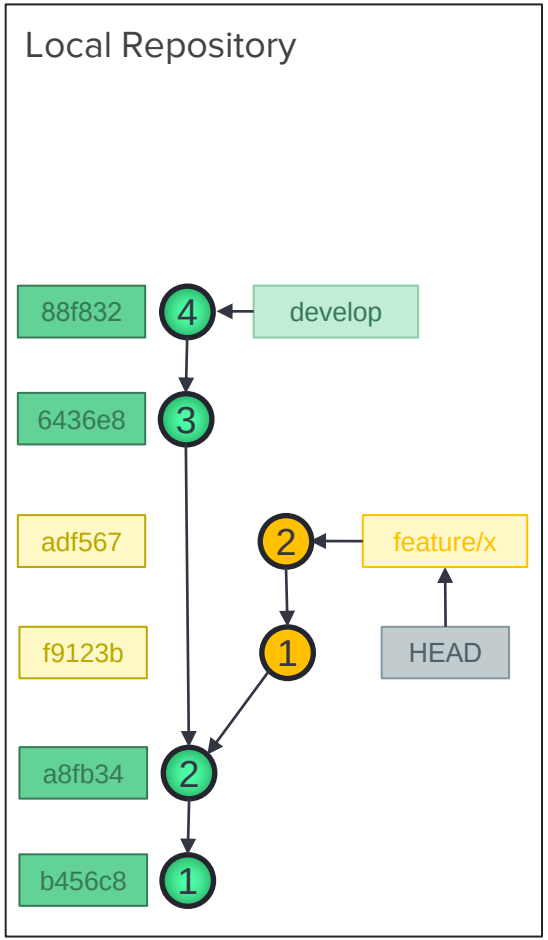
- **Always tag** a release merge commit in master, same for hotfix
- **Commit very often**, as often as you'd hit "quick save" in a game
- Push your work **at least once per day**, yes, including feature branches
- Merge develop into your feature branch **at least once per day** (yes you can merge because later you will squash anyway)
- `git pull --rebase` develop **at least once per day**, you can also `git pull --ff-only` and do a rebase only if this fails

Tips and Tricks

Easy Squash

Easy Squash

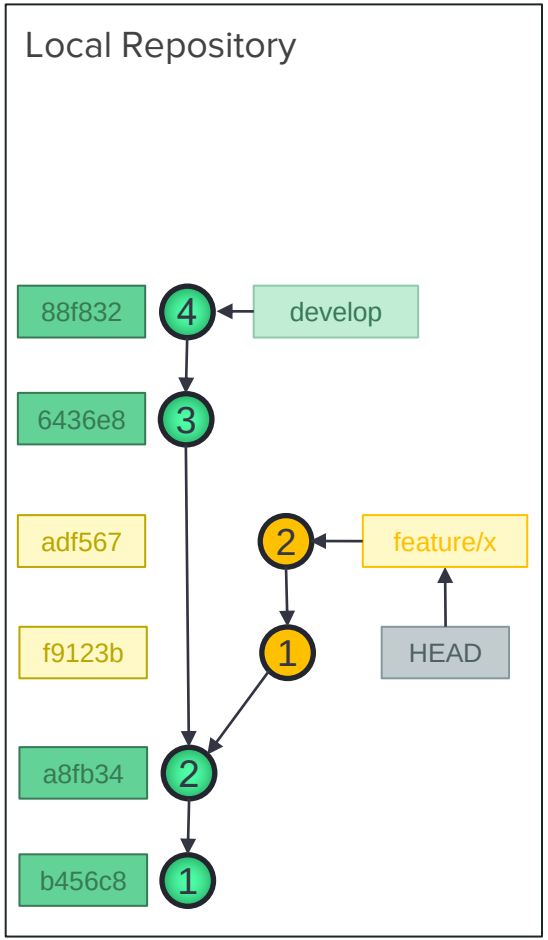
```
$ |
```


Local Repository
88f832
4
develop
6436e8
3
adf567
2
feature/x
f9123b
1
HEAD
a8fb34
2
b456c8
1


Local Repository
1
2
Staging Area
1
2
Working Directory
1
2

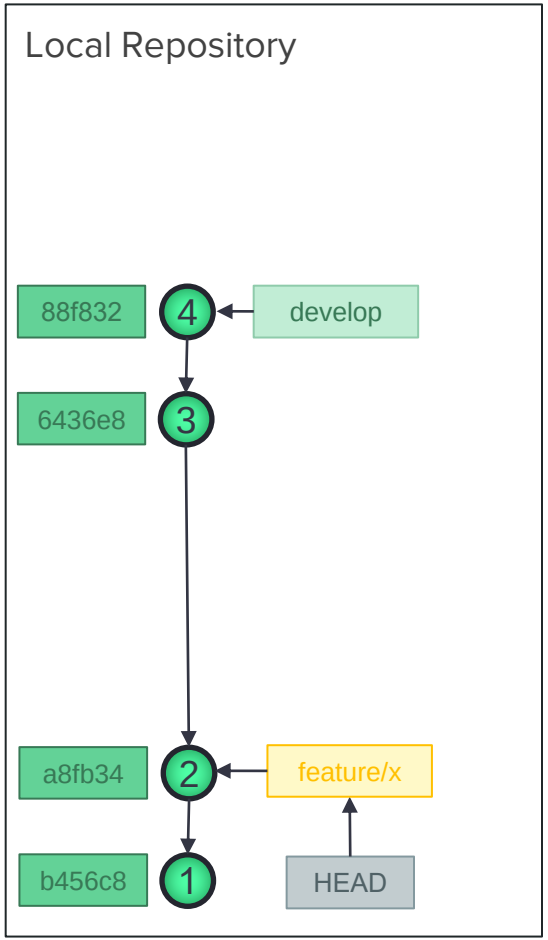
Easy Squash

```
$ git reset --soft
  HEAD~3

$ |
```



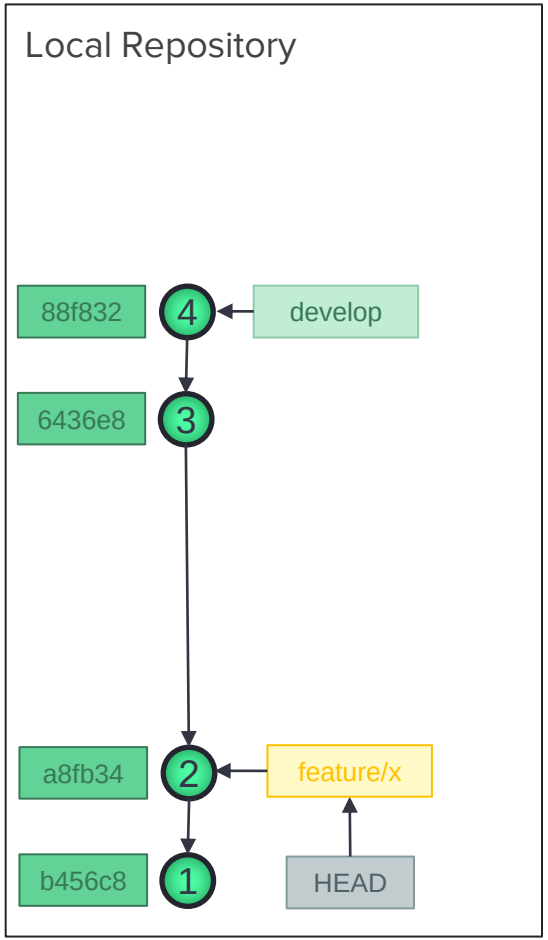
Easy Squash

```
$ git reset --soft
  HEAD~3

$ |
```



