Software Version Control

What is Git?

What is Version Control?

Version control is a software that **tracks and manages changes** to files over time

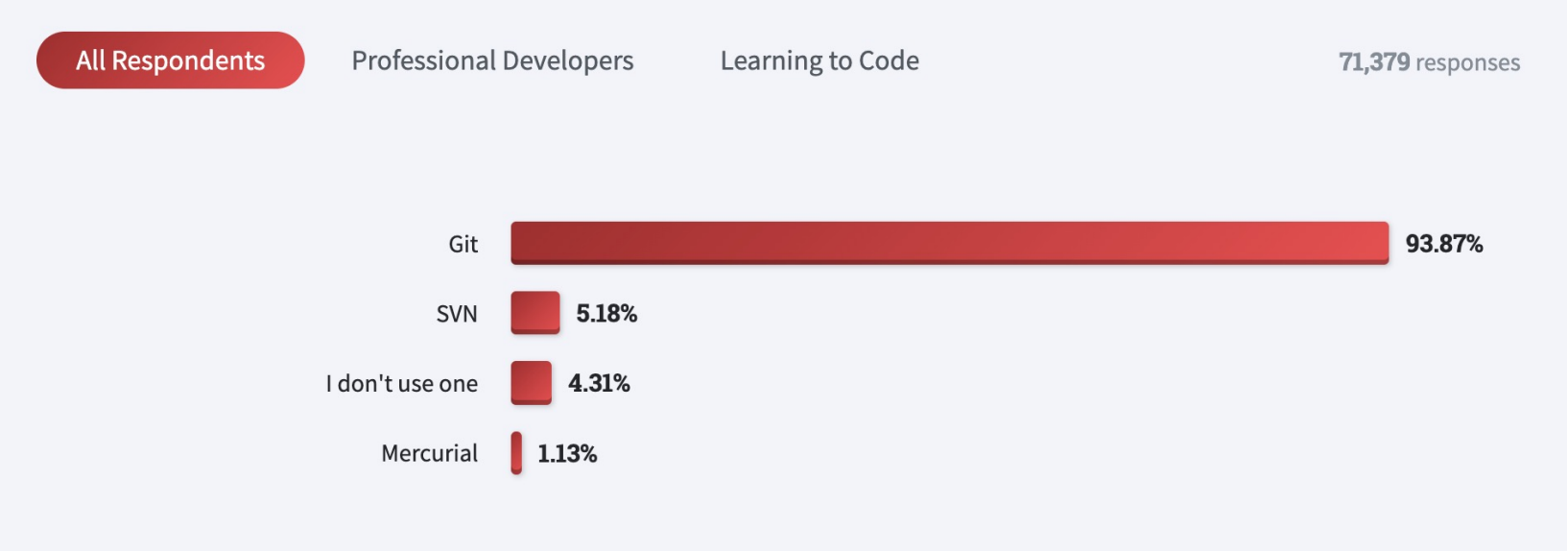
Version control systems generally allow users to revisit earlier versions of the files, compare changes between versions, undo changes and a whole lot more

Git is just one VCS

- Git is just one of the many version control system available today. Other well-know ones includes Subversion, CVS and Mercurial
- They all have similar goals but vary significantly in how they achieve those goals. Fortunately, we only need to care about Git because...

Git is the clear “winner”

No other technology is as widely used as Git. Especially among Professional Developers. But for those learning to code, 17% still do not use a version control system.

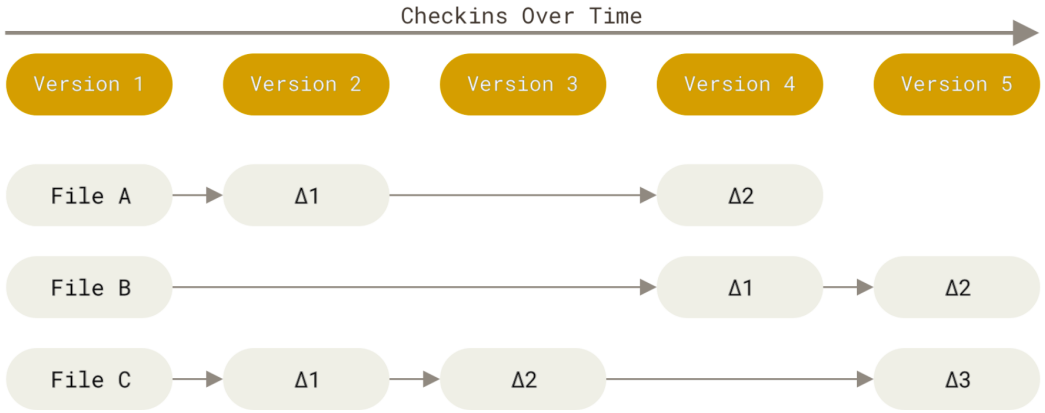


https://survey.stackoverflow.co/2022/#worked-with-vs-want-to-work-with-office-stack-sync-worked-want

Git helps us...

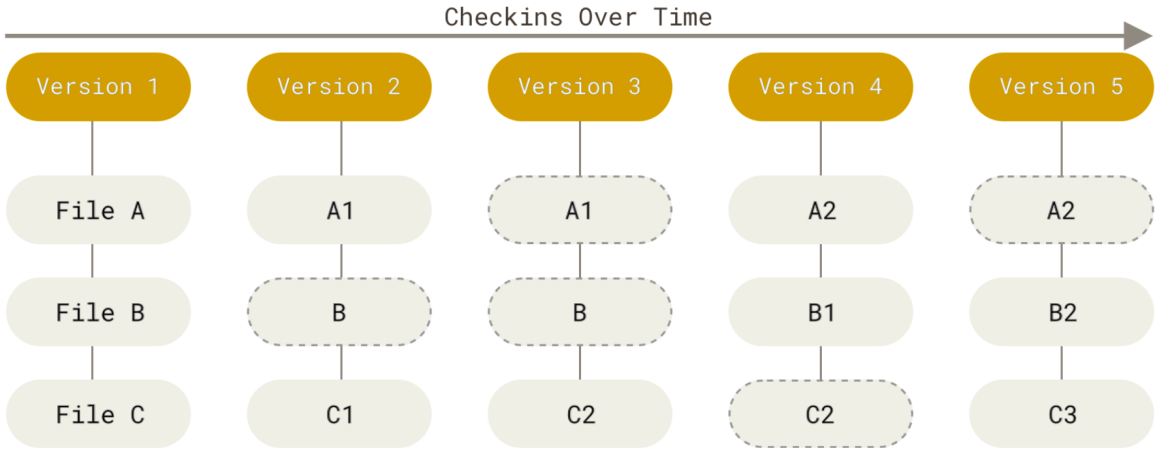
- Track changes across multiple files
- Compare versions of a project
- "Time travel" back to old versions
- Revert to a previous version
- Collaborate and share changes
- Combine changes

Snapshots, Not Differences



`delta-based` version control

Storing data as changes to a base version of each file



`GIT`

Storing data as snapshots of the project over time

TRAINOCATE

First-Time Git Setup

How To I Get Started

1. Installing Git

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

```
$ sudo apt install git-all
```

2. Configure your identity on Git

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
$ git config --list
```

Getting and Creating Projects

TRAINOCATE

Initializing A Repository In An Existing Directory

- If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory.

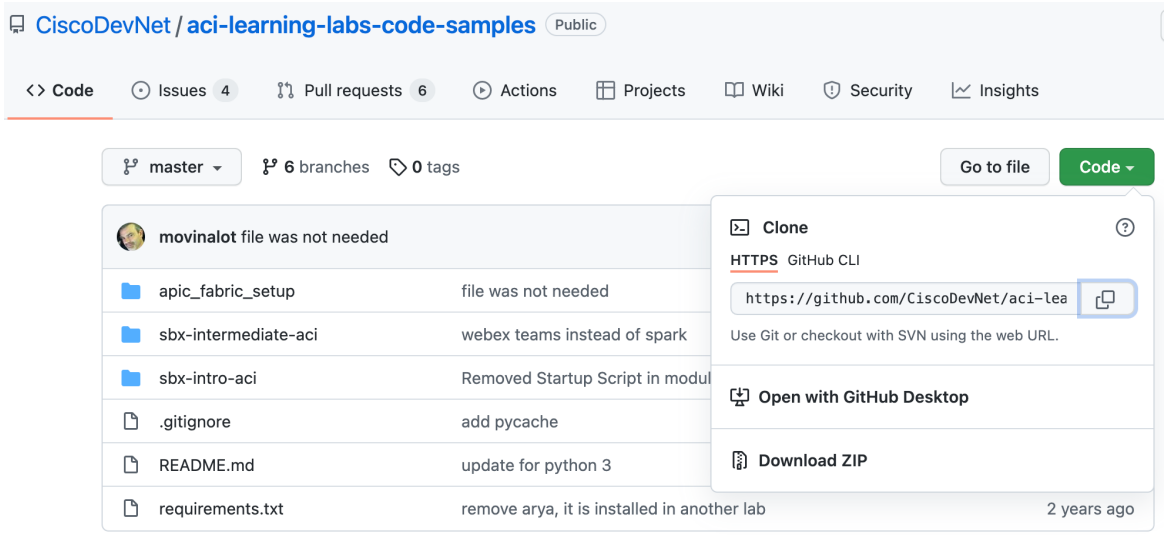
```
$ cd /home/user/my_project
$ git init
```

- This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet.

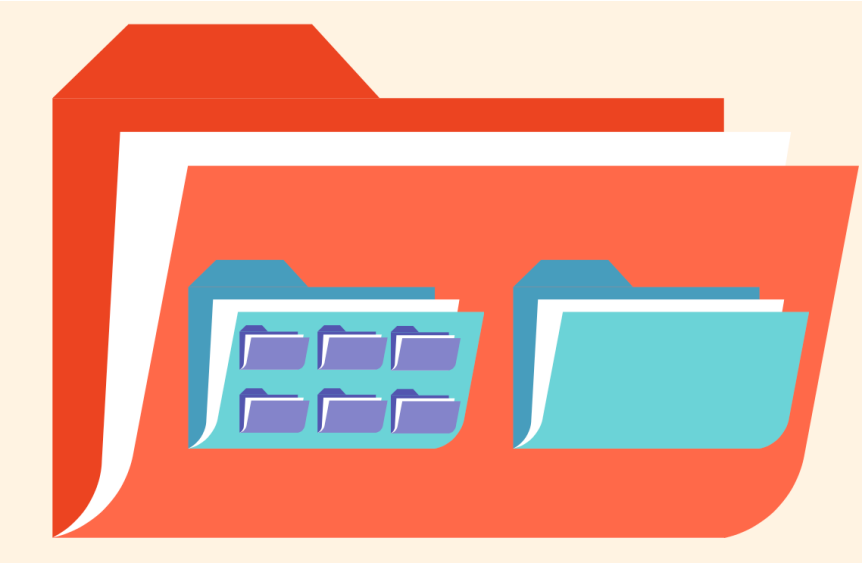
Cloning An Existing Repository

- If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is git clone

```
$ git clone https://github.com/CiscoDevNet/aci-learning-labs-code-samples.git
```



Git tracks A Directory And All Nested Subdirectories



- WARNING: Do not init a repo inside of a repo.
- Before running git init, use git status to verify that you are not currently inside of a repo
- If you do end up making a repo inside of a repo, you can delete it and try again!
- To delete a repo, locate the associated .git directory and delete it.

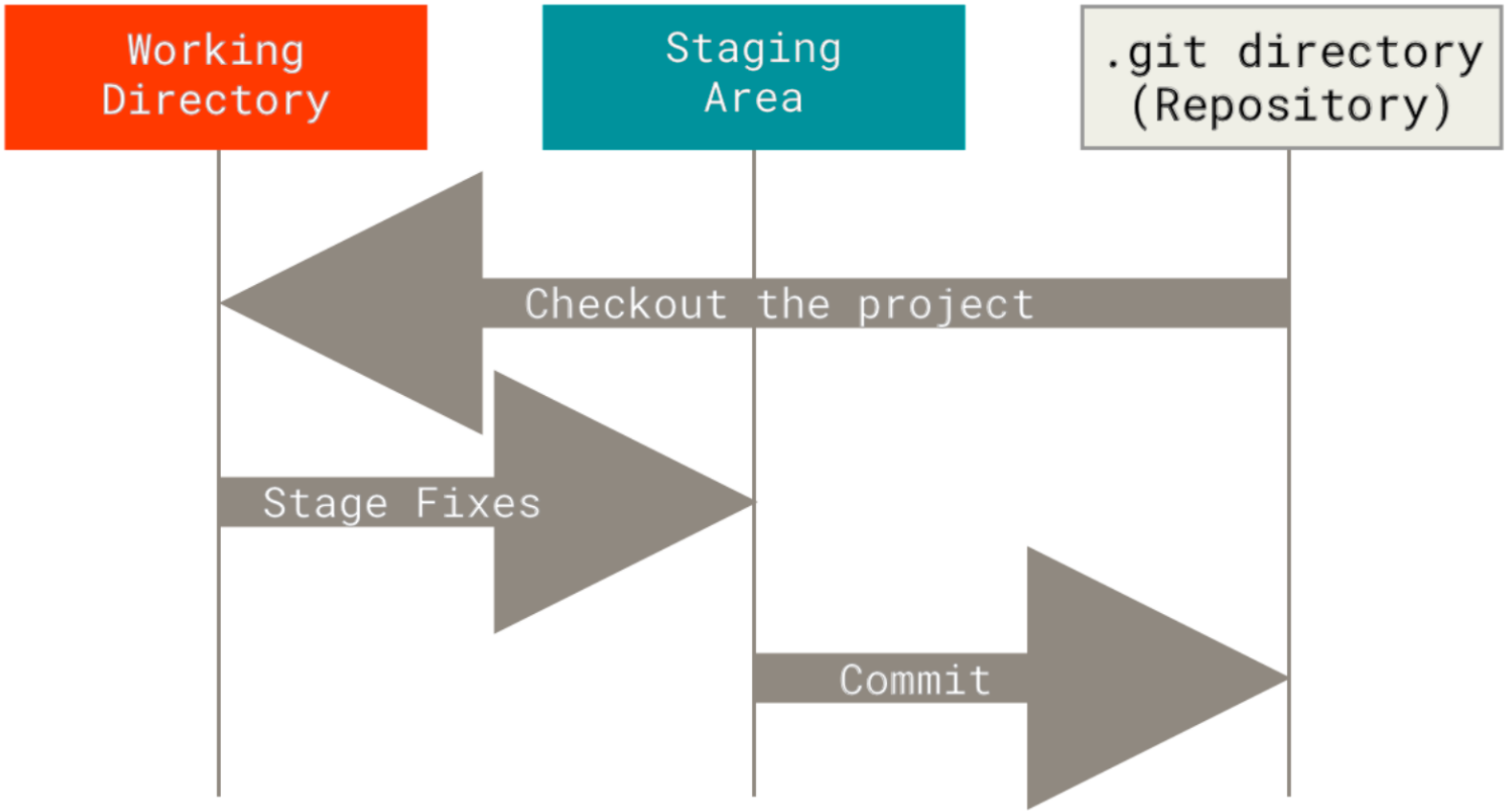
Git basics

TRAINOCATE

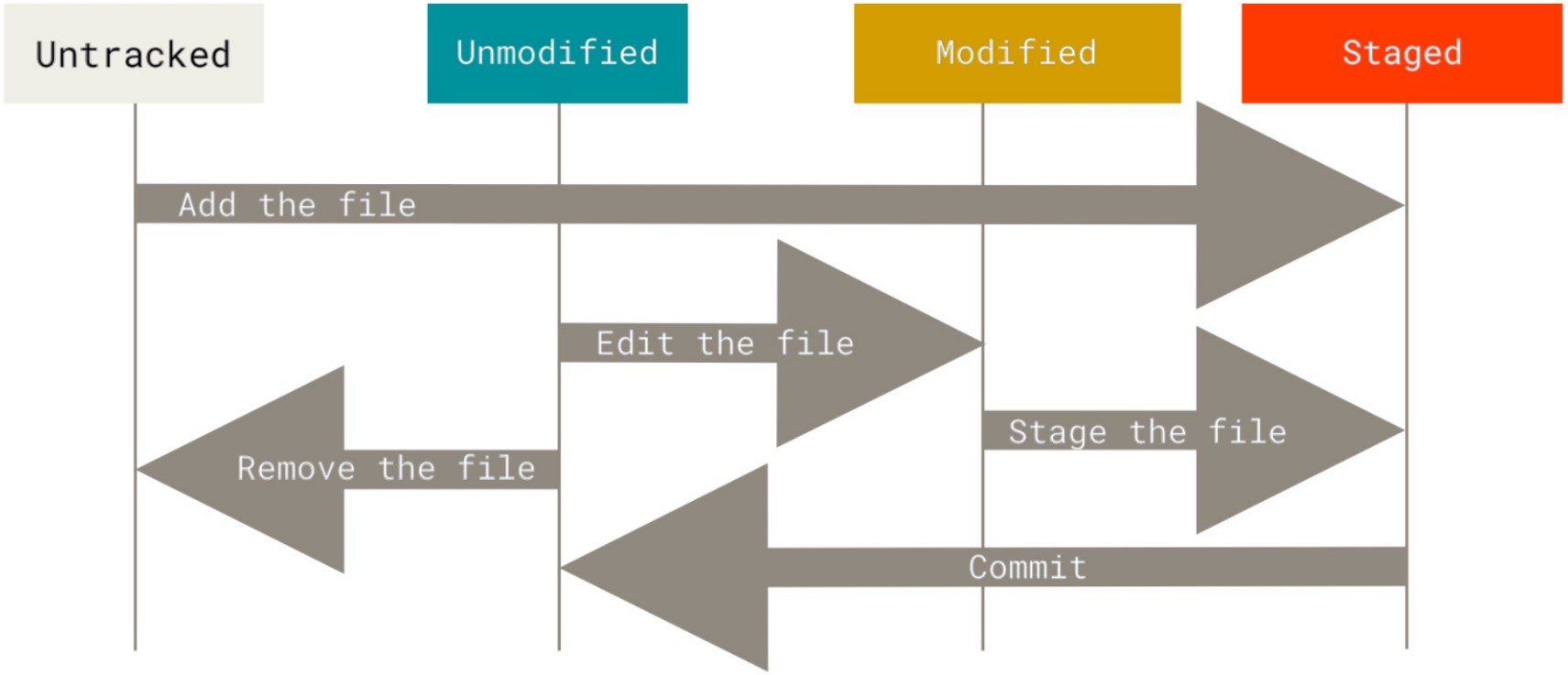
Working tree, staging area, and Git directory

- The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well
- The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

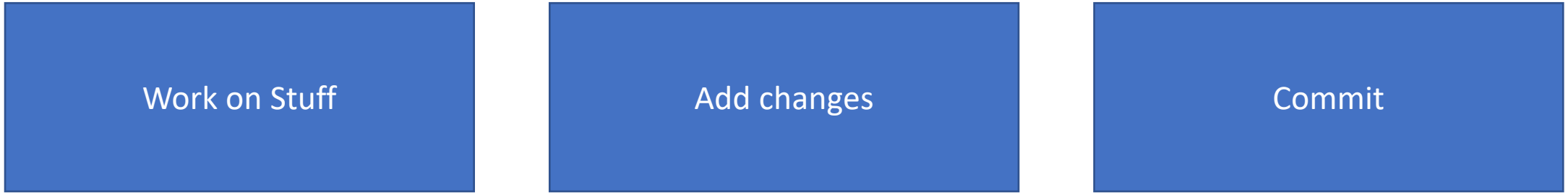
Working tree, staging area, and Git directory



The lifecycle of the status of your files



The basic Git workflow

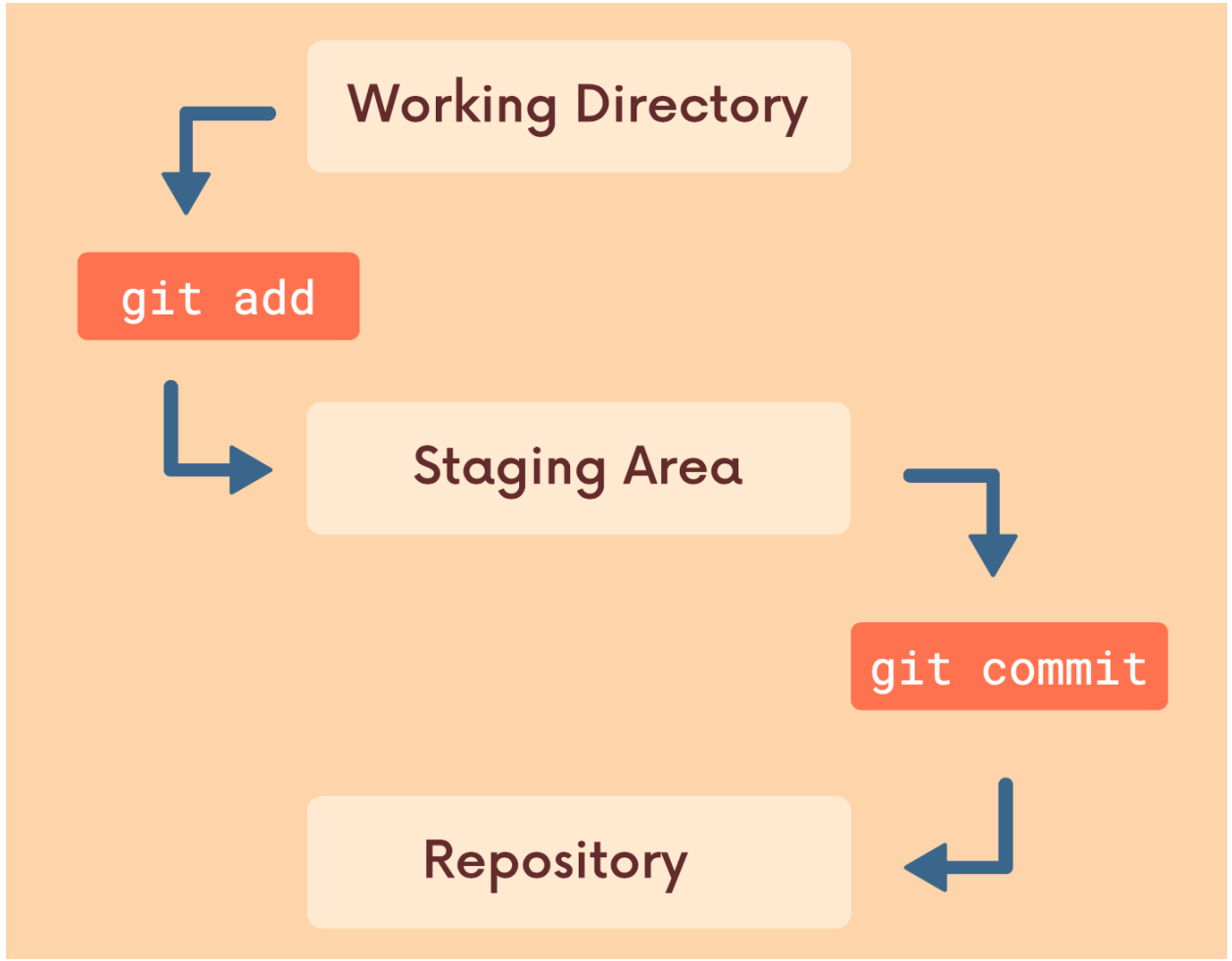


Make new files, edit files, delete files, etc

Group specific changes together, in preparation of committing.

Commit everything that was previously added

The basic Git workflow



git status

- Git status command show the list of paths/file that have differences between the local directory and the last commit in the repository

```
$ git status
```

- Also shows a list of untracked files that are not part of the repository

```
$$ cd project1
$ touch app.py
$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
app.py
nothing added to commit but untracked files present (use "git add" to track)
```

git add

- In order to begin tracking a new file, you use the command git add. To begin tracking the app.py file, you can run this:

```
$ git add app.py
```

- Add files with a wildcard

```
$ git add a*.py
```

- Add all files that have been changed

```
$ git add .
```

```
$ git status
On branch master

No commits yet

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

TRAINOCATE

Staging Modified Files

```
$ git status
On branch master

No commits yet

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be
committed)
  (use "git restore <file>..." to discard changes in
working directory)
        modified:   app.py
```

```
$ git add app.py
$ git status
On branch master

No commits yet

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

If you modify a file after you run git add, you have to run git add again to stage the latest version of the file

git commit

- We use the git commit command to actually commit changes from the staging area.
- Remember that anything that is still unstaged — any files you have created or modified that you haven't run git add on since you edited them — won't go into this commit. They will stay as modified files on your disk.
- When making a commit, we need to provide a commit message that summarizes the changes and work snapshotted in the commit
- Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

```
$ git commit -m "initial version"
[master (root-commit) 4700147] initial version
 1 file changed, 2 insertions(+)
 create mode 100644 app.py
```

Git commit

- Running git commit will commit all staged changes. It also opens up a text editor and prompts you for a commit message.

```
$ git commit
```

- The -m flag allows us to pass in an inline commit message, rather than launching a text editor.

```
$ git commit –m “my message”
```

Atomic commits

- When possible, a commit should encompass a single feature, change or fix. In other words, try to keep each commit focused on a single thing
- This makes it much easier to undo or rollback change later on. It also makes your code or project easier to review

Removing Files

- To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

```
$ rm app.py
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    app.py

no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git rm app.py
rm 'app.py'
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    app.py
$ git commit -m "deleting app.py"
[master 4ca4881] deleting app.py
 1 file changed, 2 deletions(-)
 delete mode 100644 app.py
```

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

```
$ git log
commit
4ca488171144265134db8f907d525e1116a25c4f
(HEAD -> master)
Author: devnet <devnet.cisco.vn@gmail.com>
Date:   Sat Apr 2 11:19:33 2022 -0400

    deleting app.py

commit
47001474f2613cace52f9fca24c42ebca4241c17
Author: devnet <devnet.cisco.vn@gmail.com>
Date:   Sat Apr 2 11:07:55 2022 -0400

    initial version
```

Common Options To Git Log

Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the –stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

```
$ git log --oneline
4ca4881 (HEAD -> master) deleting app.py
4700147 initial version
```

git log --pretty=format

Specifier	Description of Output
%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author email
%ad	Author date (format respects the --date=option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

```
$ git log --pretty=format:"%h - %an, %ar : %s"
4ca4881 - devnet, 12 hours ago : deleting app.py
4700147 - devnet, 13 hours ago : initial version
```

Limiting Log Output

Option	Description
`-<n>`	Show only the last n commits
`--since`, `--after`	Limit the commits to those made after the specified date.
`--until`, `--before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string
`-S`	Only show commits adding or removing code matching the string

git show

```
$ git show 4700147
commit 47001474f2613cace52f9fca24c42ebca4241c17
Author: devnet <devnet.cisco.vn@gmail.com>
Date:   Sat Apr 2 11:07:55 2022 -0400