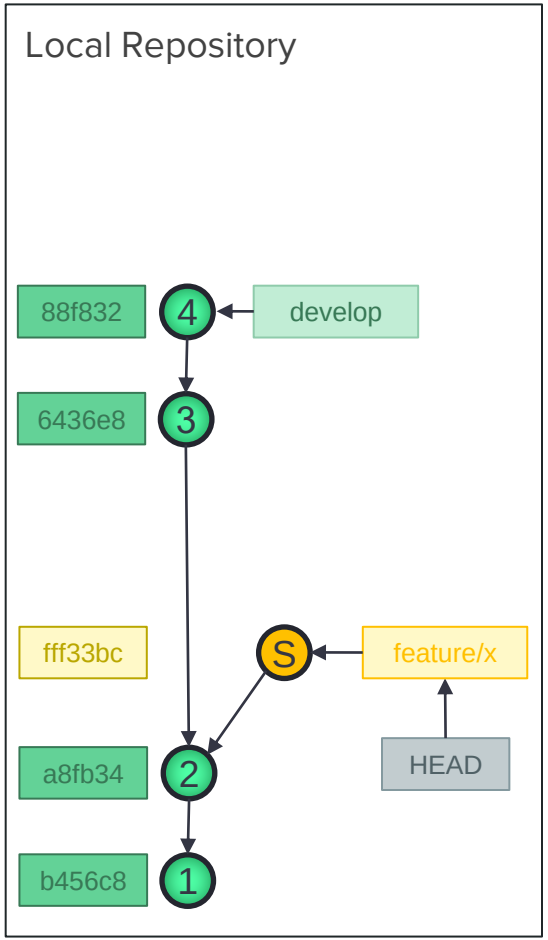
Easy Squash

```
$ git reset --soft
  HEAD~3

$ git commit

$ |
```



Easy Squash

```
$ git reset --soft
  HEAD~3

$ git commit

$ |
```



Recommendations

- Easy squash avoids the need to edit that interactive rebase file
- Always squash dummy commits (with comments like: “forgot a file”, “another change”, “small fix”, “fixed build”, “forgot a file”, stuff like that that that always pops up)

Tips and Tricks

Use the repository hosting service, Luke!

Use pull requests

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: master ← compare: develop ✓ Able to merge. These branches can be automatically merged.