    initial version

diff --git a/app.py b/app.py
new file mode 100644
index 0000000..1f7d4f7
--- /dev/null
+++ b/app.py
@@ -0,0 +1,2 @@
+def build_config:
+    return
```

$ git show <commit-hash>

Redo a commit

- Suppose you just made a commit and then realized you forgot to include a file! Or maybe you made a typo in the commit message that you want to correct.
- Rather than making a brand-new separate commit, you can "redo" the previous commit using the --amend option

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

Ignoring files

We can tell Git which files and directories to ignore in each repository, using a .gitignore file. This is useful for files you know you NEVER want to commit, including:

- Secrets, API keys, credentials, etc.
- Operating System Files (.DS_Store on Mac)
- Log files
- Dependencies and packages

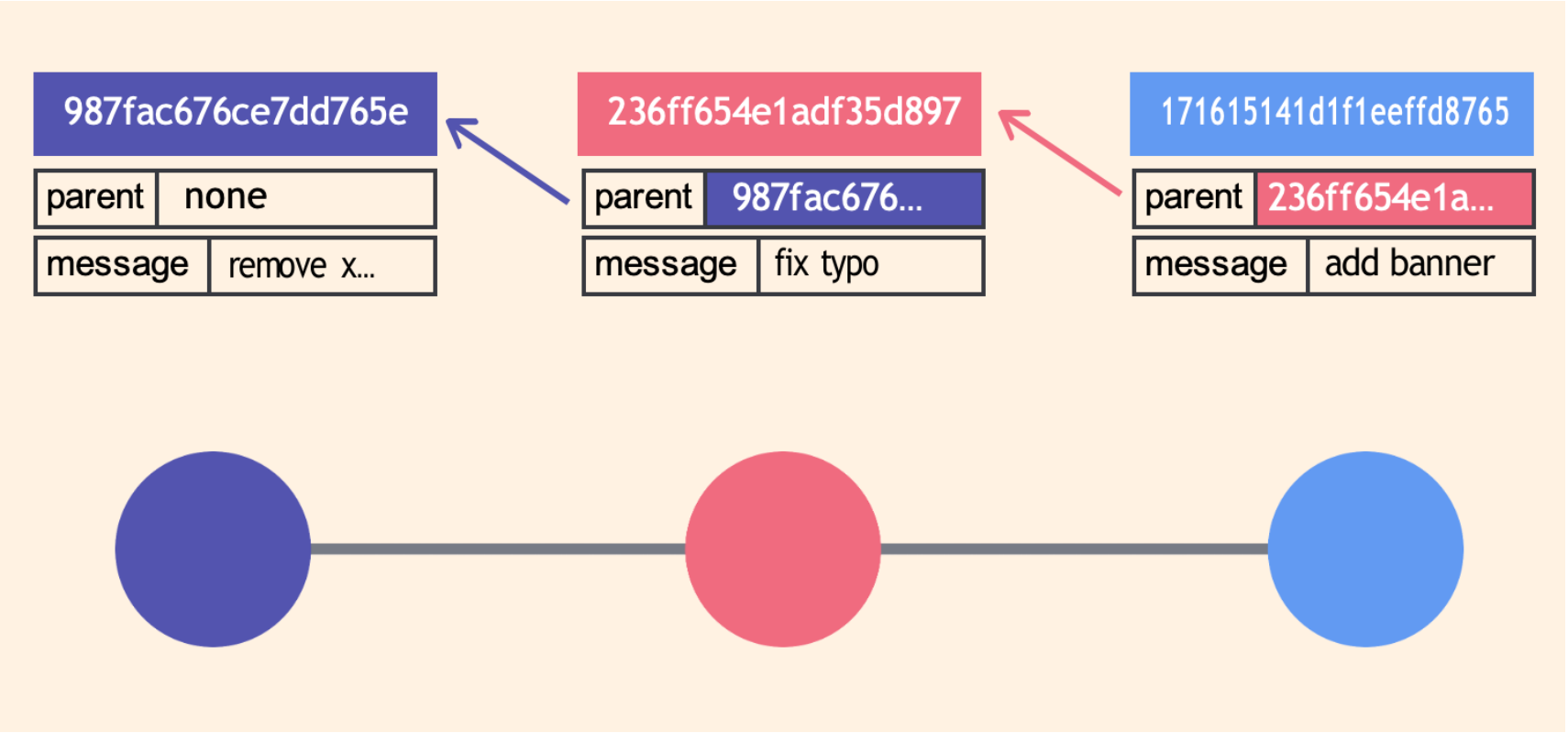
.gitignore

Create a file called .gitignore in the root of a repository. Inside a file, you can write patterns to tell Git which files and folders to ignores:

- .DS_Store will ignore files name .DS_Store
- folderName/ will ignore an entire directory
- *.log will ignore any files with .log extension

Branching

Commits and their parents



Contexts

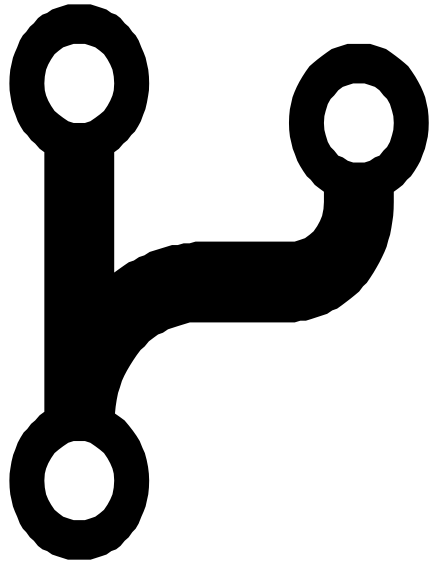
On large projects, we often work in multiple contexts:

- You're working on 2 different color scheme variations for your website at the same time, unsure of which you like best
- You're also trying to fix a horrible bug, but it's proving tough to solve. You need to really hunt around and toggle some code on and off to figure it out.
- A teammate is also working on adding a new chat widget to present at the next meeting. It's unclear if your company will end up using it.
- Another coworker is updating the search bar autocomplete.
- Another developer is doing an experimental radical design overhaul of the entire layout to present next month

TRAINOCATE

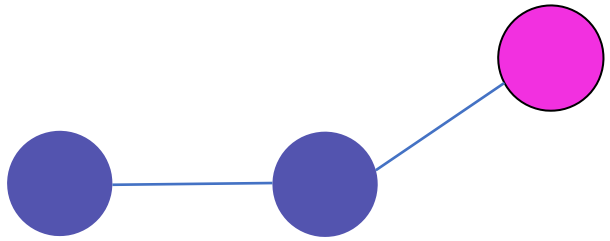
Branches

- Branches are an essential part of Git!
- Think of branches as alternative timelines for a project.
- They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.
- If we make changes on one branch, they do not impact the other branches (unless we merge the changes)



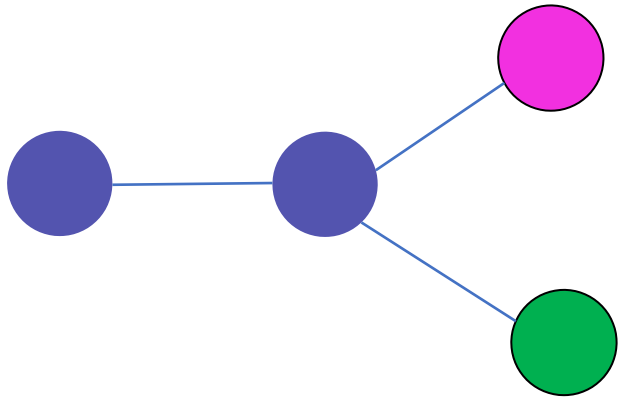
TRAINOCATE

New Color Theme

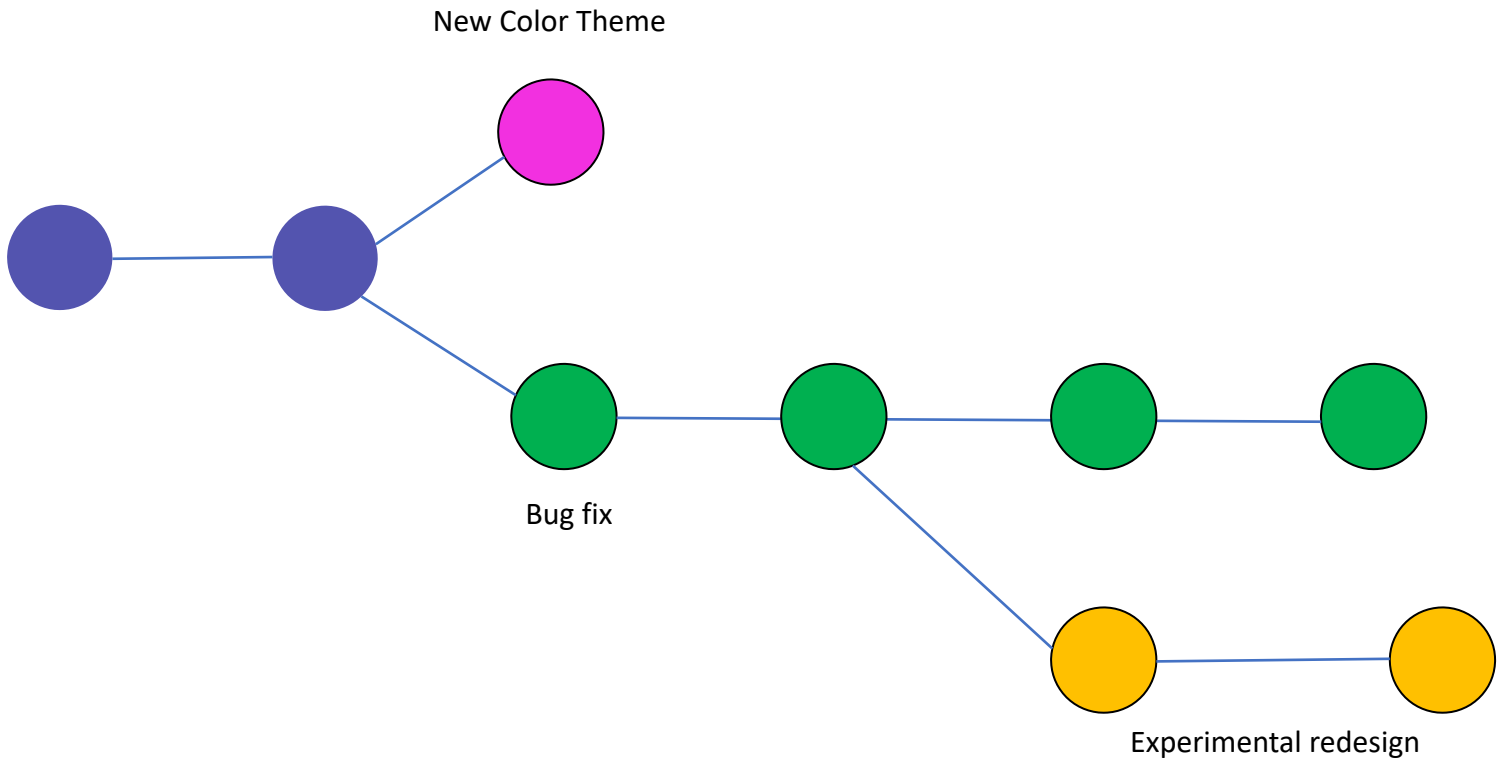


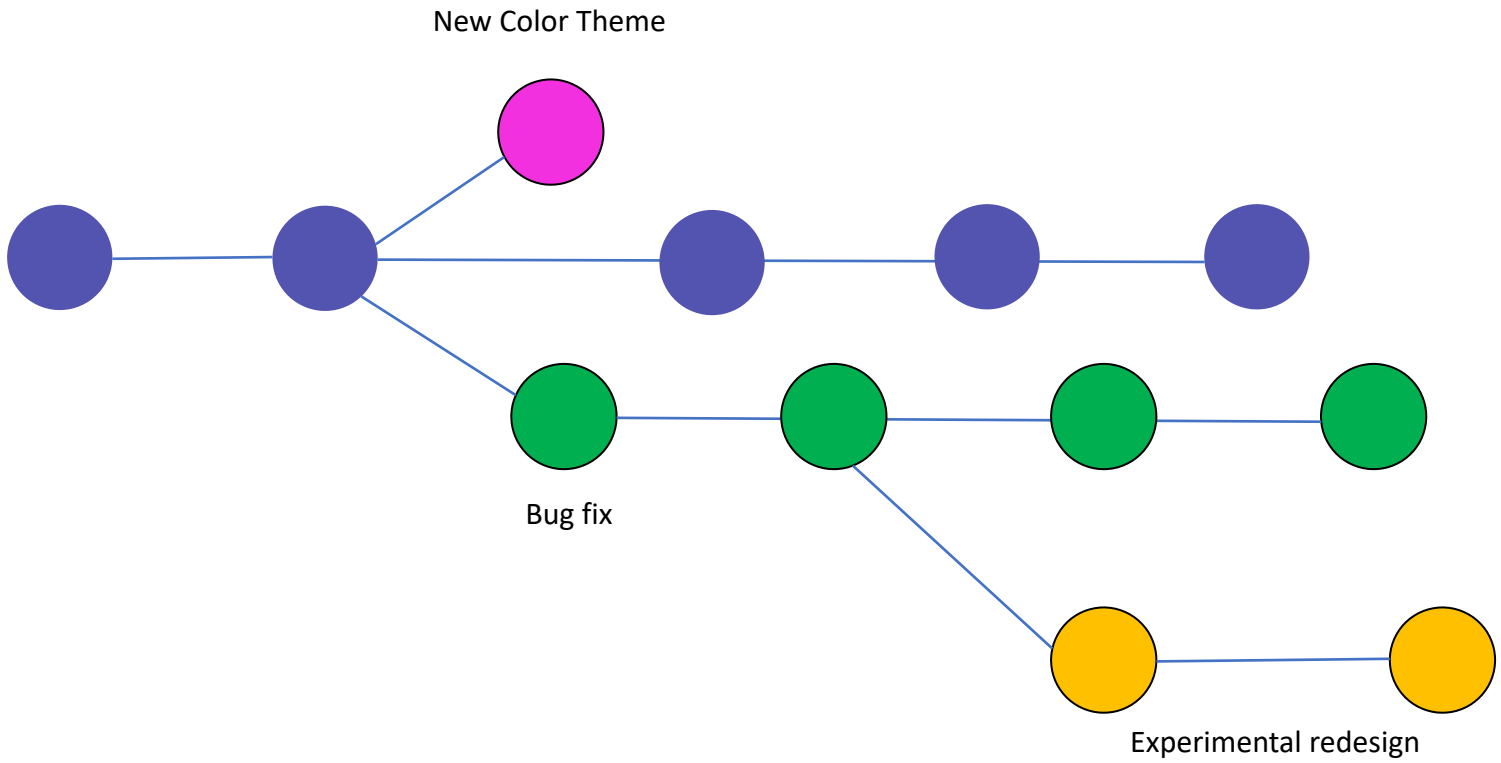
TRAINOCATE

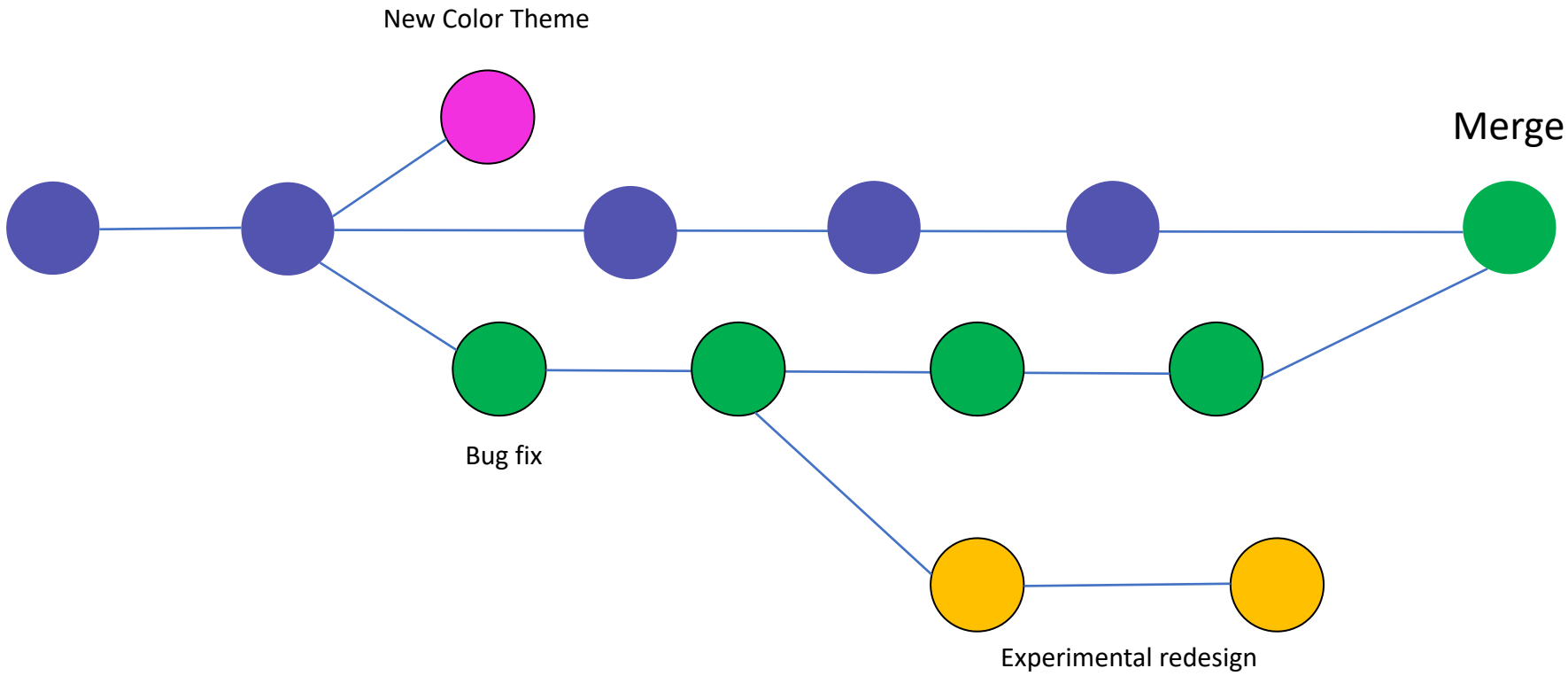
New Color Theme



Bug fix

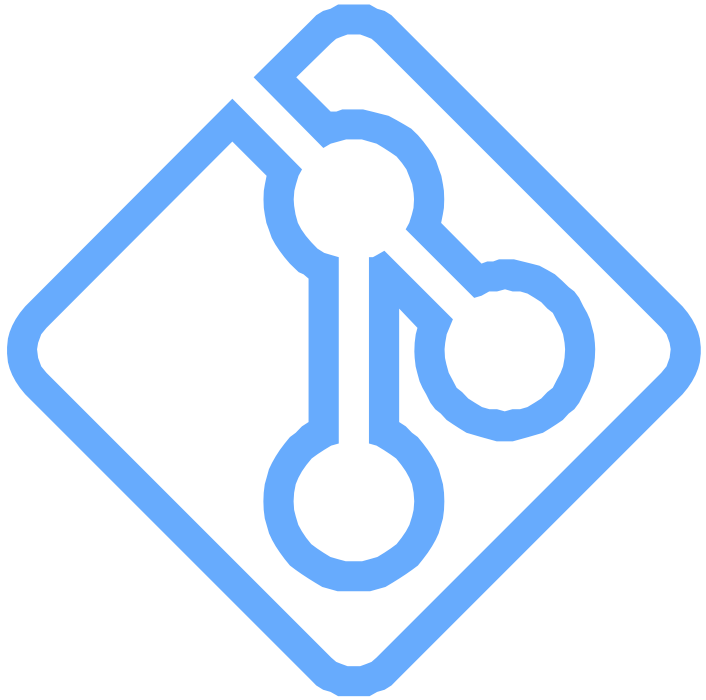






The master branch

- In git, we are always working on a branch. The default branch name is master.
- It doesn't do anything special or have fancy powers. It's just like any other branch.



Master

- Many people designate the master branch as their "source of truth" or the "official branch" for their codebase, but that is left to you to decide.

- From Git's perspective, the master branch is just like any other branch. It does not have to hold the "master copy" of your project.

Branching

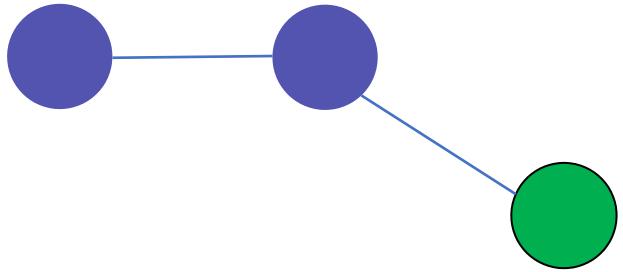
Master branch

Branching

Master branch

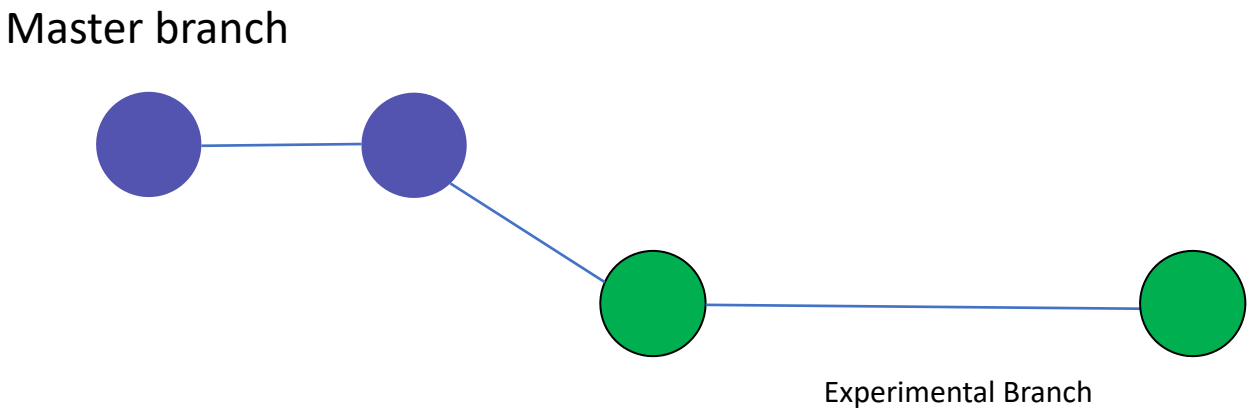
Branching

Master branch

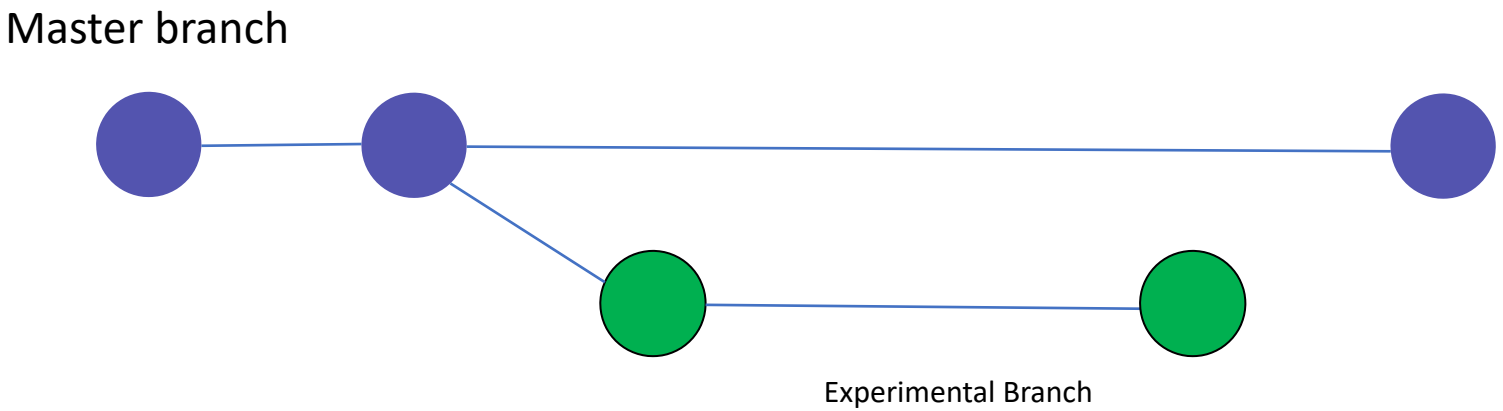


Experimental Branch

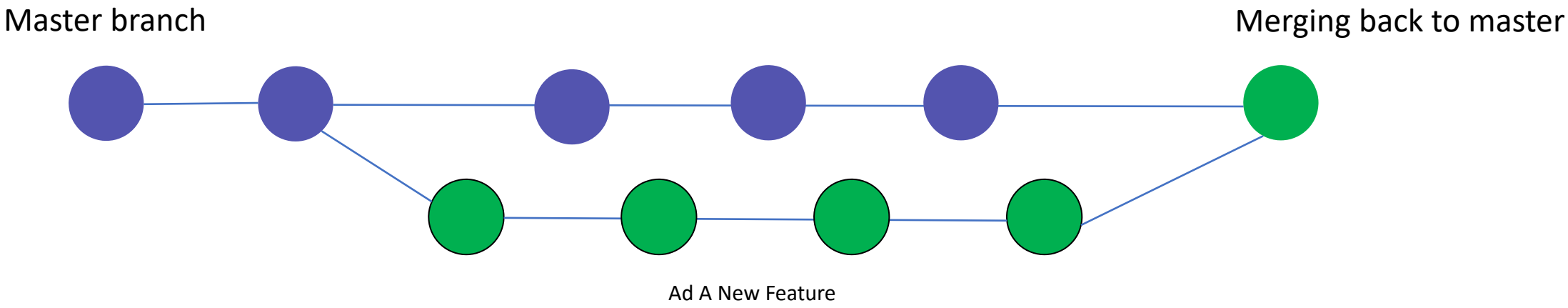
Branching



Branching



A Common Workflow



Viewing branches

- Use git branch to view your existing branches. The default branch in every git repo is master, though you can configure this.

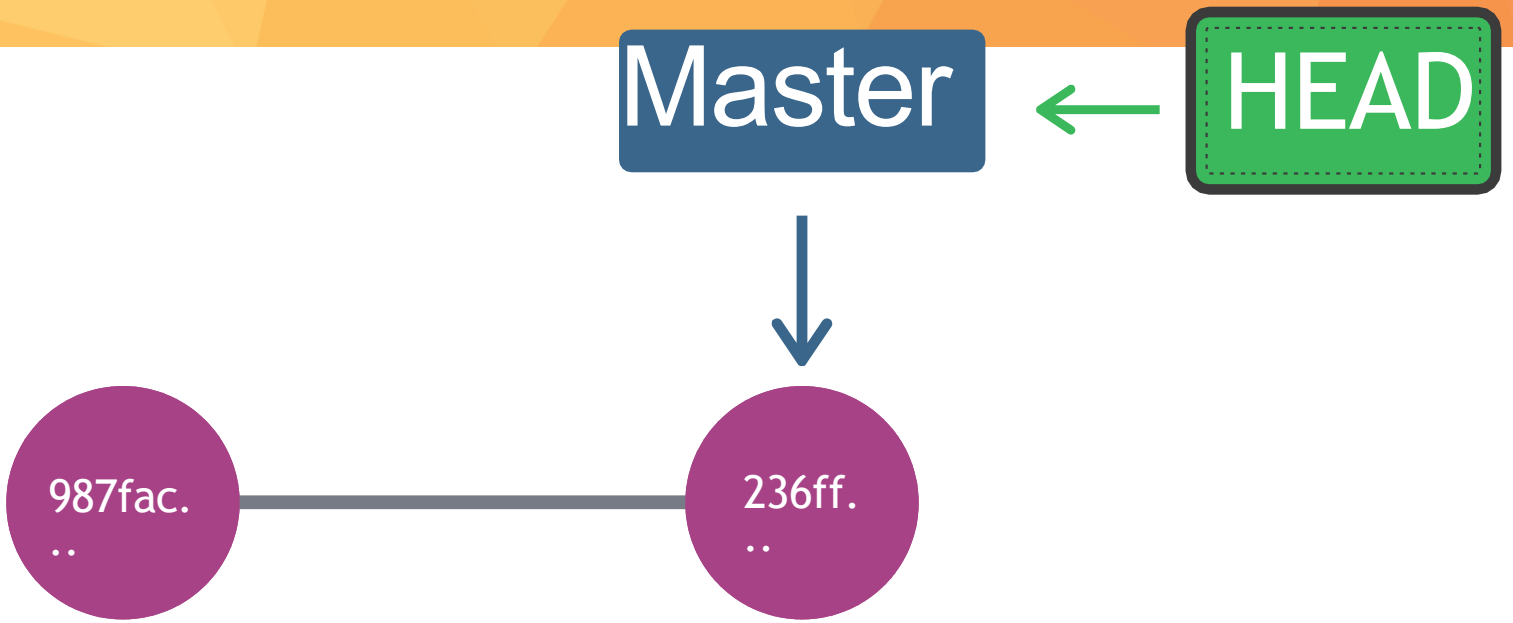
- Look for the * which indicates the branch you are currently on.

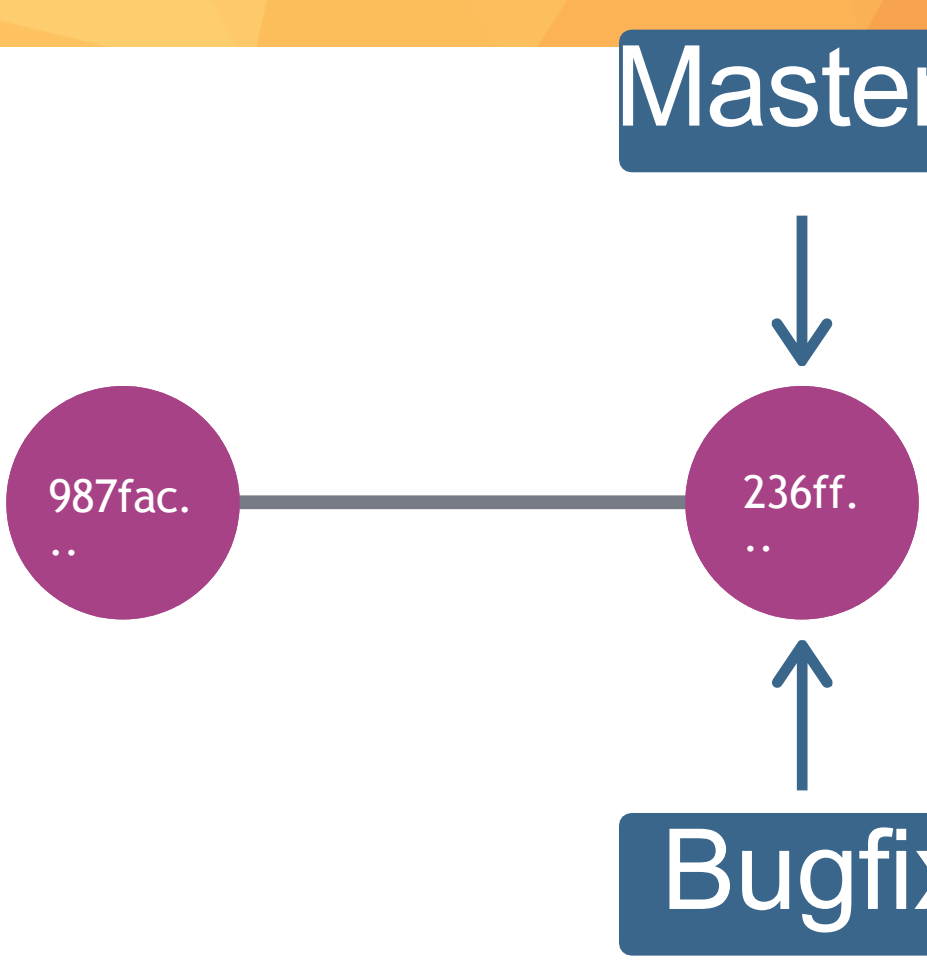
```
$ git branch
```

Creating Branches

- Use git branch <branch-name> to make a new branch based upon the current HEAD
-
- This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