1 develop

Write Preview

2 develop

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

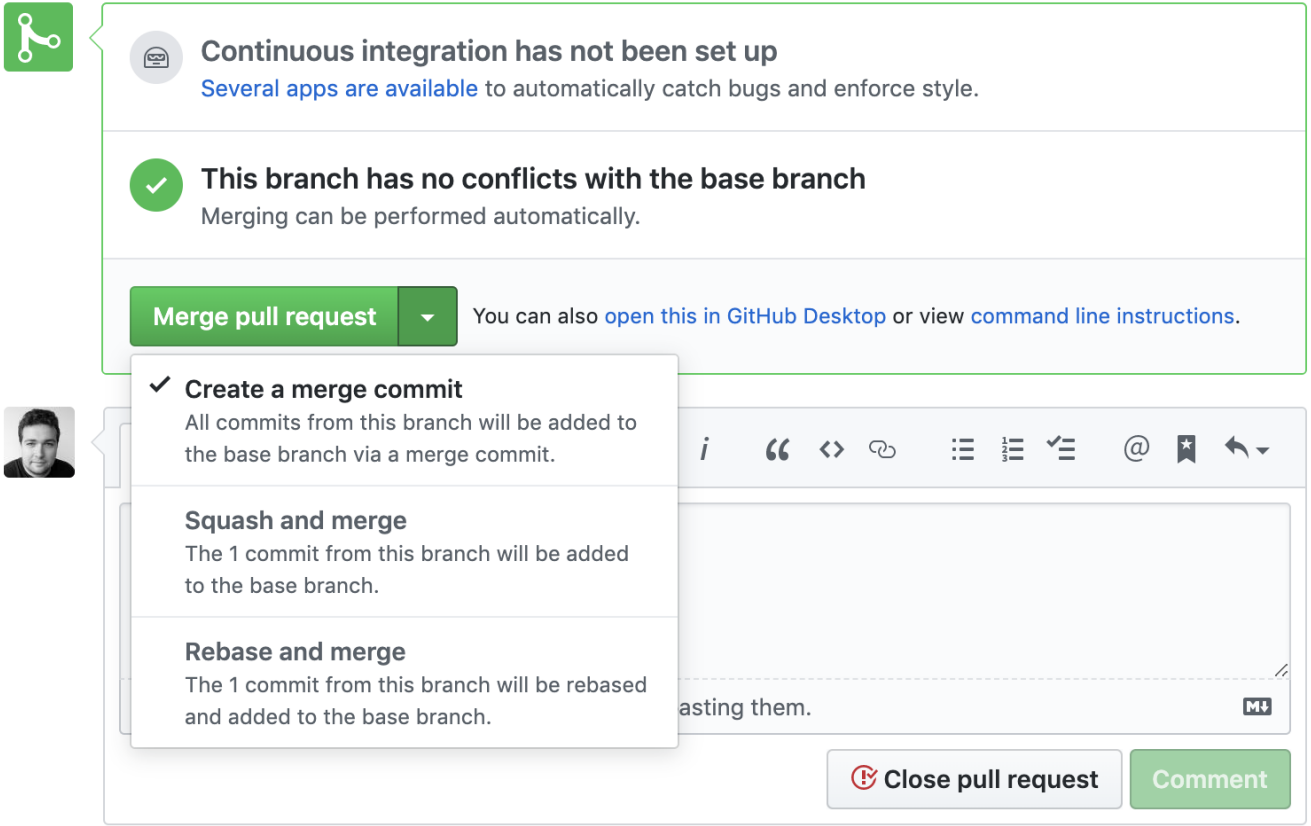
Projects

None yet

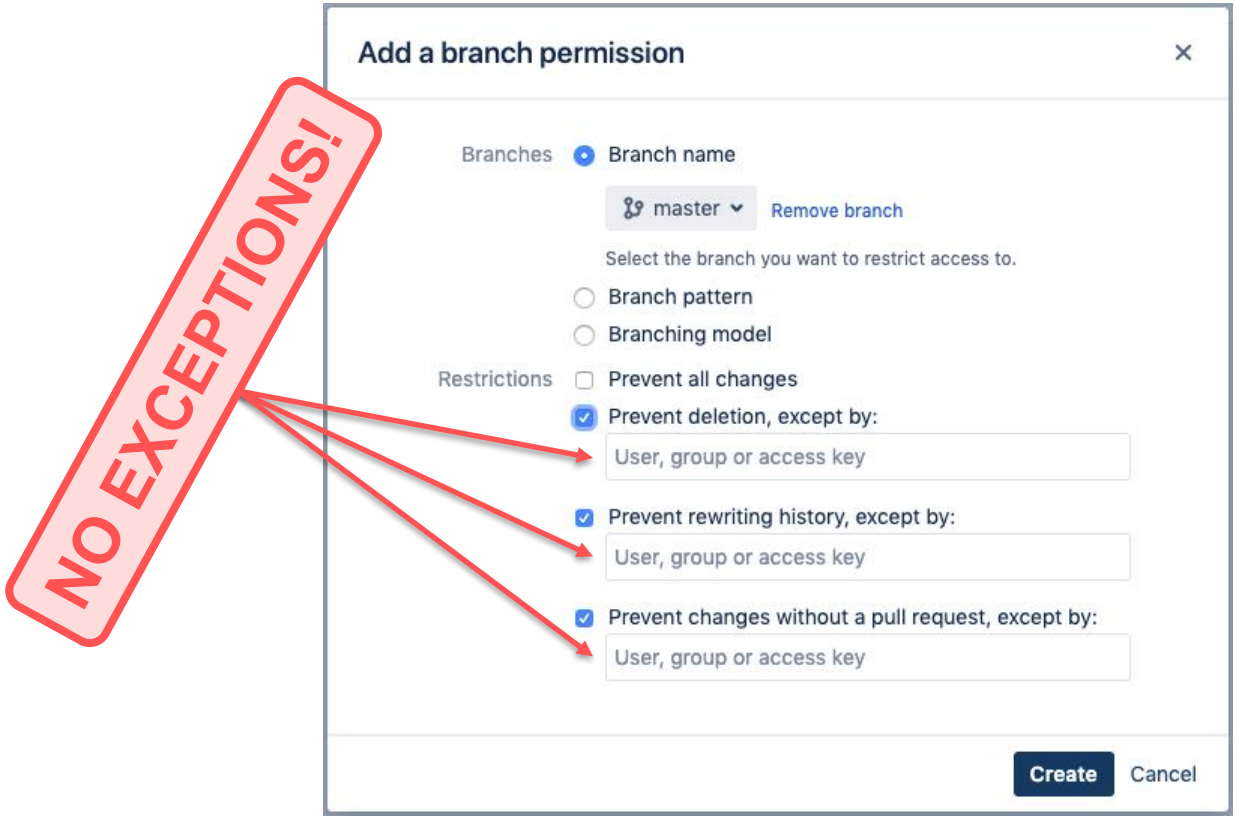
Milestone

No milestone

Advanced Merging Options



Protect Branches - Bitbucket



Protect Branches - Bitbucket

Repository settings

Repository details

SECURITY

Repository permissions

Branch permissions

Access keys

Audit log

WORKFLOW

Branching model

Hooks

Webhooks

Hooks

Add hook

Hooks allow you to extend what Bitbucket does every time a repository changes, for example when code is pushed or a pull request is merged. Hooks are installed by system administrators and can be enabled for all repositories in a project by a project administrator (in project settings), or for individual repositories. Learn more about repository hooks.

Pre receive

Pre receive hooks allow you to control which commits go into your repository before pushes are committed or pull requests are merged.

Reject Force Push

Reject all force pushes (git push --force) to this repository

✓ Enabled

Protect Branches – Github Free

NO EXCEPTIONS!

- Options
- Collaborators
- Branches
- Webhooks
- Notifications
- Integrations & services
- Deploy keys

Moderation

- Interaction limits

Branch protection rule

Branch name pattern

develop

Applies to **0** branches

Rule settings

Protect matching branches
Disables force-pushes to all matching branches and prevents them from being deleted.

- [x] **Require pull request reviews before merging**
  When enabled, all commits must be made to a non-protected branch and submitted via a pull request with the required number of approving reviews and no changes requested before it can be merged into a branch that matches this rule.

  Required approving reviews: 1

  - [x] **Dismiss stale pull request approvals when new commits are pushed**
    New reviewable commits pushed to a matching branch will dismiss pull request review approvals.
  - [ ] **Require review from Code Owners**
    Require an approved review in pull requests including files with a designated code owner.

- [ ] **Require status checks to pass before merging**
  Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

- [ ] **Require signed commits**
  Commits pushed to matching branches must have verified signatures.

- [x] **Include administrators**
  Enforce all configured restrictions for administrators.

Tips and Tricks

RERERE

RERERE

git rerere is an actual command

It is also a setting you should probably enable after seriously reading up on it!

It means Reuse Recorder Resolution

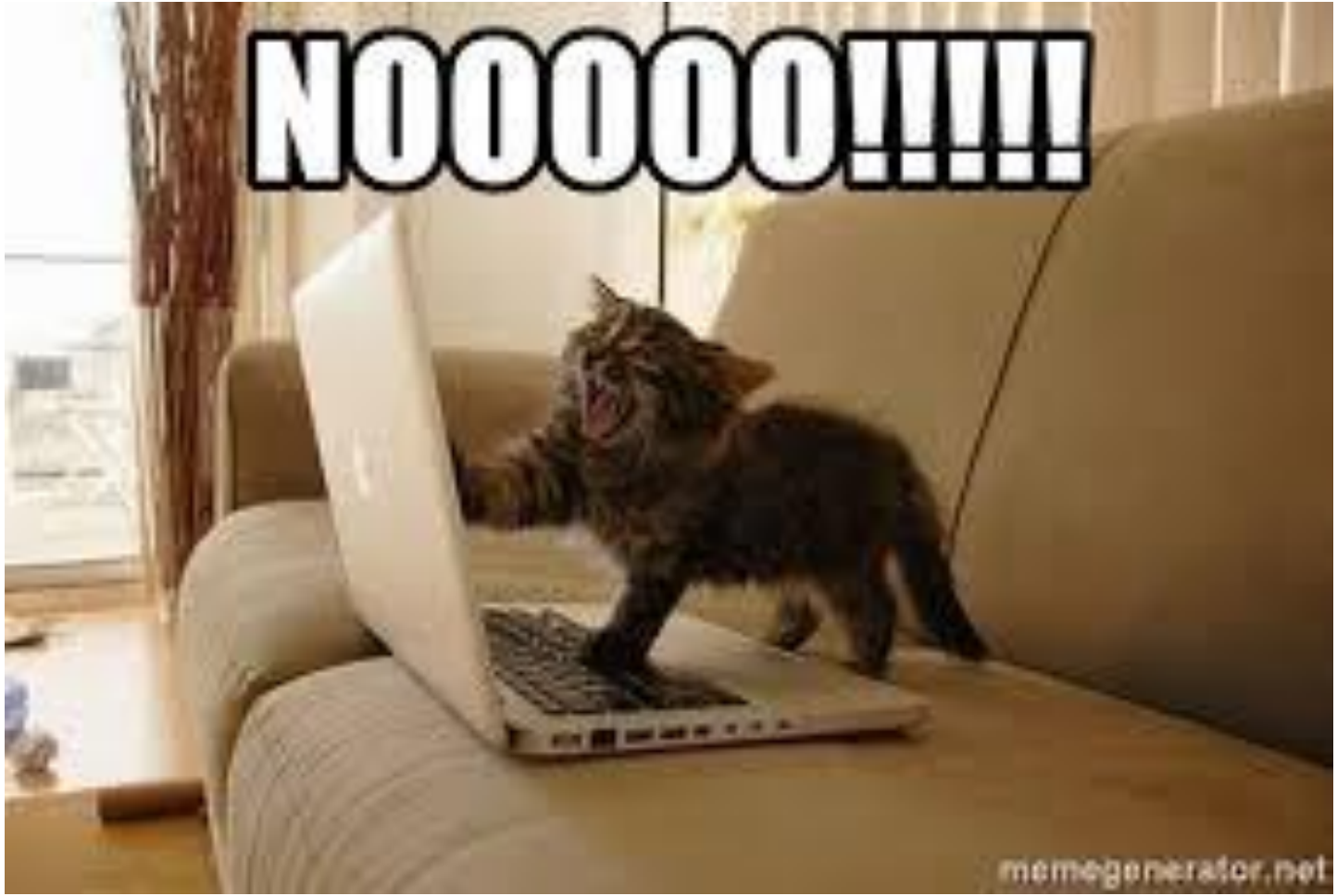
- It automatically records per-file resolutions in case the conflict is encountered again
- It automatically applies recorded resolutions during conflicts
- It allows for test merges before rebases, meaning you can probably only fix conflicts a single time and have a rebase automagically work