```
$ git branch <branch-name>
```


Master
HEAD
987fac.
..
236ff.
..




```
$ git branch bugfix
```

I've made a new branch,
but I'm still working on master

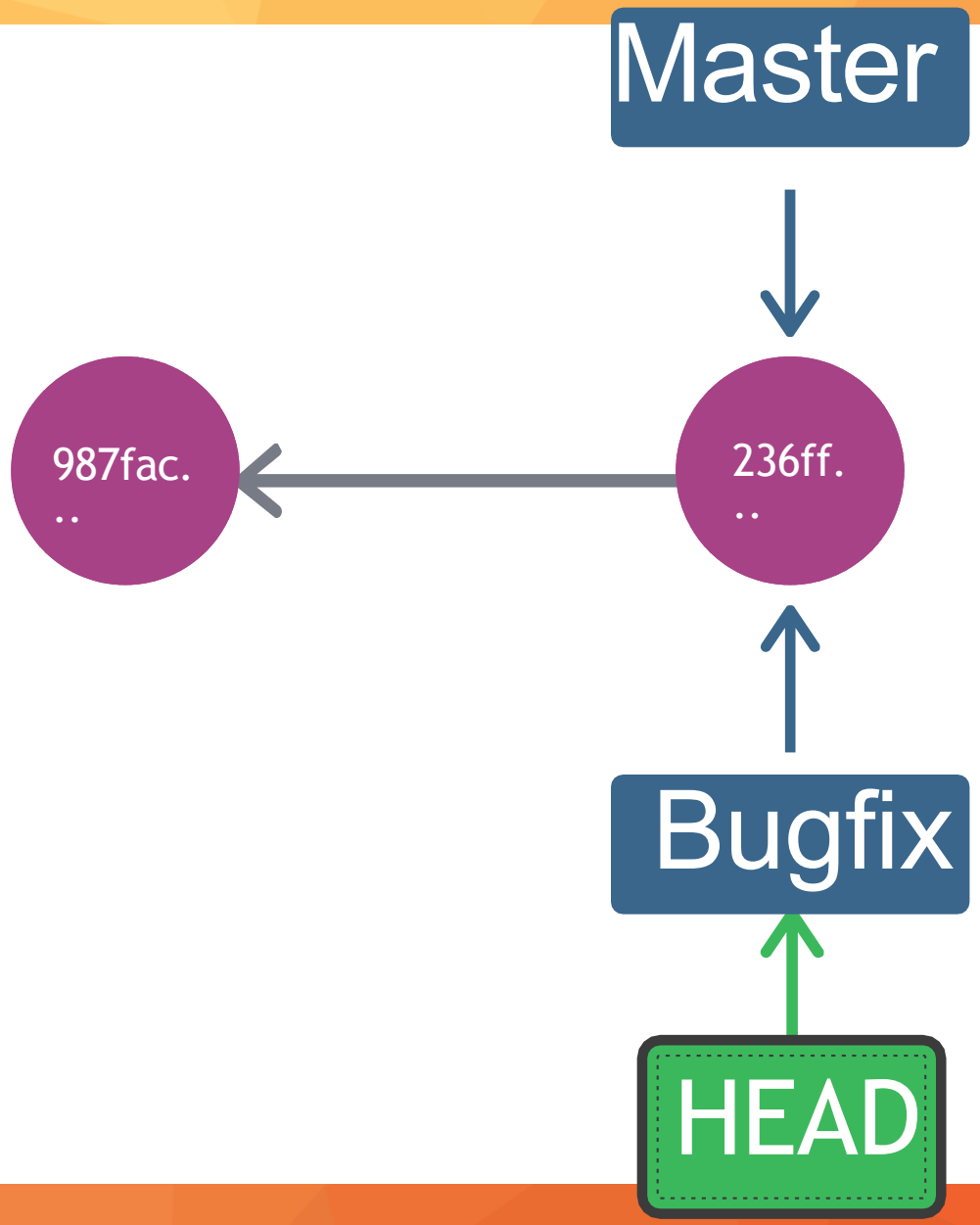
HEAD

- We'll often come across the term HEAD in Git.
- HEAD is simply a pointer that refers to the current "location" in your repository. It points to a particular branch reference.
- So far, HEAD always points to the latest commit you made on the master branch, but soon we'll see that we can move around, and HEAD will change!

Switching Branches

- Once you have created a new branch,
- use git switch <branch-name> to switch to it.

```
$ git switch <branch-name>
```

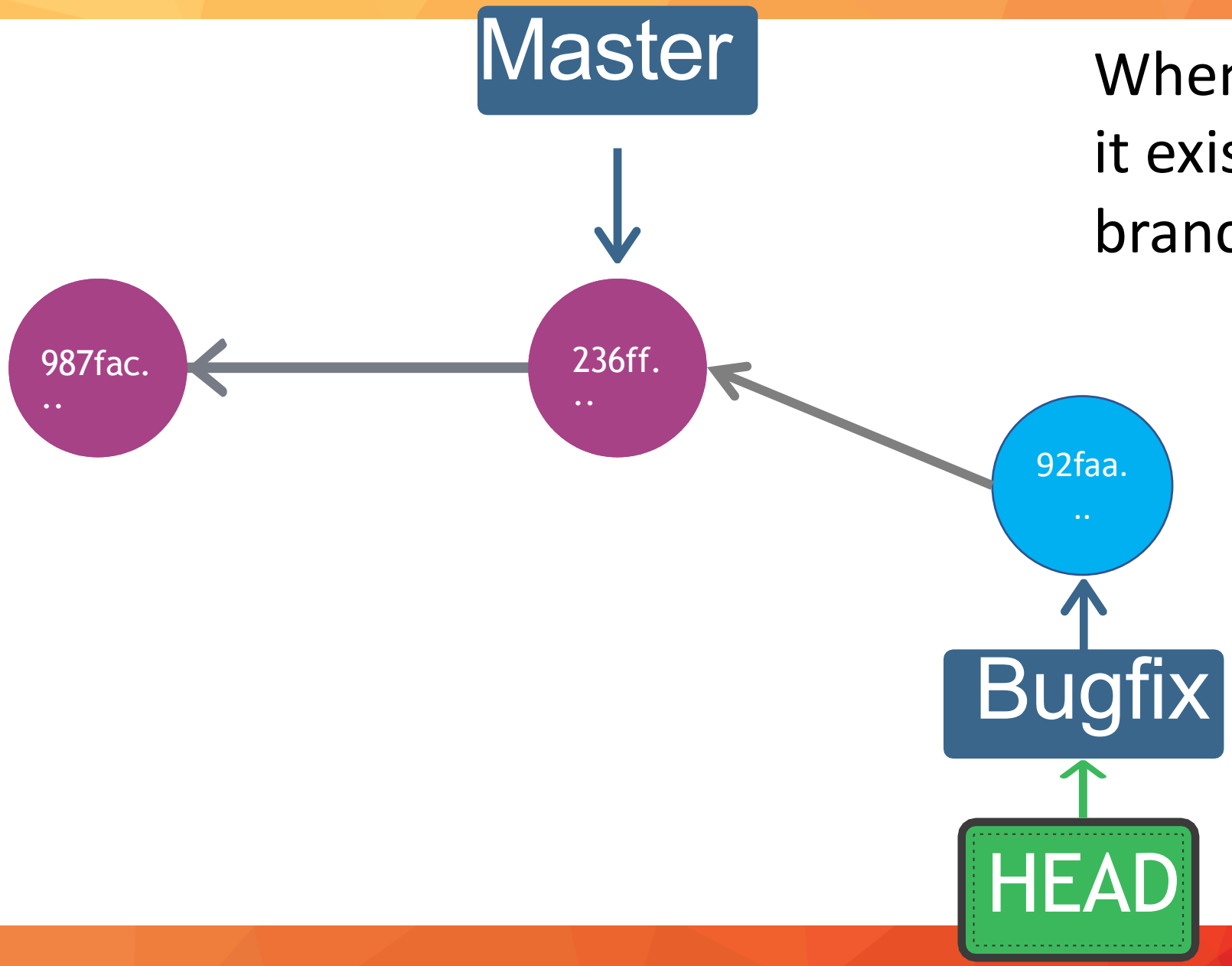


$ git switch Bugfix

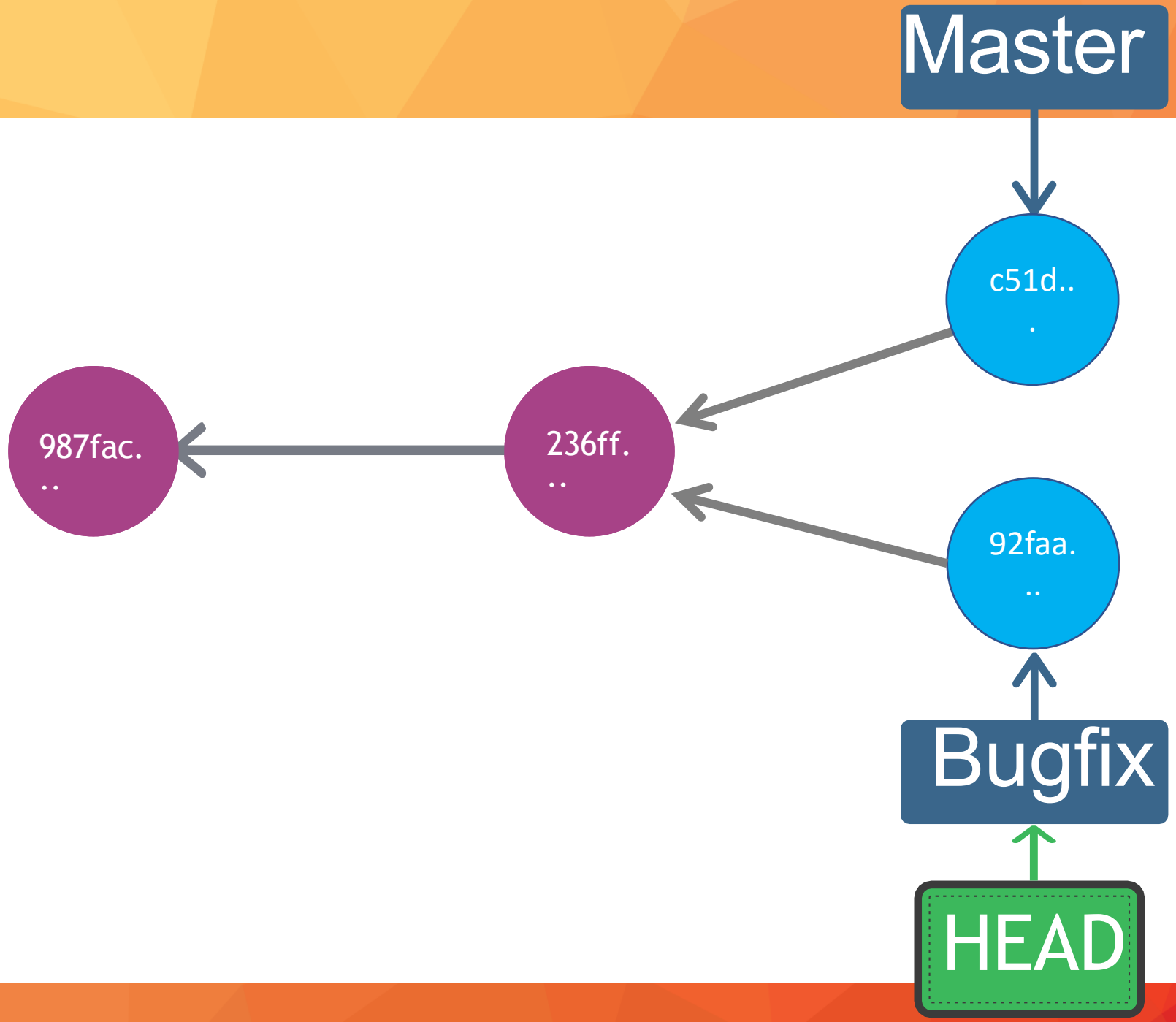
I've now switched over to the new bugfix branch

Notice that HEAD is now pointing to Bugfix, not master

TRAINOCATE



When I make a new commit, it exists only on my new branch, not on master



Divergent history

Another way of switching

- Historically, we used **git checkout <branch-name>** to switch branches. This still works.