Tips and Tricks

FAQ

I just reset/ammended/rebased/squashed …

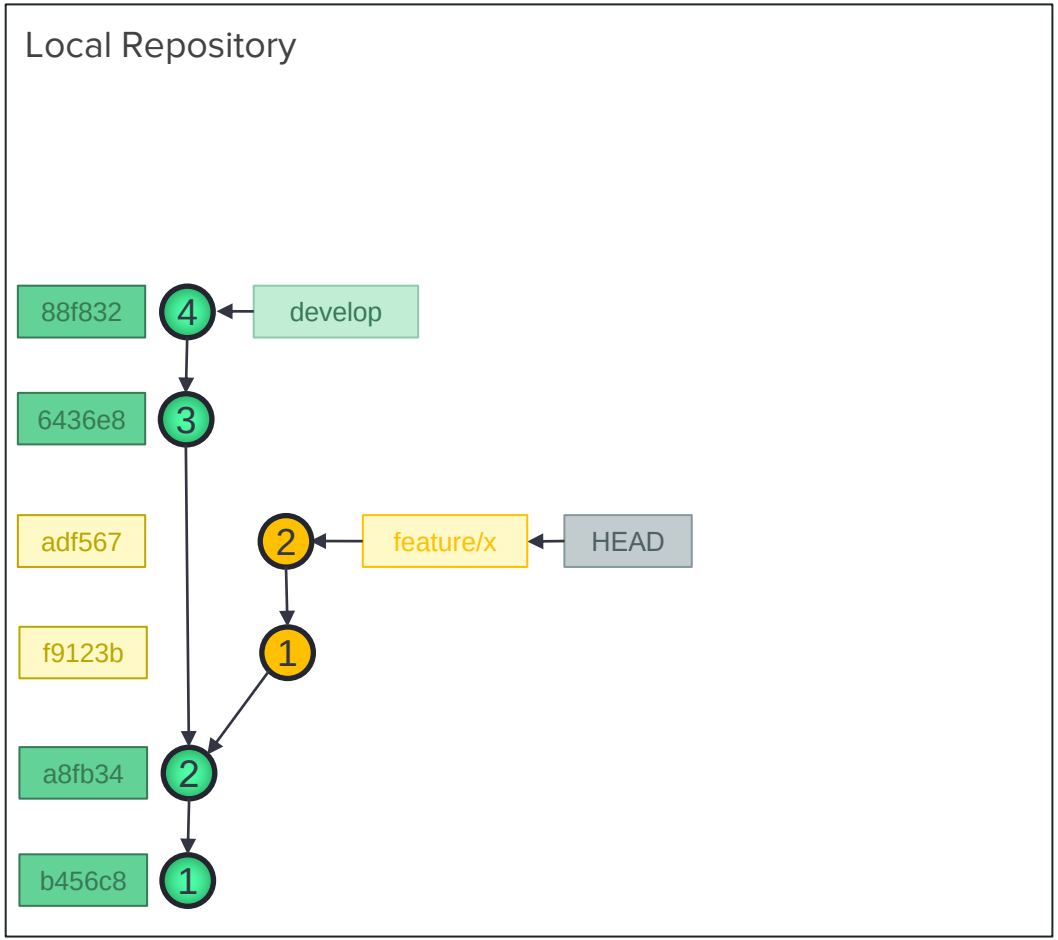
… something I shouldn’t have, is it now lost?