```
$ git checkout <branch-name>
```

- The checkout command does a million additional things, so the decision was made to add a standalone switch command which is much simpler.

- You will see older tutorials and docs using checkout rather than switch. Both now work.

Creating And Switching

- Use git switch with the -c flag to create a new branch AND switch to it all in one go.

```
$ git switch -c <branch-name>
```

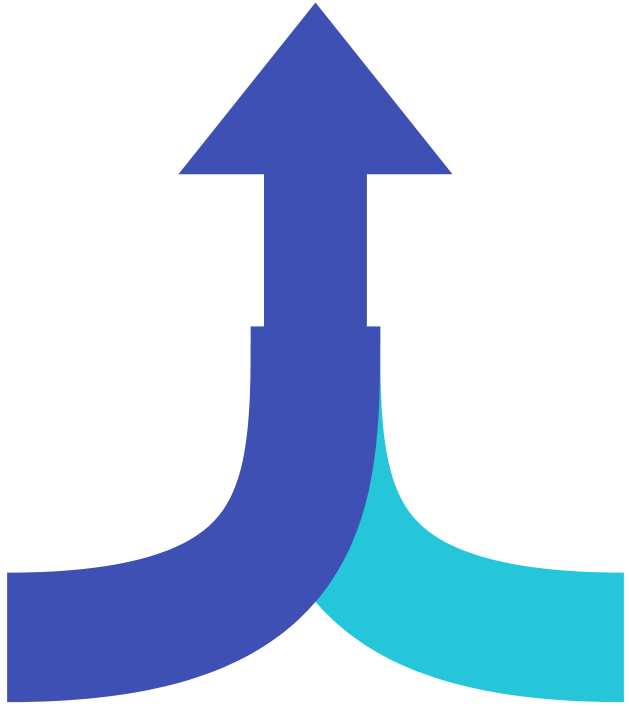
- Remember -c as short for "create"

Merging branches

Merging

- Branching makes it super easy to work within self- contained contexts, but often we want to incorporate changes from one branch into another!

- We can do this using the git merge command



Merging

The merge command can sometimes confuse students early on. Remember these two merging concepts:

- We merge branches, not specific commits
- We always merge to the current HEAD branch

Git merge

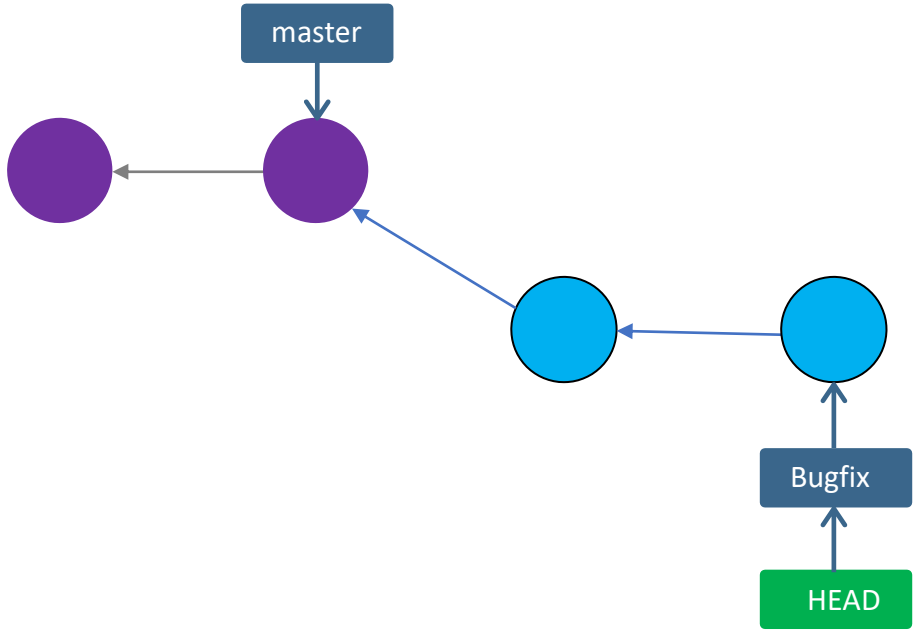
To merge, follow these basic steps:

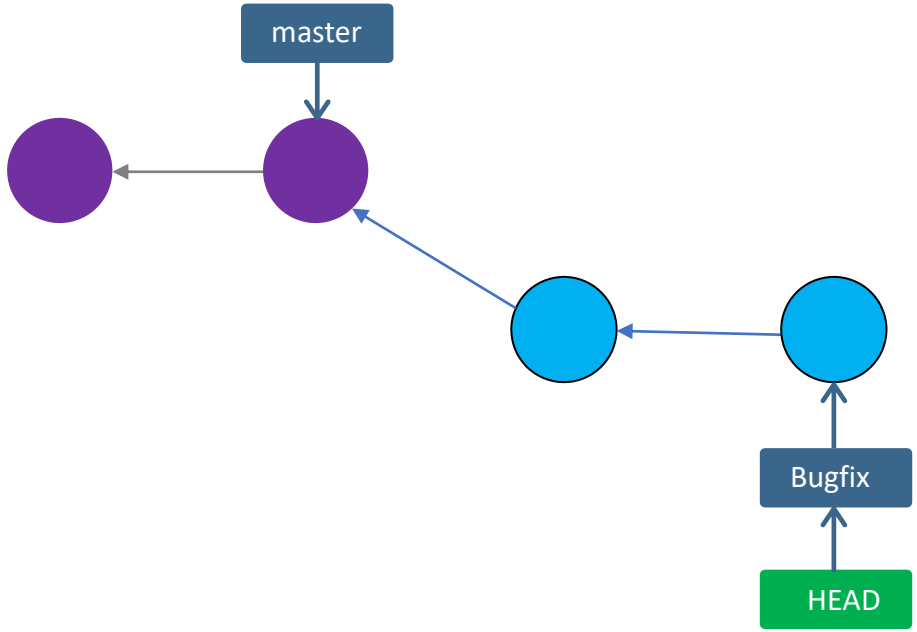
- Switch to or checkout the branch you want to merge the changes into (the receiving branch)
- Use the git merge command to merge changes from a specific branch into the current branch

To merge the Bugfix branch into master

```
$ git switch master
$ git merge Bugfix
```

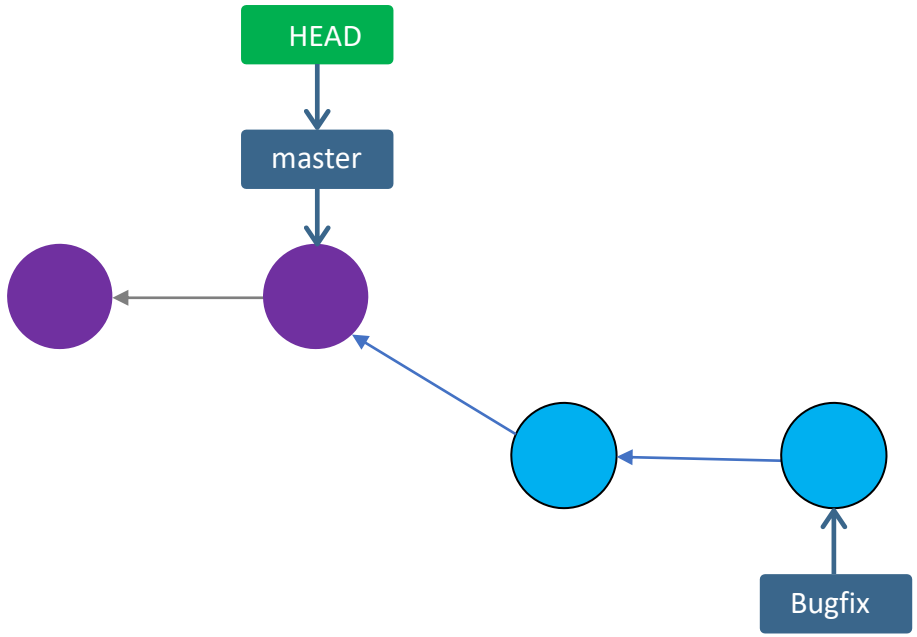
To merge the Bugfix branch into master branch





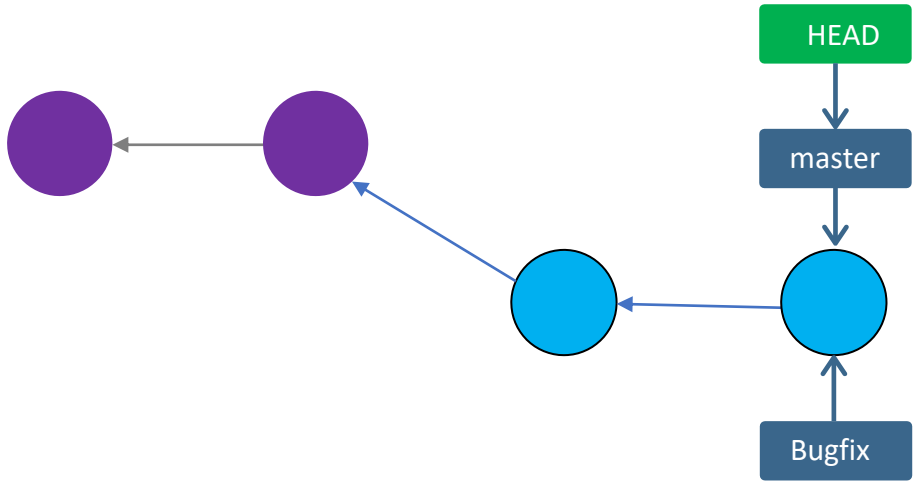

```
$ git switch master
```

Switch to master branch




```
$ git switch master
```

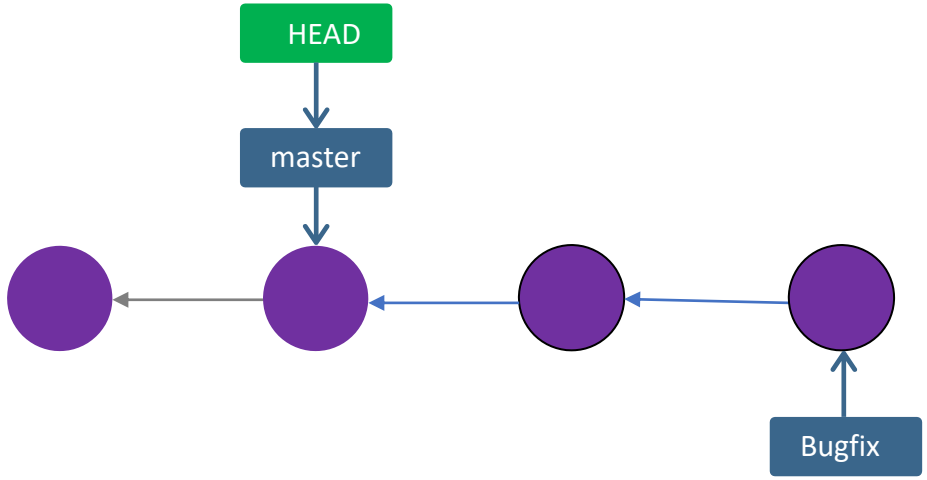
Switch to master branch



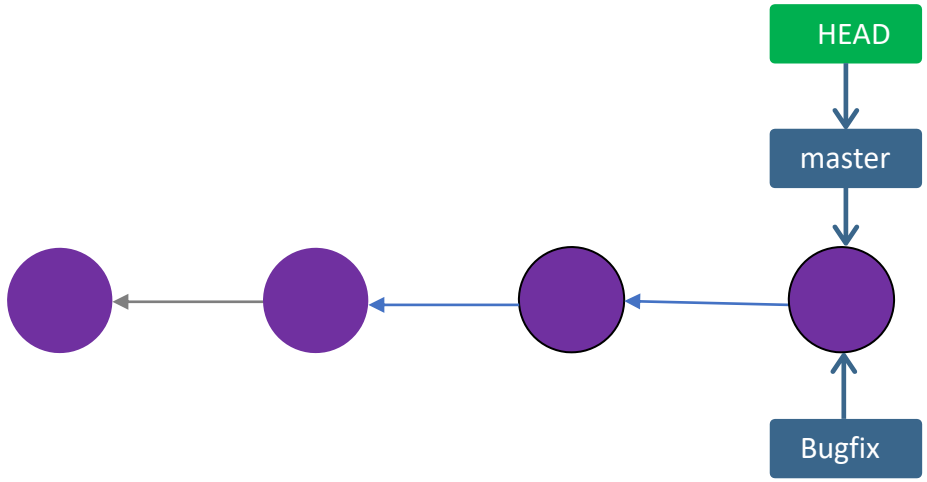