NOOOOOO!!!!!
memegenerator.net

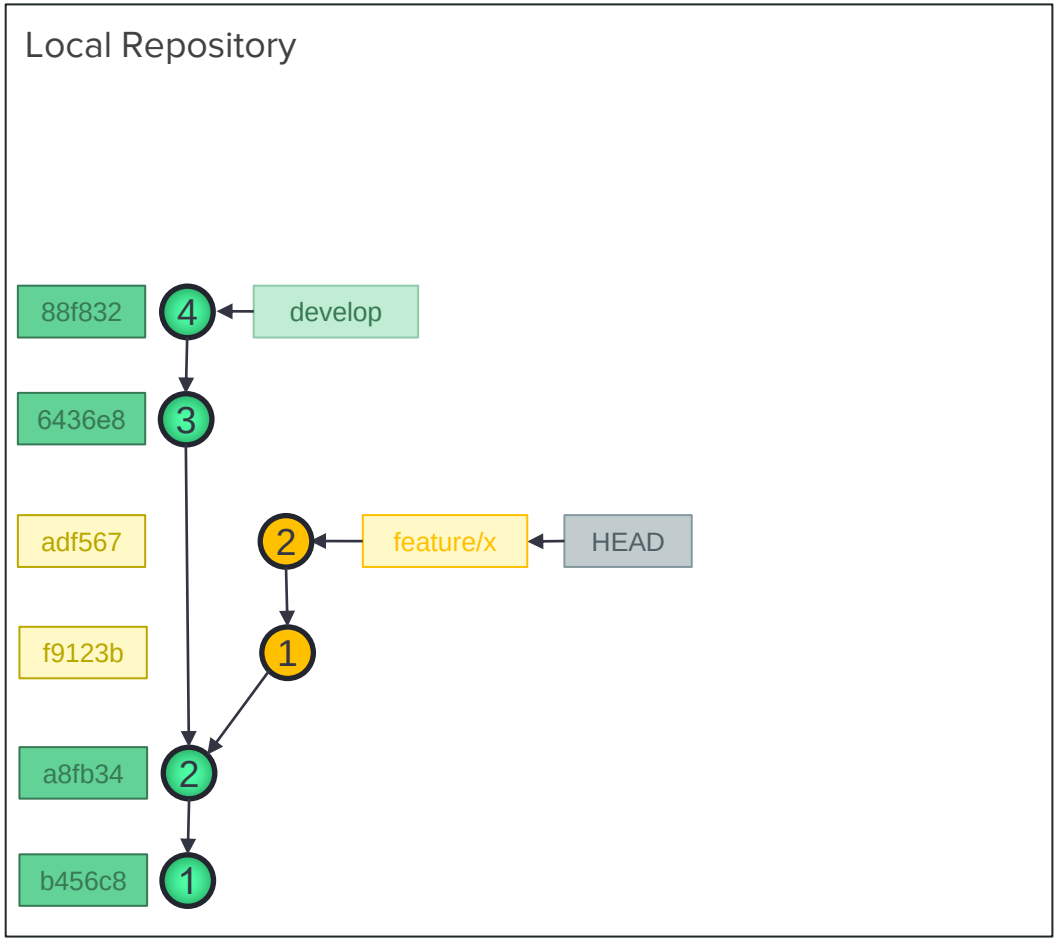
References

Write this SHA1 down before squashing / resetting / rebasing

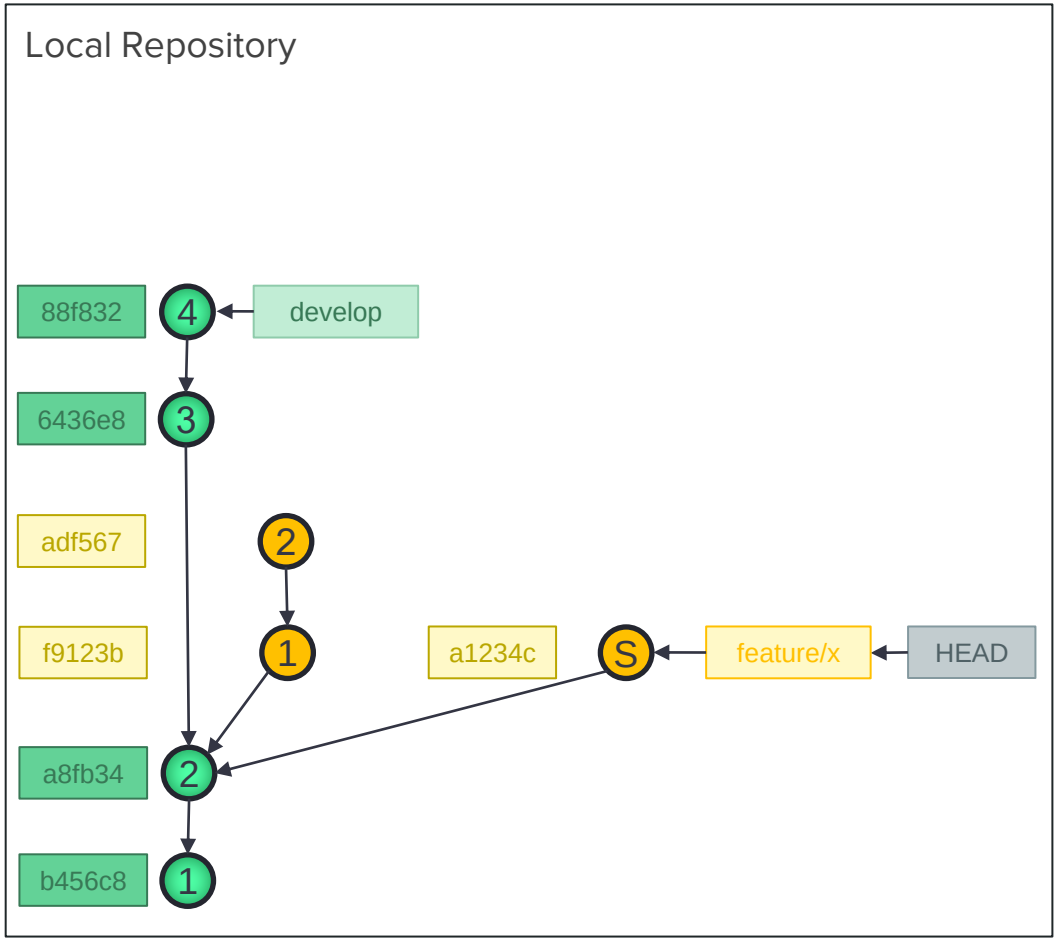
OR



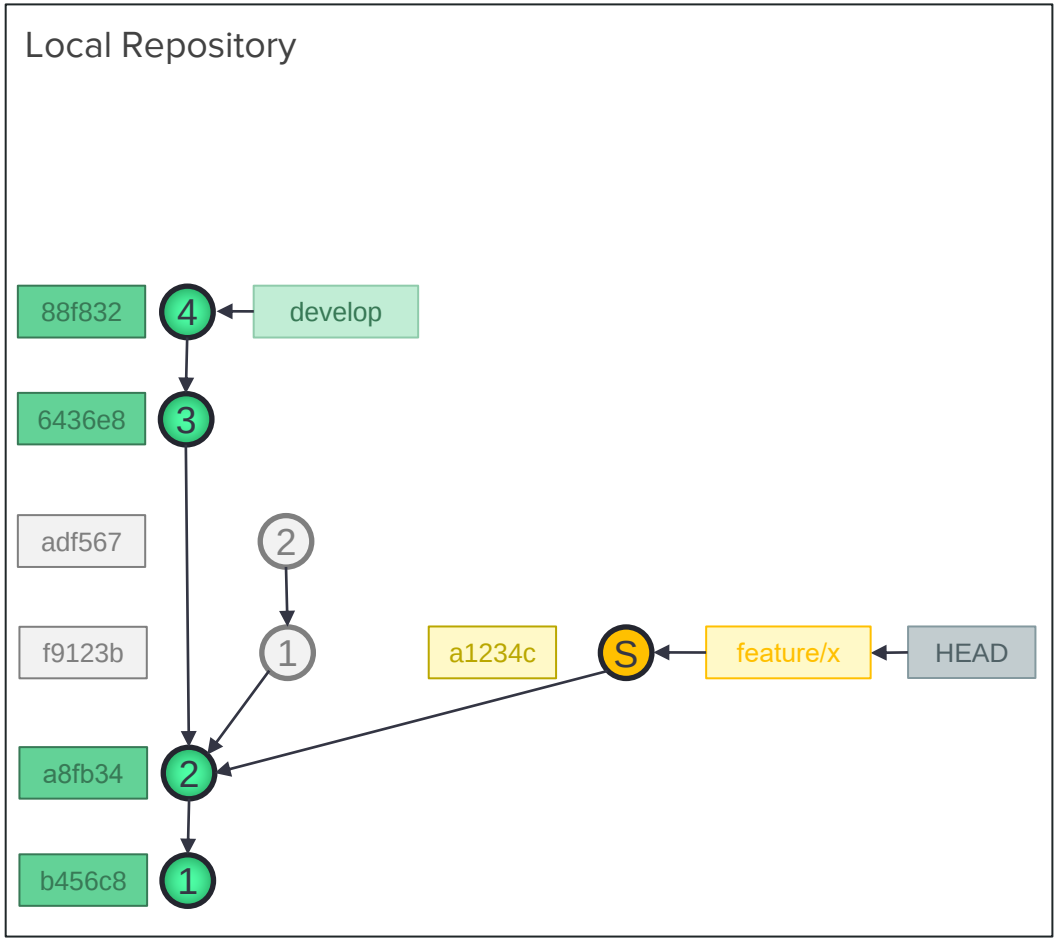
The Reflog


Local Repository
88f832
4
develop
6436e8
3
adf567
2
feature/x
HEAD
f9123b
1
a8fb34
2
b456c8
1