```
$ git merge Bugfix
```

Merge Bugfix into master

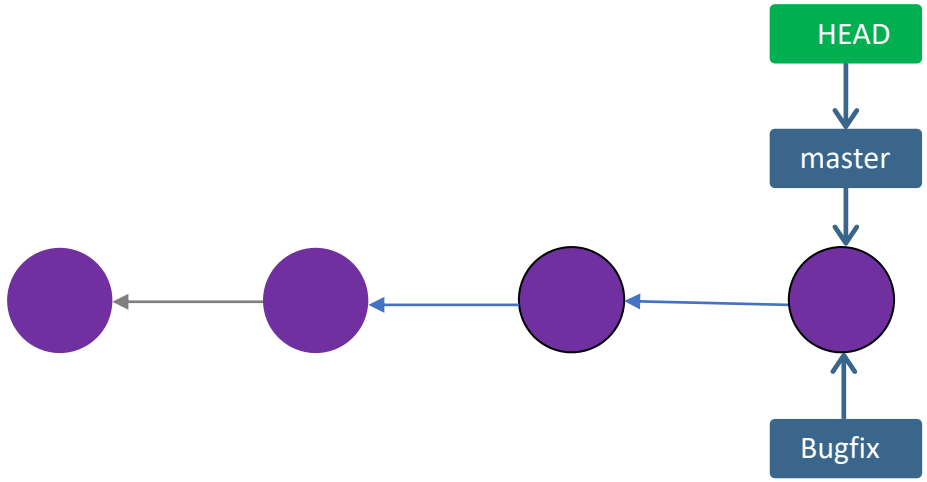
This is what it really looks like



This is what it really looks like

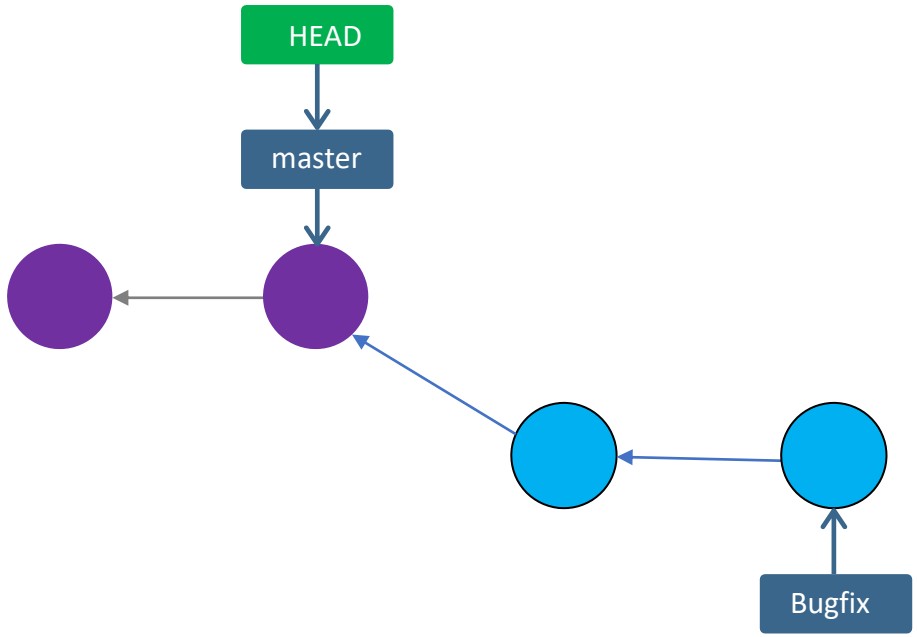


This Is Called A Fast-Forward

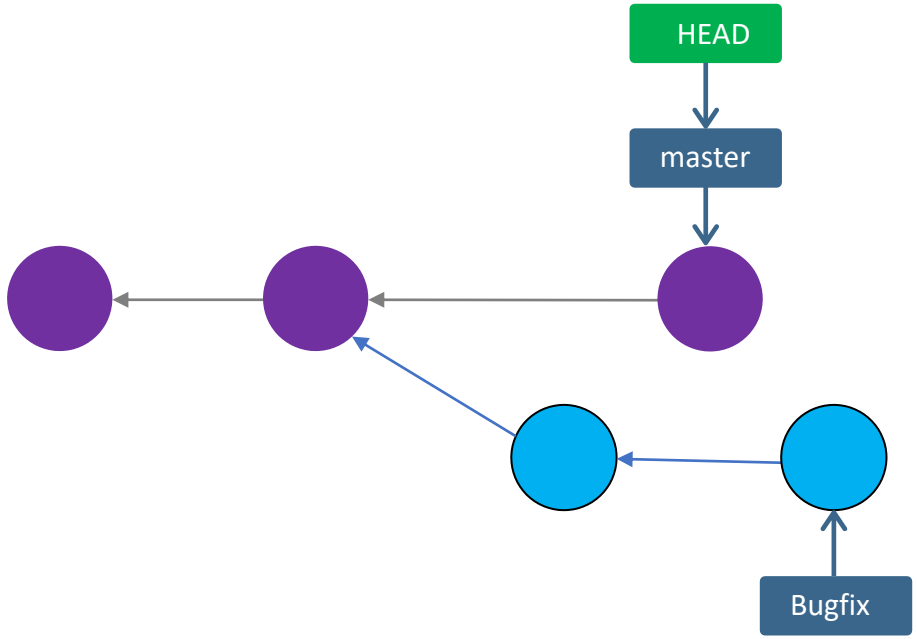


Master simply caught up on the commits from Bugfix

Not All Merges Are Fast-Forward

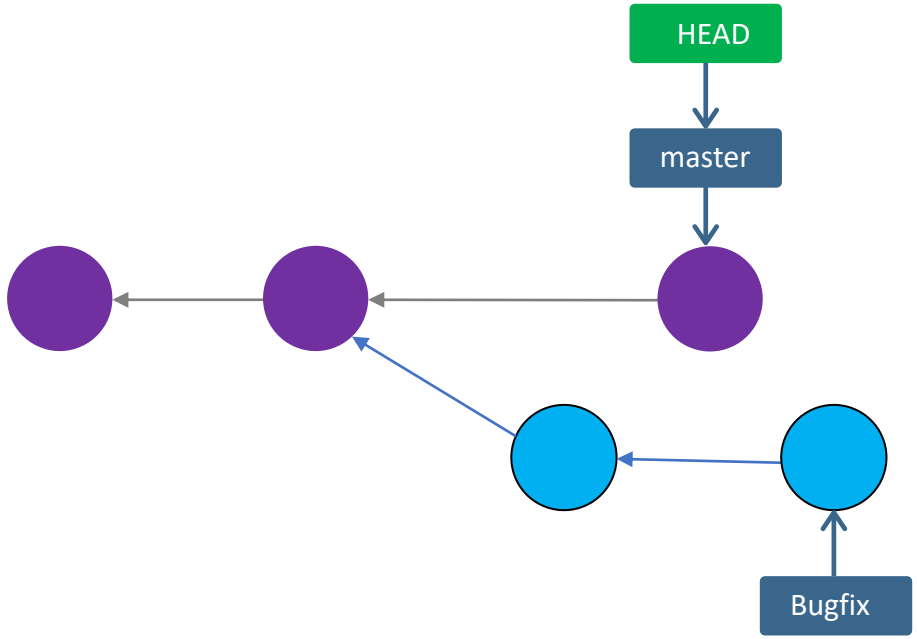


What if we add a commit on master?



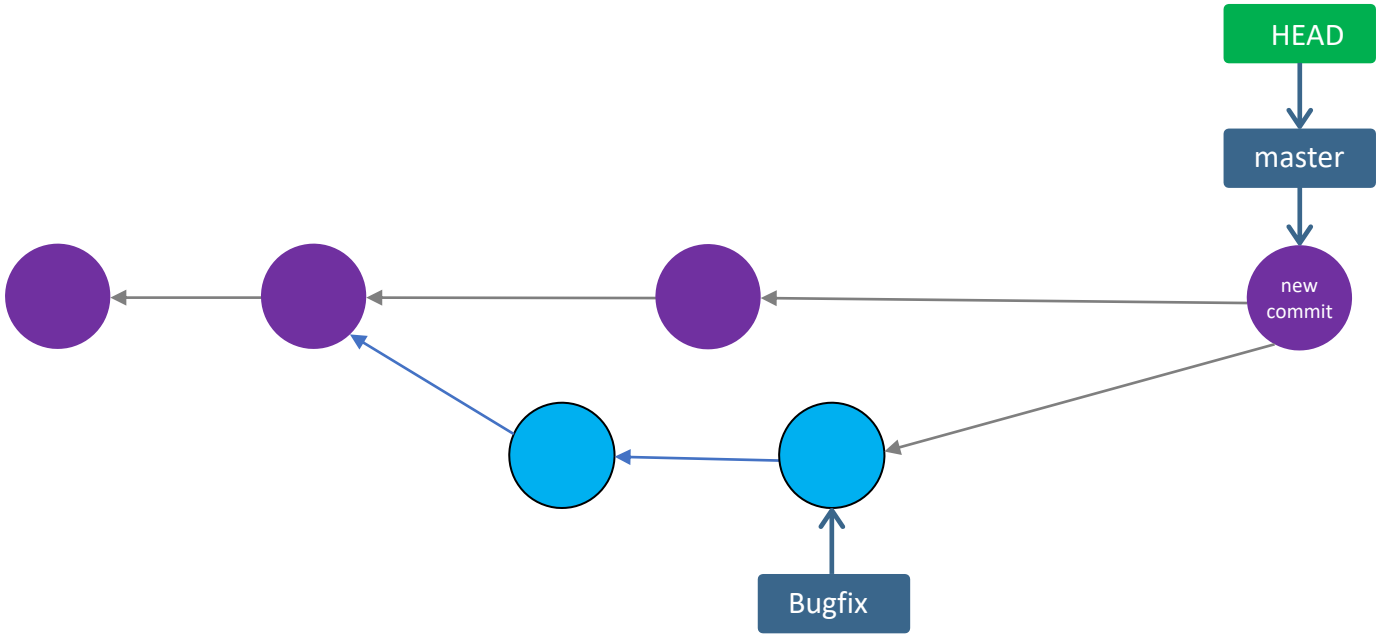
This happens all the time! Imagine one of your teammates merged in a new feature or change to master while you were working on a branch

What happens when I try to merge?



This happens all the time! Imagine one of your teammates merged in a new feature or change to master while you were working on a branch

What happens when I try to merge?



Rather than performing a simple fast forward, git performs a "merge commit"
We end up with a new commit on the master branch.

Git will prompt you for a message.

Merge Conflicts

- Depending on the specific changes you are trying to merge, Git may not be able to automatically merge.
- This results in merge conflicts, which you need to manually resolve.

Merge Conflict

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have a dog :(
>>>>>>> bug-fix
```

- When you encounter a merge conflict, Git warns you in the console that it could not automatically merge.
- It also changes the contents of your files to indicate the conflicts that it wants you to resolve.

Conflict Markers

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have a dog :(
>>>>>>> bug-fix
```

- The content from your current HEAD (the branch you are trying to merge content into) is displayed between the <<<<<<< HEAD and =======

Conflict Markers

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have a dog :(
>>>>>>> bug-fix
```

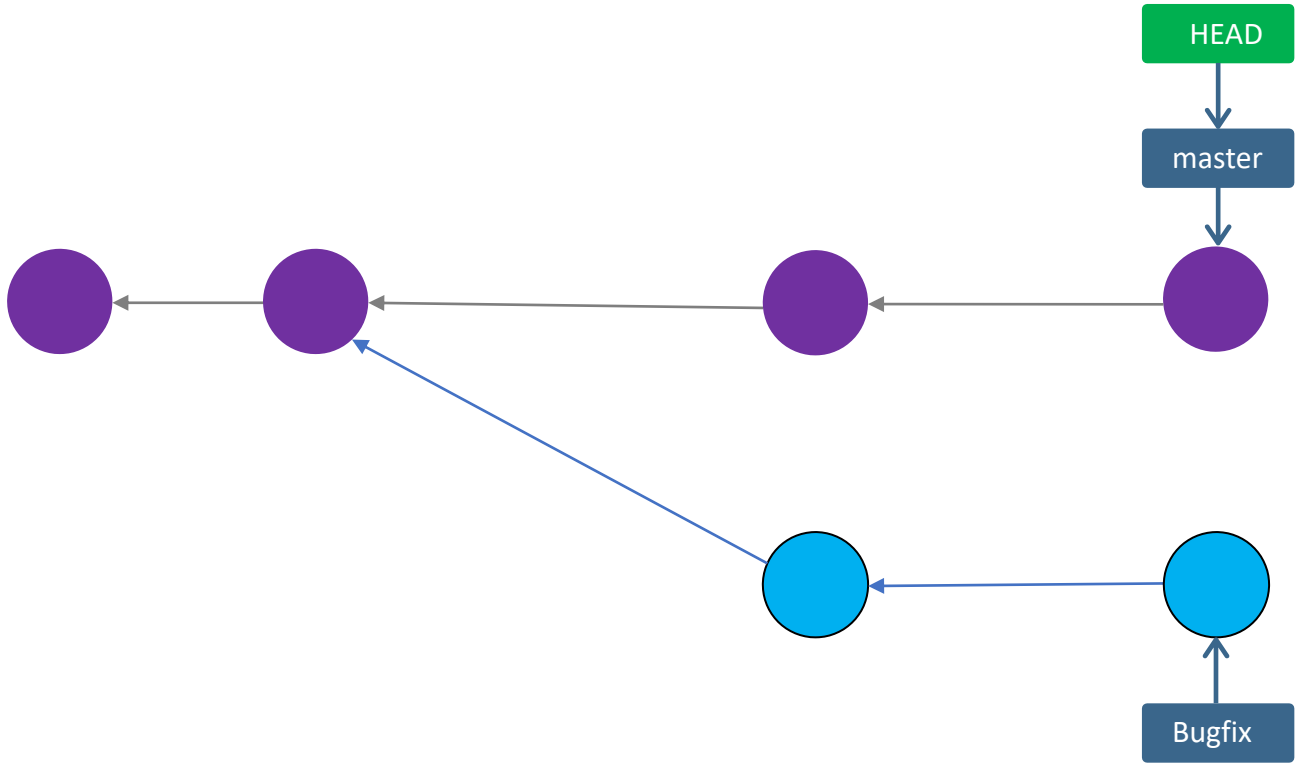
The content from the branch you are trying to merge from is displayed between the ======= and >>>>>>> symbols.

Resolving Conflicts

Whenever you encounter merge conflicts, follow these steps to resolve them:

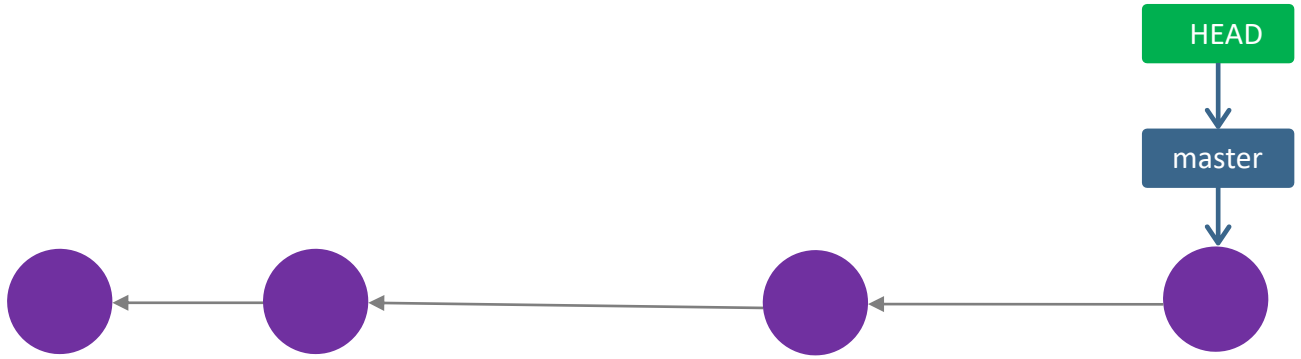
1. Open up the file(s) with merge conflicts
2. Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
3. Remove the conflict "markers" in the document
4. Add your changes and then make a commit

Deleting A Branch




```
$ git branch -d Bugfix
```

Deleting A Branch




```
$ git branch -d Bugfix
```

Undoing Things

Unstaging a Staged File With Git Restore

- If you have accidentally added a file to your staging area with git add and you don't wish to include it in the next commit, you can use git restore to remove it from staging.

- Use the --staged option like this: git restore --staged app.py

```
$ git restore  --staged <file-name>
```

Unmodifying A Modified File With Git Restore

- It's important to understand that git restore -- <file> is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes, anything you lose that was never committed is likely never to be seen again.

- If you would like to keep the changes you've made to that file but still need to get it out of the way for now, Git Branching is a generally better way to go.