The Reflog

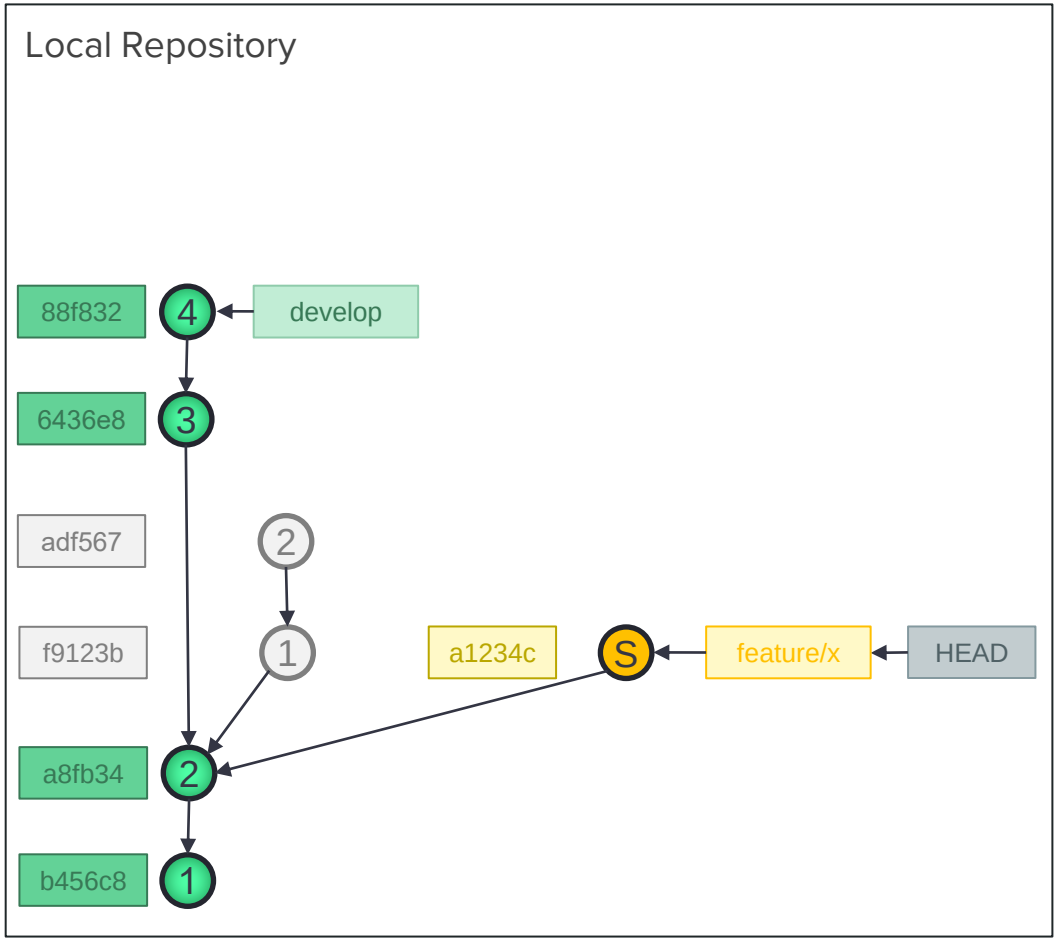

Local Repository
88f832
4
develop
6436e8
3
adf567
2
f9123b
1
a1234c
S
feature/x
HEAD
a8fb34
2
b456c8
1

The Reflog


Local Repository
88f832
4
develop
6436e8
3
adf567
2
f9123b
1
a1234c
S
feature/x
HEAD
a8fb34
2
b456c8
1

The Reflog

```
$ git reflog

$ |
```

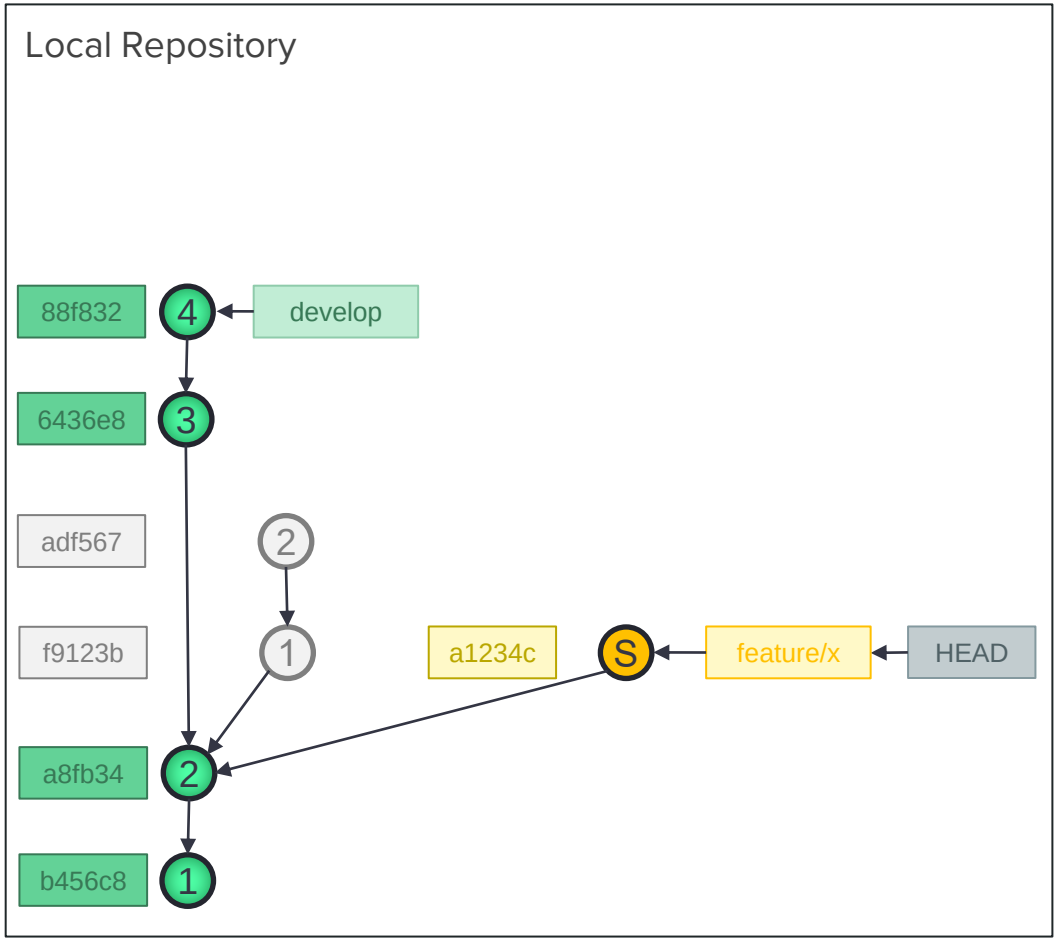


The Reflog

Try to find this commit.

It looks something like:
adf567 HEAD@{20}: commit: 2 develop

This bit here is the commit message!

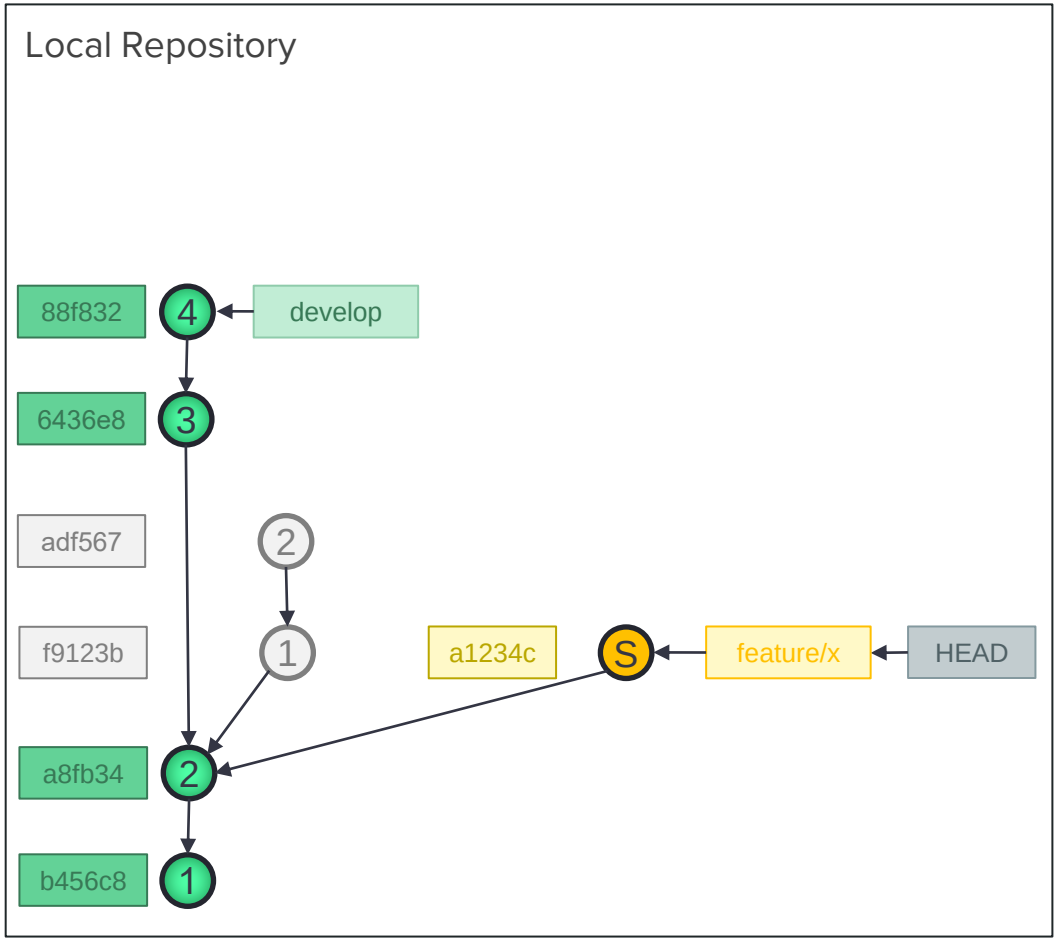


The Reflog

Try to find this commit.

It looks something like:
adf567 HEAD@{20}: commit: 2 develop

Copy this bit



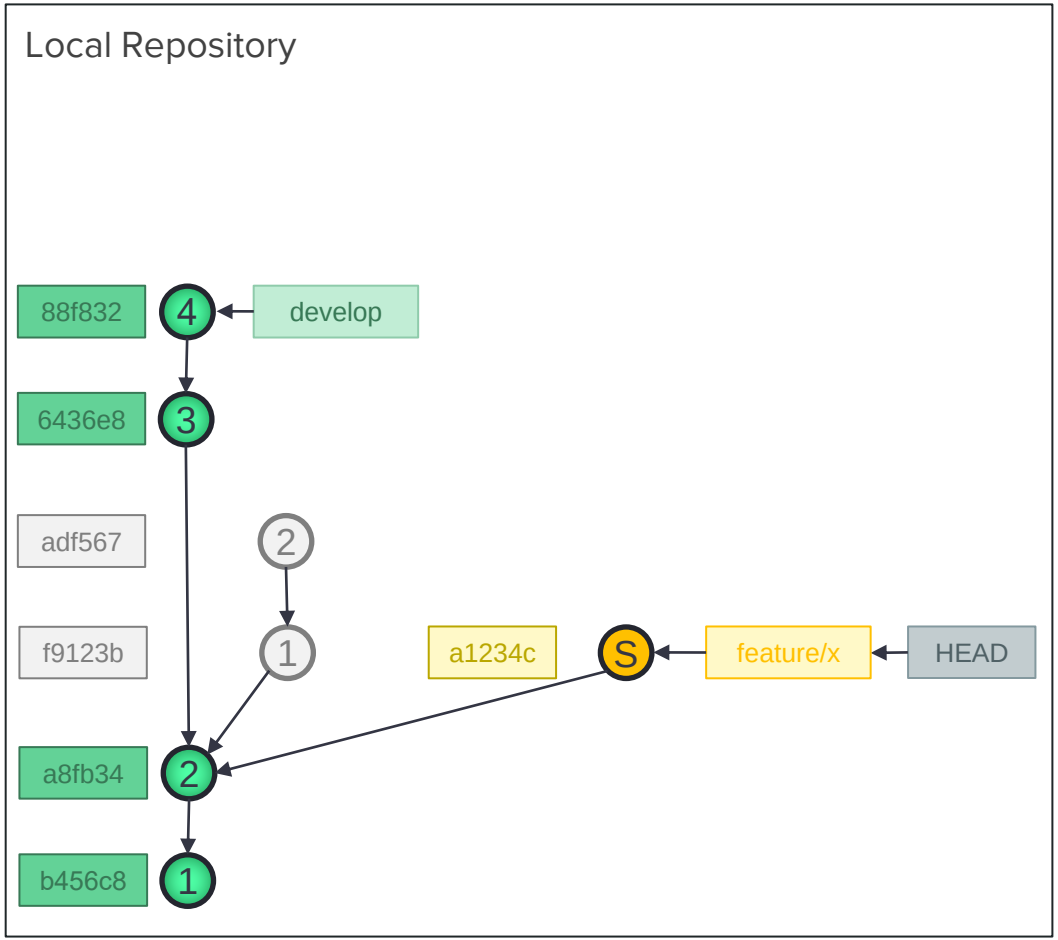
The reflog

The reflog is a place where you can see all actions you have performed with git

- Commits **are not instantly lost** when losing a reference, they stay in the repo until garbage collected (90 days by default)
- If you can find the SHA1 of a commit that **was the tip of the branch before** your history rewriting actions you are home free!

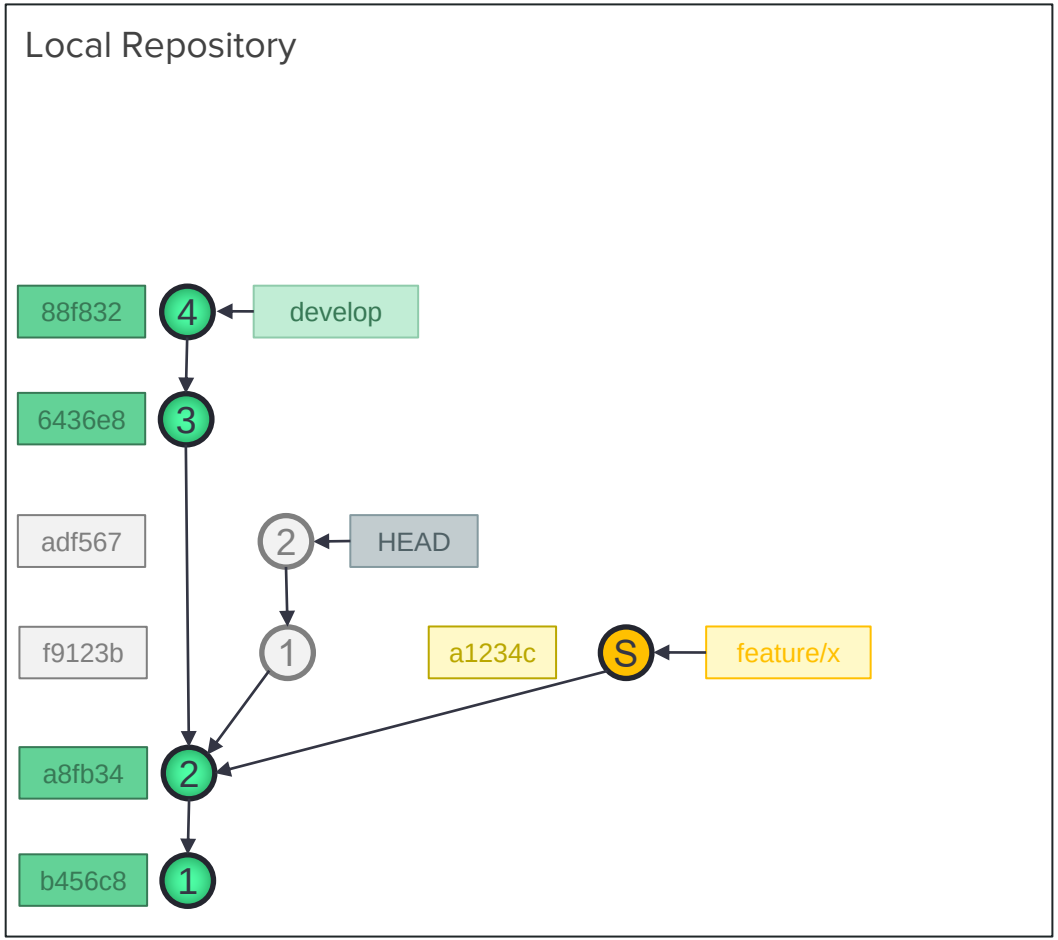
The Reflog

```
$ git reflog

$ git checkout adf567

$ |
```