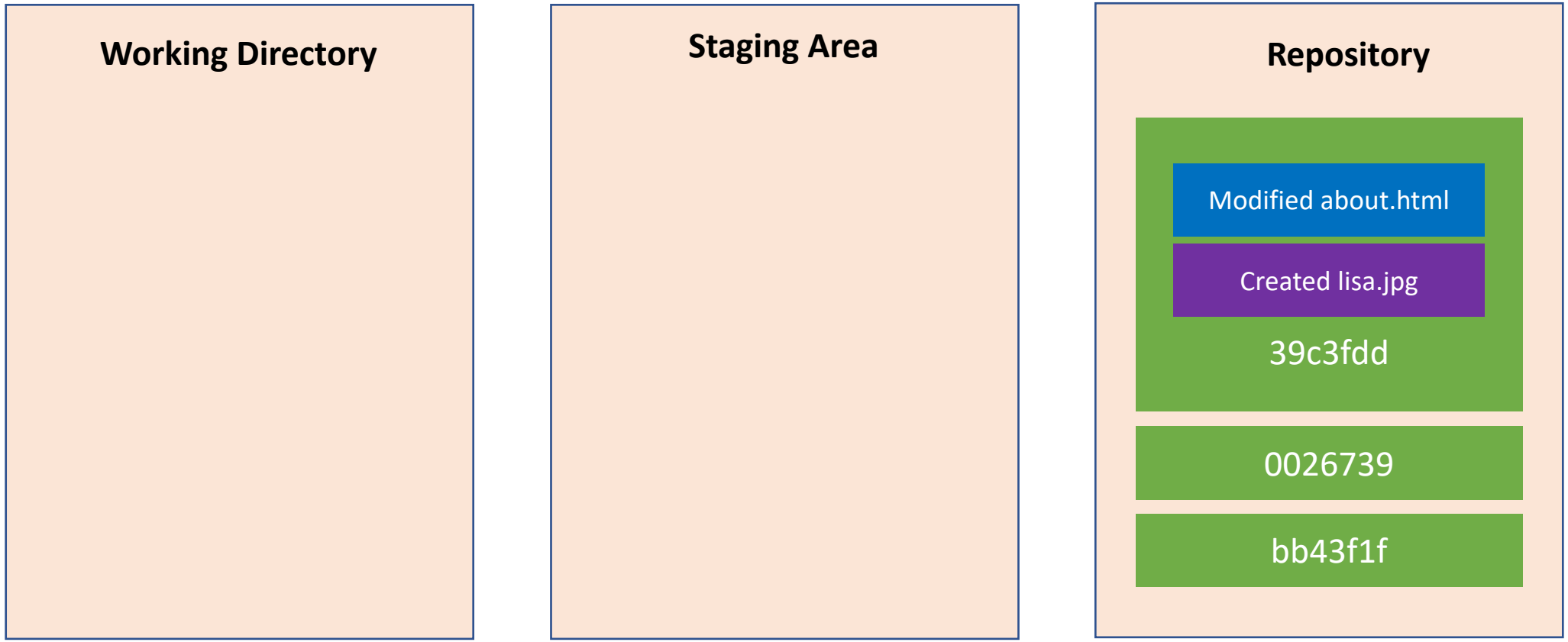
```
$ git restore <file-name>
```

Git Reset

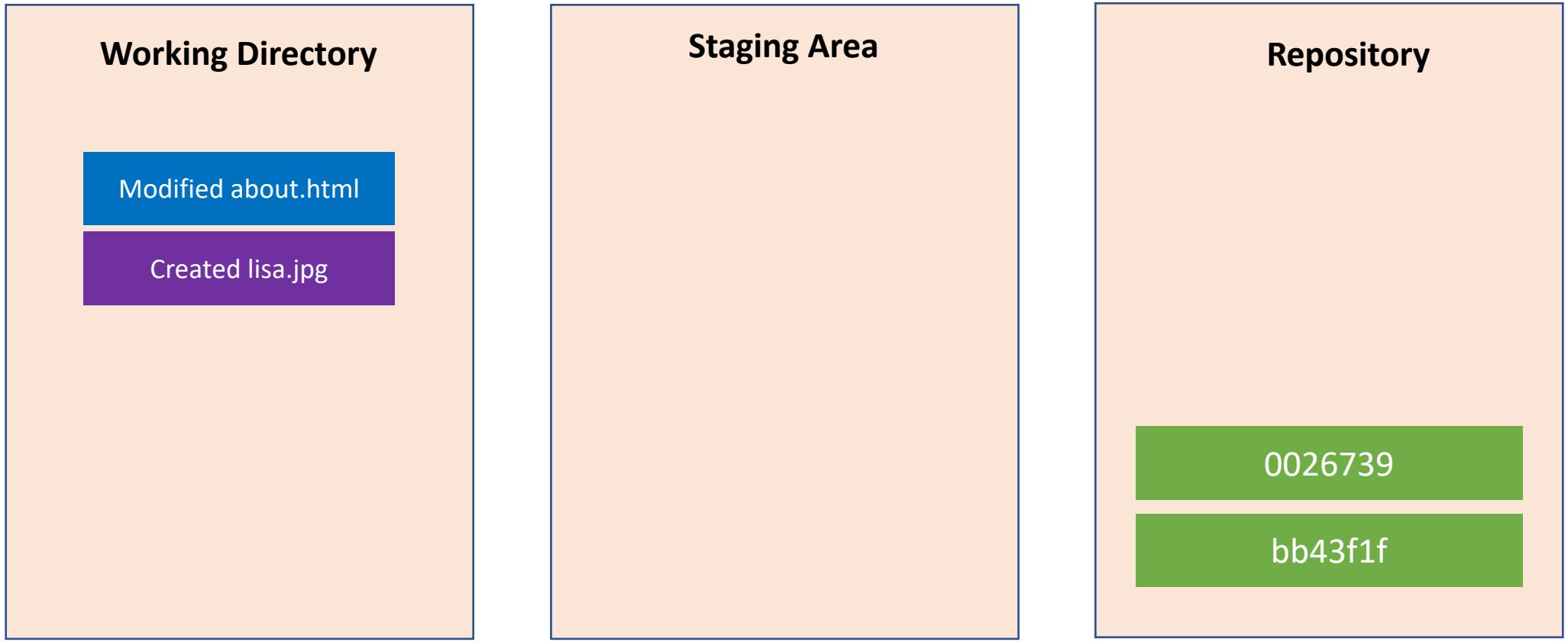
- Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commits, you can use git reset.

- git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone

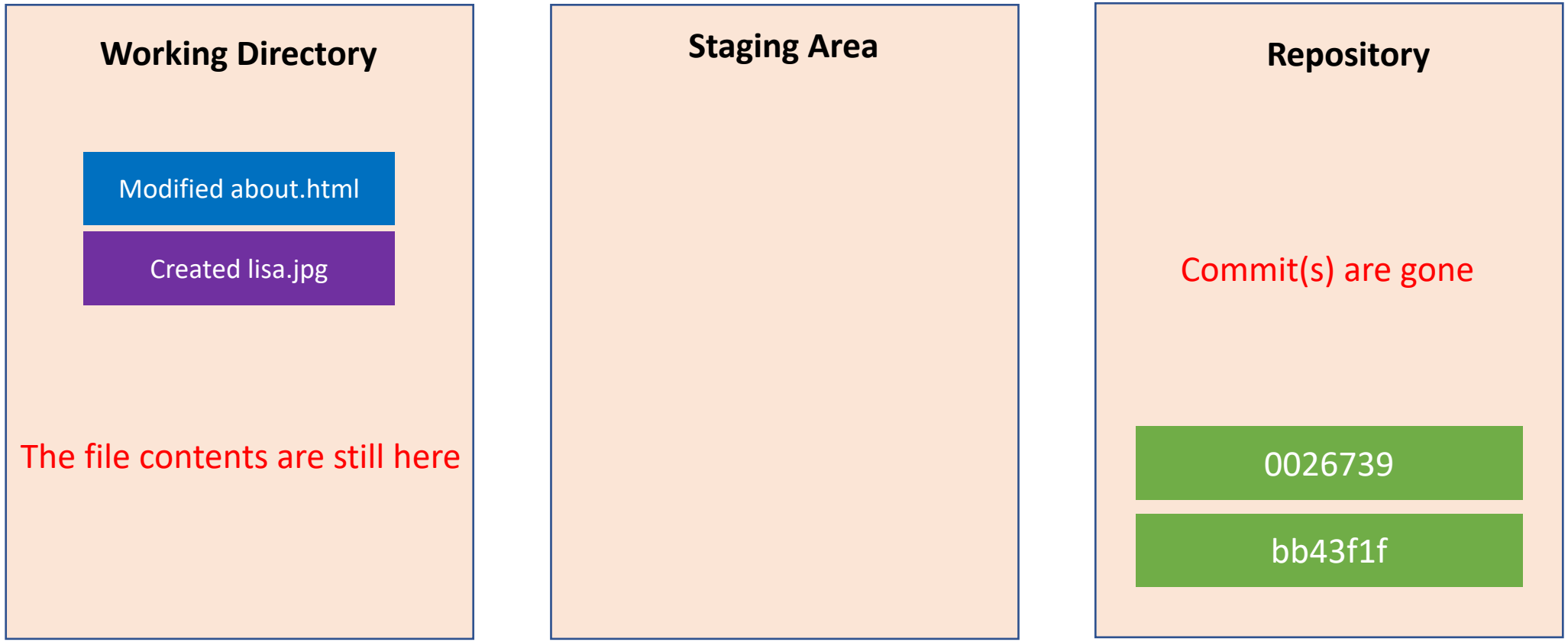
```
$ git reset --soft/mixed/hard  <commit-hash>
```

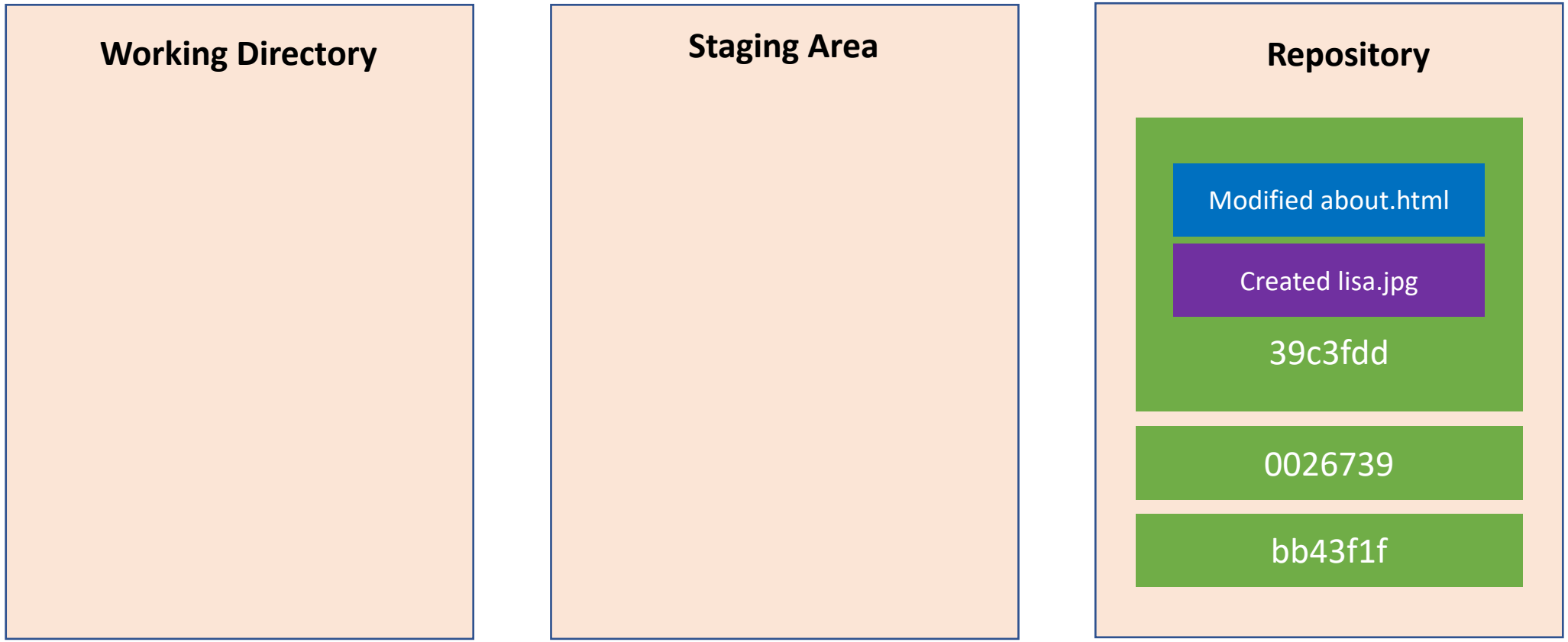

Working Directory
Staging Area
Repository
Modified about.html
Created lisa.jpg
39c3fdd
0026739
bb43f1f

```
$ git reset --mixed  0026739
$ git reset  0026739
```

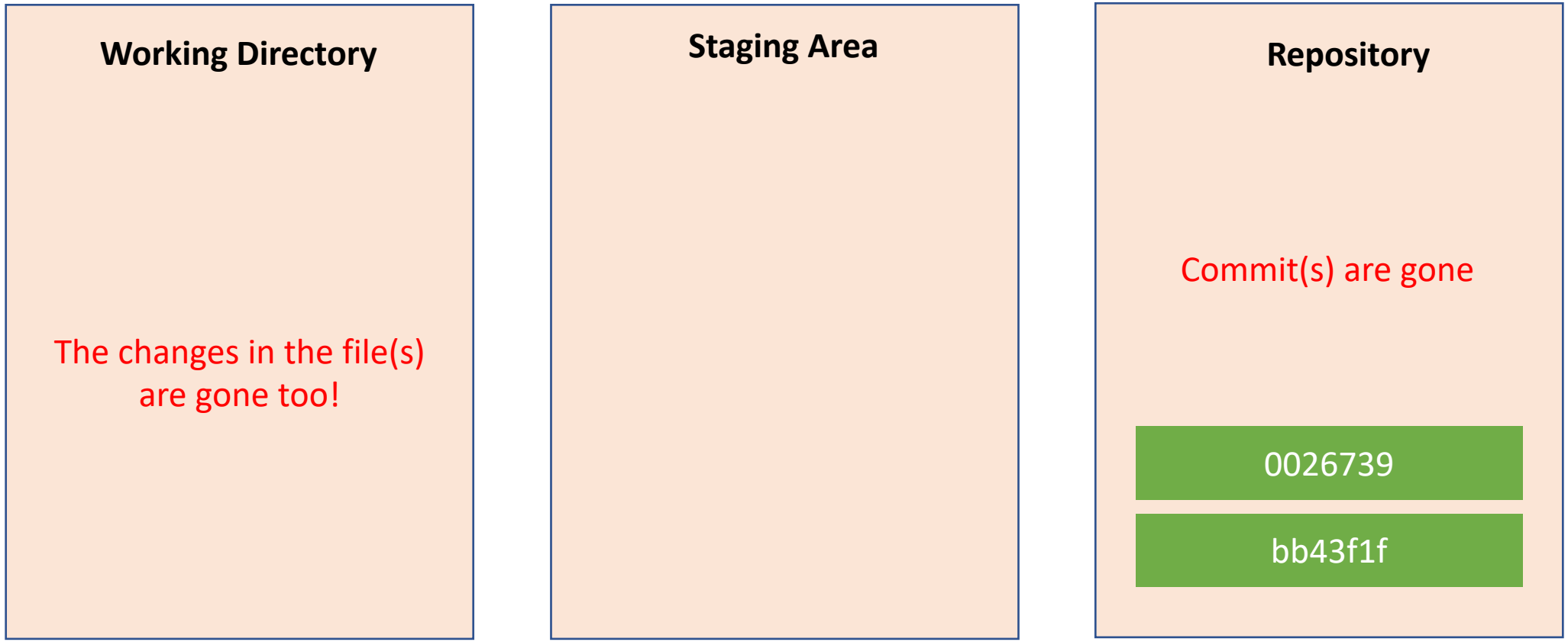


```
$ git reset --mixed  0026739
$ git reset  0026739
```

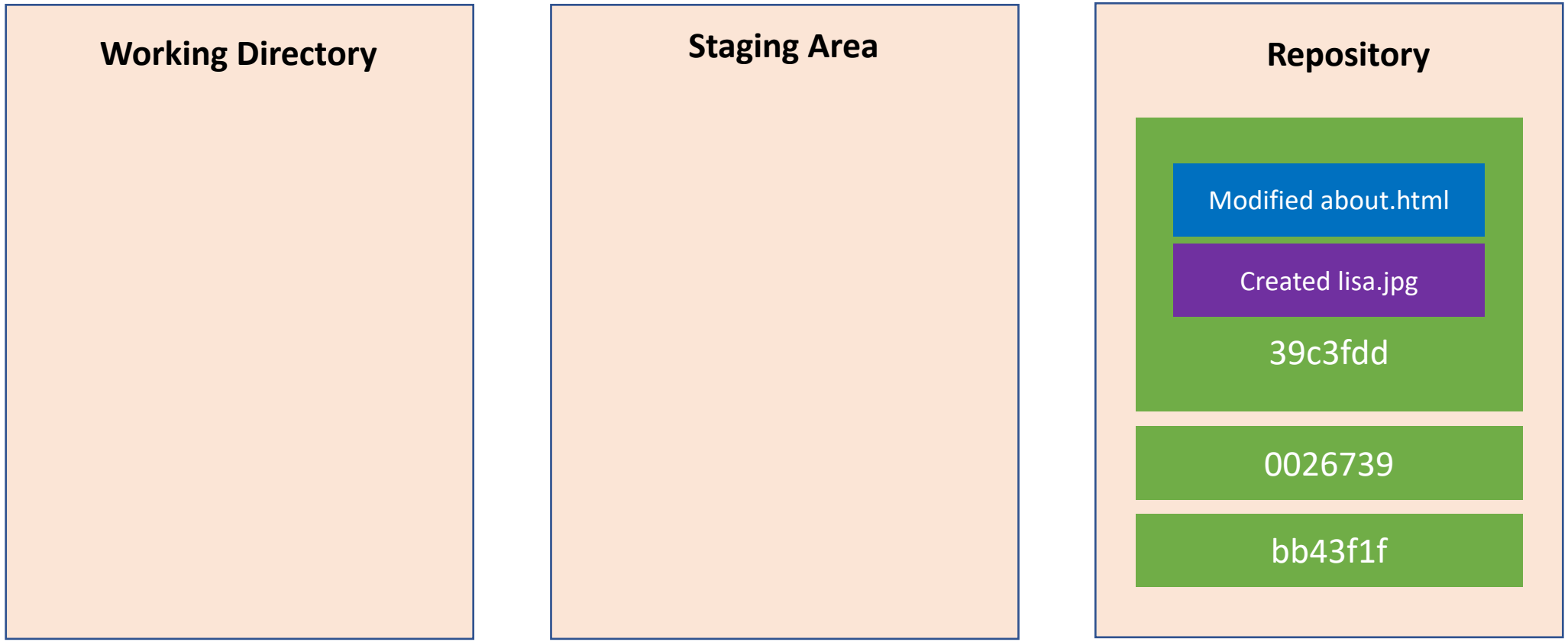



Working Directory
Staging Area
Repository
Modified about.html
Created lisa.jpg
39c3fdd
0026739
bb43f1f

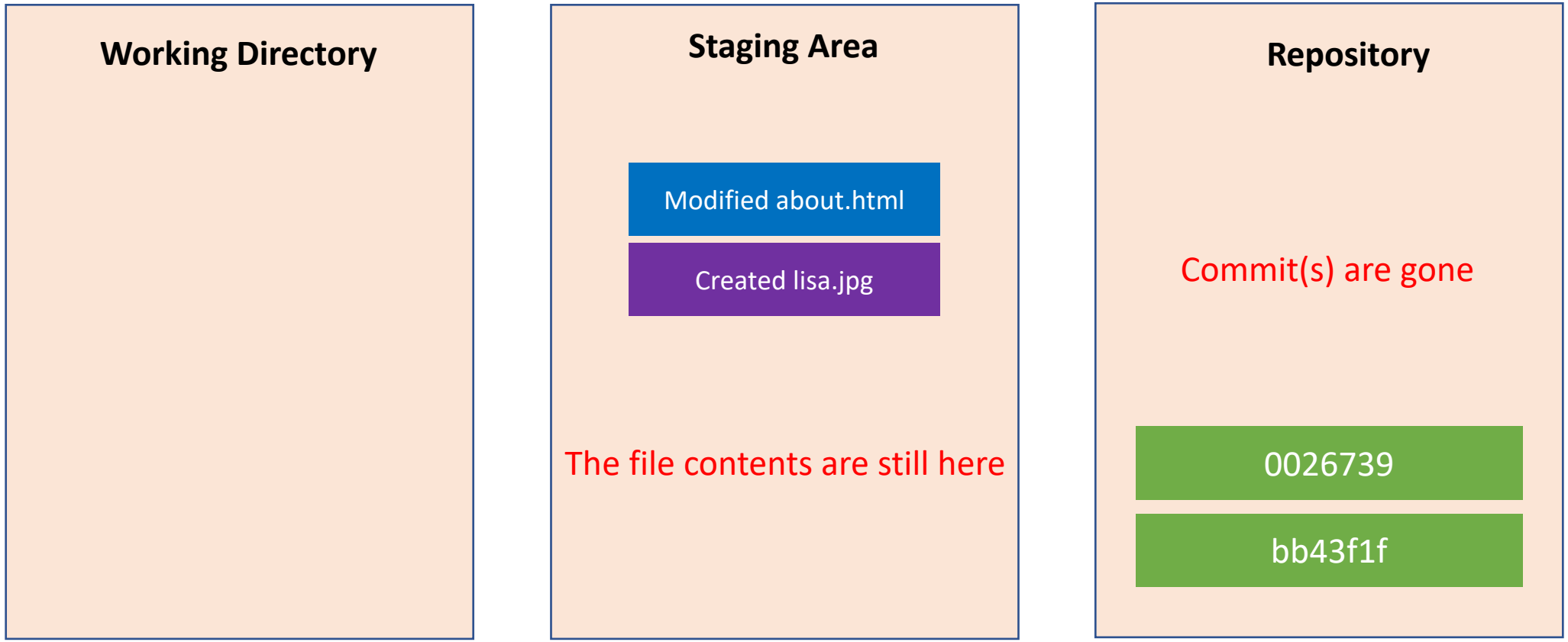
$ git reset --hard 0026739



If you want to undo both the commits AND the actual changes in your files, you can use the --hard option.