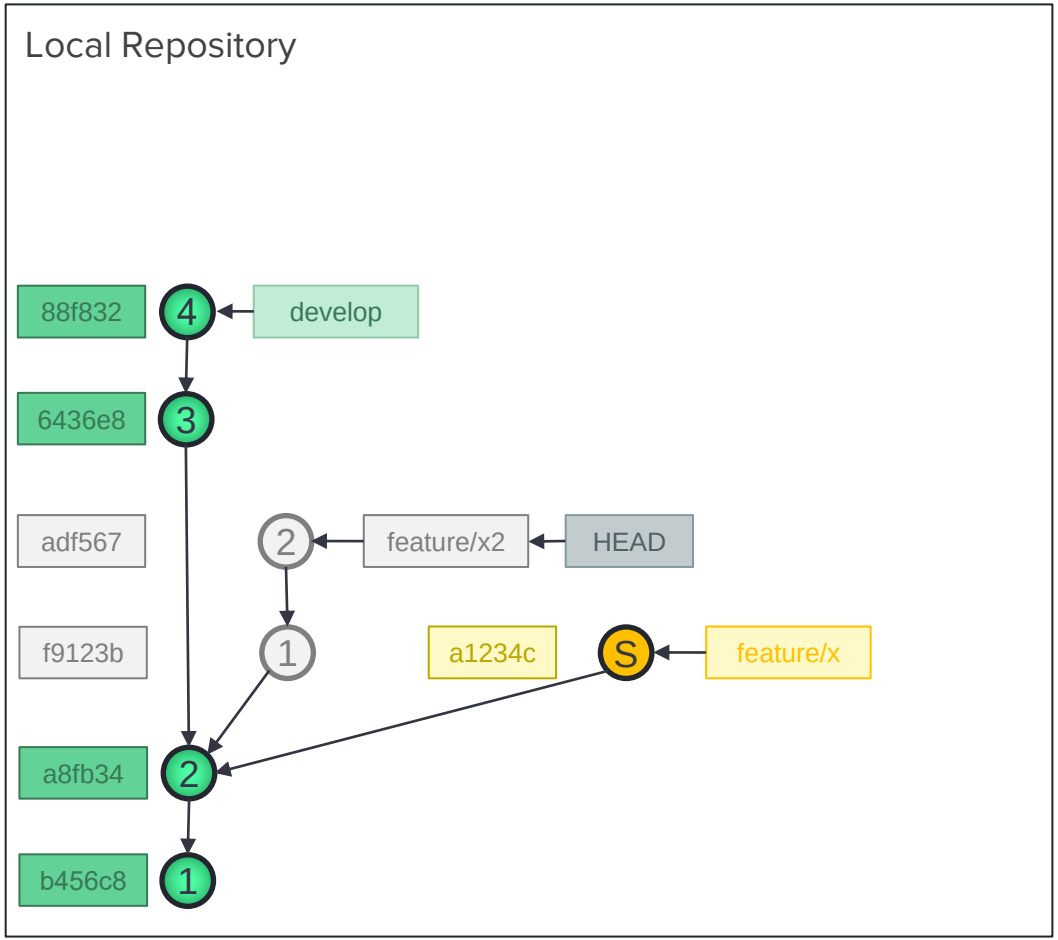
The Reflog

```
$ git reflog

$ git checkout adf567

$ |
```



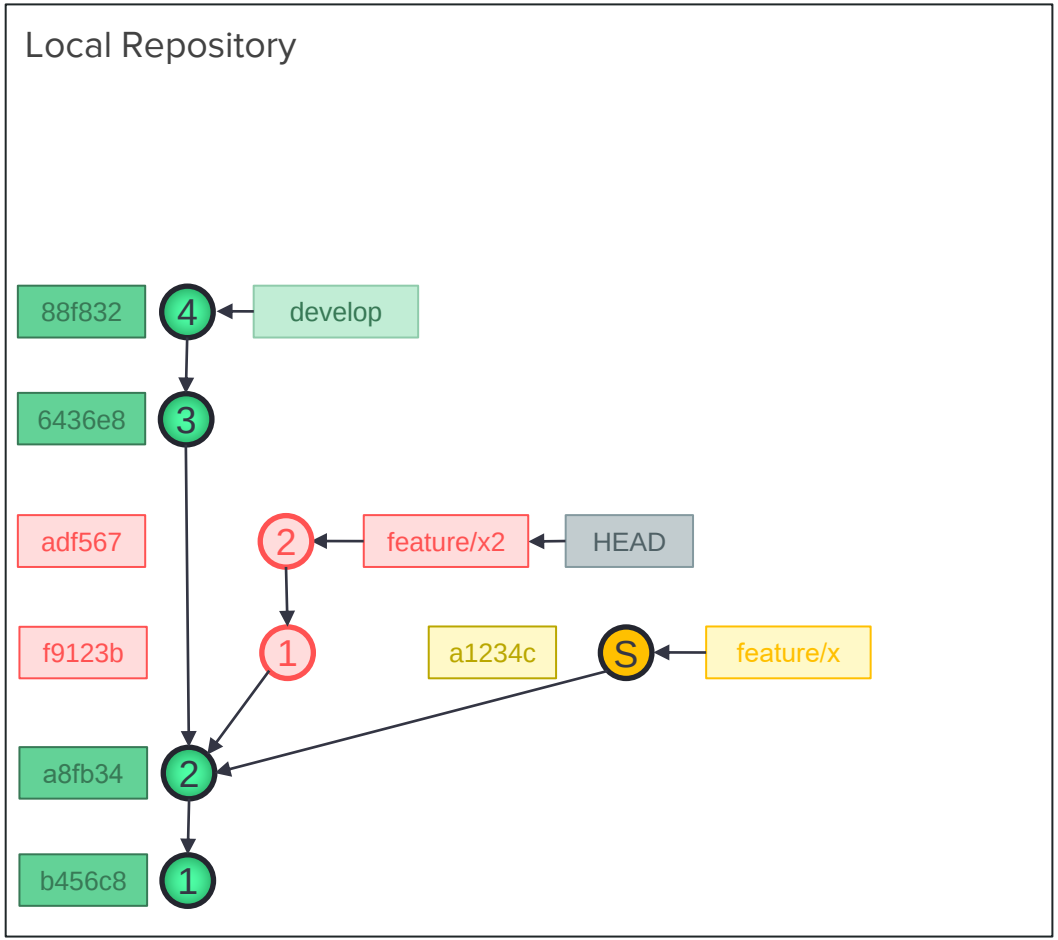
The Reflog

```
$ git reflog

$ git checkout adf567

$ git checkout -b
  feature/x2

$ |
```



The Reflog

```
$ git reflog

$ git checkout adf567

$ git checkout -b
  feature/x2

$ |
```



The reflog

- Similarly you can recover from resets:
  - Look for the SHA, right before the reset action started and use that.
- And normal rebases:
  - Just like the squash only you don’t have that extra squash commit, you have multiple new commits

Don’t forget!

- You can abort a rebase that has conflicts and do a merge instead (much easier)

- And most importantly ...

The Golden Rule of History Manipulation

The Golden Rule

- Before doing any history manipulation ask yourself:

Is anyone else looking at this branch?

Yes

No

The Golden Rule

Is anyone else looking at this branch?

- Is this branch pushed, is it a public branch?
- Is this branch a source branch for other branches?

YES!

Somebody else is also looking at this branch!


STOP

Thank you!