Working Directory
Staging Area
Repository
Modified about.html
Created lisa.jpg
39c3fdd
0026739
bb43f1f

$ git reset --soft 0026739

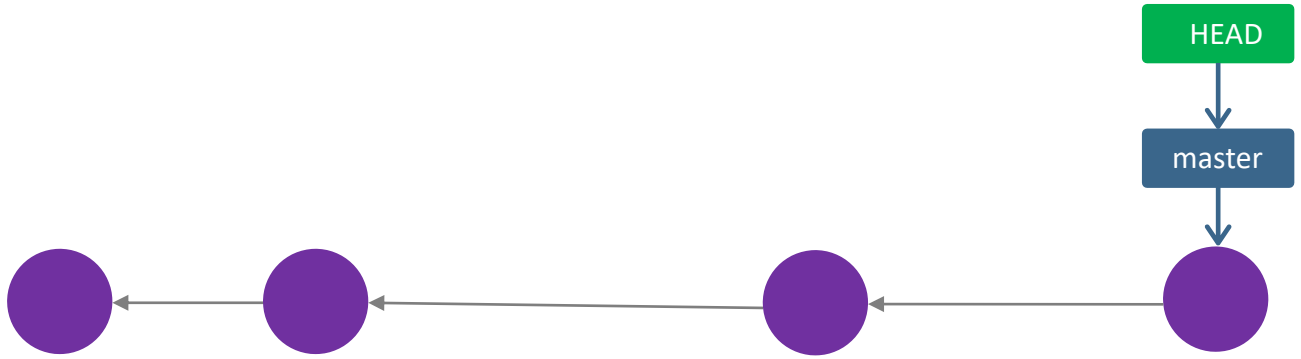


Git revert

- git revert is similar to git reset in that they both "undo" changes, but they accomplish it in different ways.
- git reset moves the branch pointer backwards, eliminating commits.
- git revert instead creates a brand-new commit which reverses/undos the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

```
$ git revert  <commit-hash>
```

“Undoing” With Reset



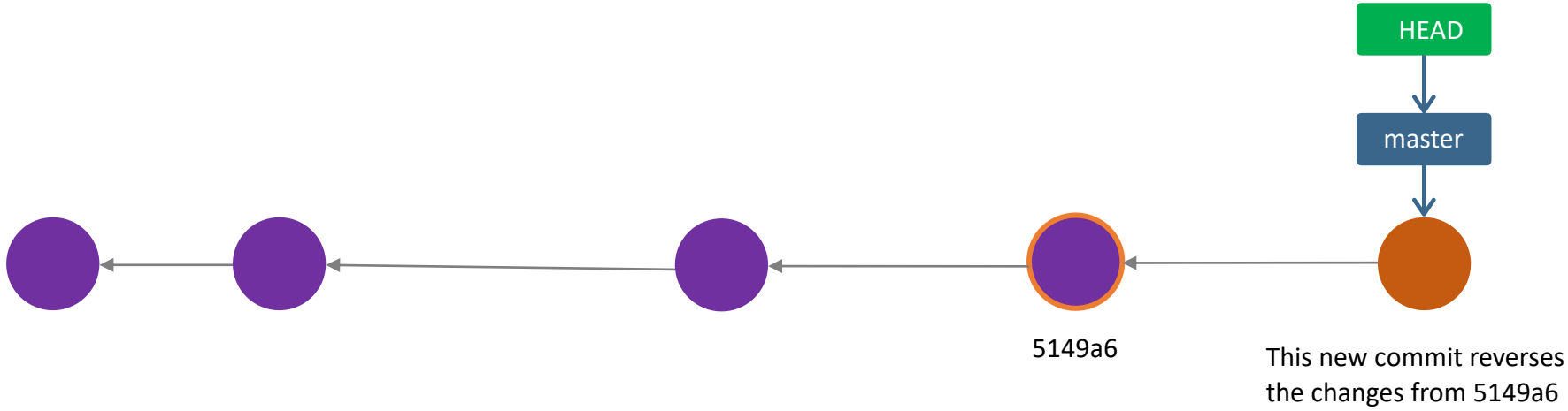
TRAINOCATE

“Undoing” With Reset


```
$ git reset  HEAD~2
```

The branch pointer is moved back to an earlier commit, erasing the 2 later commits

“Undoing” With Revert




```
$ git revert 5149a6
```

Which One Should I Use

- Both git reset and git revert help us reverse changes, but there is a significant difference when it comes to collaboration (which we have yet to discuss but is coming up soon!)

- If you want to reverse some commits that other people already have on their machines, you should use revert.

- If you want to reverse commits that you haven't shared with others, use reset, and no one will ever know!

Github basic

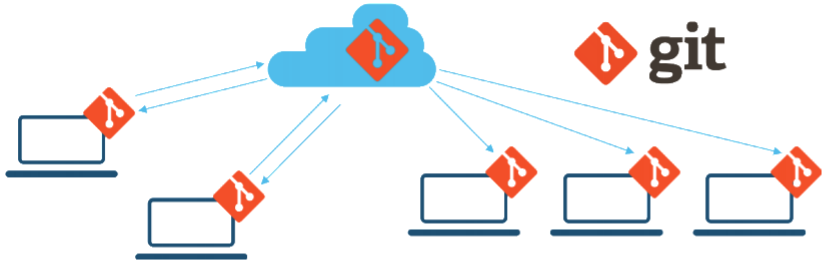
TRAINOCATE

What Is Github?

- Github is a hosting platform for git repositories. You can put your own Git repos on Github and access them from anywhere and share them with people around the world.

- Beyond hosting repos, Github also provides additional collaboration features that are not native to Git (but are super useful). Basically, Github helps people share and collaborate on repos

Git vs. GitHub



Git is an open source Distributed Version Control System

GitHub is a commercial company, that runs GitHub.com based on Git Version Control System

Connecting To GitHub With SSH

Checking for existing SSH keys

1. Open Terminal.
2. Enter `ls -al ~/.ssh` to see if existing SSH keys are present:

```
$ ls -al ~/.ssh
# Lists the files in your .ssh directory, if they exist
```

3. Check the directory listing to see if you already have a public SSH key. By default, the filenames of the public keys are one of the following:
   - *id_rsa.pub*
   - *id_ecdsa.pub*
   - *id_ed25519.pub*

https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/connecting-to-github-with-ssh

Generating a new SSH key

1. Open Terminal.
2. Paste the text below, substituting in your GitHub email address.

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```

> **Note:** If you are using a legacy system that doesn't support the Ed25519 algorithm, use:
>
> ```
> $ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
> ```

This creates a new ssh key, using the provided email as a label.

```
> Generating public/private ed25519 key pair.
```

3. When you're prompted to "Enter a file in which to save the key," press Enter. This accepts the default file location.

```
> Enter a file in which to save the key (/Users/you/.ssh/id_ed25519): [Press ent
```

4. At the prompt, type a secure passphrase. For more information, see "Working with SSH key passphrases".

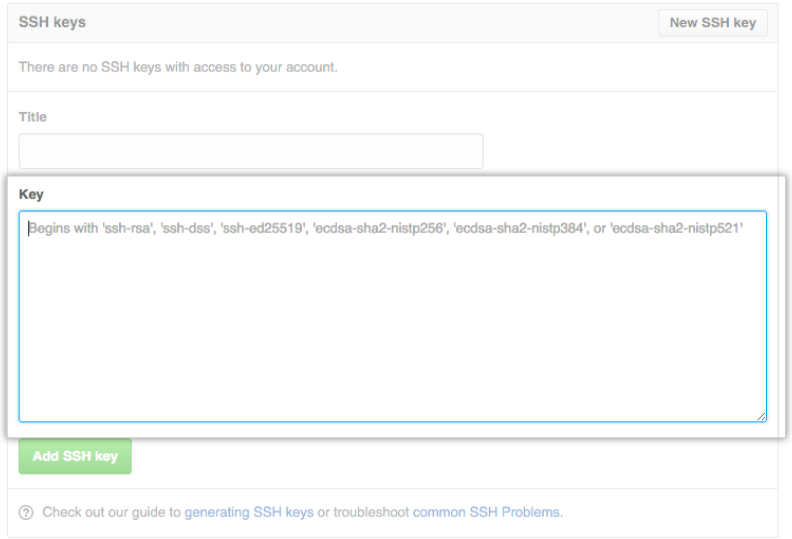
```
> Enter passphrase (empty for no passphrase): [Type a passphrase]
> Enter same passphrase again: [Type passphrase again]
```

Connecting To GitHub With SSH

Adding a new SSH key to your GitHub account

locate the hidden .ssh folder, open the file in your favorite text editor, and copy it to your clipboard.

```
osboxes@ubuntu:~/Documents/git/git-intro$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQCwWQ3t77WoGUH20oWCUE71f3AdWOdyaHFQVSS7S4
87wEylcB6hCN14G9KPt+8vXptVNFTyILvm9/vflOMapAKuYbO0yeW0LO9zDXLlS/dTafmgAeVhBsXy
yDCadHhcJGbsDWA+Bj4fMWkdAvJjLpDz8jJoDiCjxwVbZUAa8xFOkrtMoYl8ik4L0lVztk30IqD7RH
jTBRnSSk4AjQD+5udON1ZZNhYd+cUUE1rY5KMdw4AUXpaInno1Vee3YJWtmM3XY0gduKjLddmSxNY5
/d2u76oasGOxMv9UGWNybX9larkLDLbx6DKgWLB0NQ3ddQoT2VH5LgvKBz3CjmAcuP4axYLmt/Y6yI
kyXcgQldHsrEKLYpLj7Joxe7+N2PyUzHAhXP50fy7a/8WI+laJPgaoPjho2KPky/v1bSN2tutrHRjT
kV6uLCFtFwkIUUirswLly/UenC+0Hy5J/nxfUw== sinhhoang210@gmail.com
osboxes@ubuntu:~/Documents/git/git-intro$
```



TRAINOCATE

Connecting To GitHub With SSH

1. Open Terminal.
2. Enter the following:

```
$ ssh -T git@github.com
# Attempts to ssh to GitHub
```

You may see a warning like this:

```
> The authenticity of host 'github.com (IP ADDRESS)' can't be established.
> RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
> Are you sure you want to continue connecting (yes/no)?
```

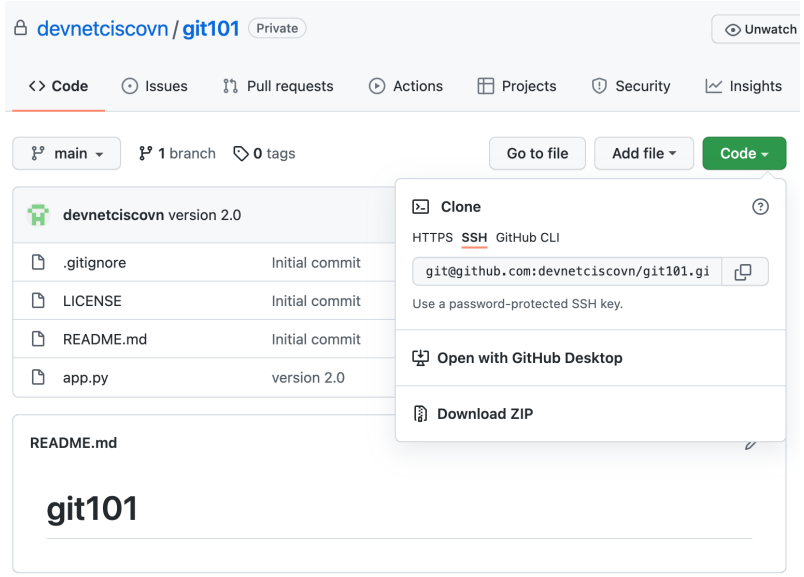
3. Verify that the fingerprint in the message you see matches one of the messages in step 2, then type `yes`:

```
> Hi username! You've successfully authenticated, but GitHub does not
> provide shell access.
```

```
$ ssh -T git@github.com
The authenticity of host 'github.com (20.205.243.166)' can't be established.
ECDSA key fingerprint is SHA256:p2QAMXNIC1TJYWeIOttrVc98/R1BUFWu3/LiyKgUfQM.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com,20.205.243.166' (ECDSA) to the list of known hosts.
Hi devnetciscovn! You've successfully authenticated, but GitHub does not provide shell access.
```

TRAINOCATE

Connecting to GitHub With SSH




```
$ ssh -T git@github.com
The authenticity of host 'github.com (20.205.243.166)' can't be established.
ECDSA key fingerprint is SHA256:p2QAMXNIC1TJYWeIOttrVc98/R1BUFWu3/LiyKgUfQM.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com,20.205.243.166' (ECDSA) to the list of known hosts.
Hi devnetciscovn! You've successfully authenticated, but GitHub does not provide shell access.
```

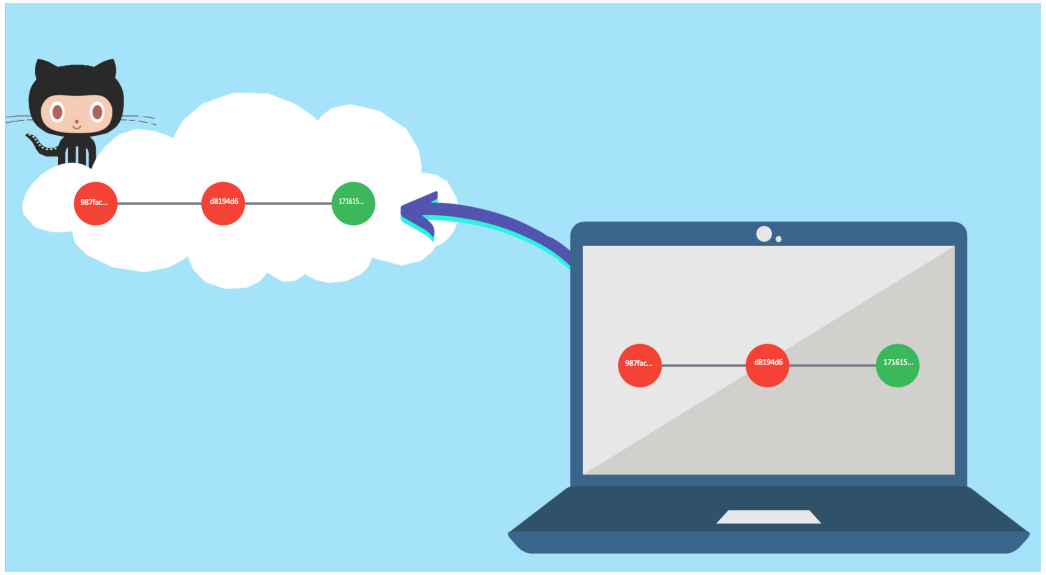
TRAINOCATE

How Do I Get My Code On Github?

Option 1: Existing Repo

If you already have an existing repo locally that you want to get on Github...

- Create a new empty repo on Github
- Copy the repo URL
- Add a remote to your local repo, using the UR
- Push your changes up to the remote



Git remote

- Before we can push anything up to Github, we need to tell Git about our remote repository on Github. We need to setup a "destination" to push up to.

- In Git, we refer to these "destinations" as remotes. Each remote is simply a URL where a hosted repository lives.

Viewing Remotes

- To view any existing remotes for your repository, we can run git remote or git remote -v (verbose, for more info)

- This just displays a list of remotes. If you haven't added any remotes yet, you won't see anything!

```
$ git remote  -v

$ git remote show <name>
```

Adding A New Remote

A remote is really two things: a URL and a label.

To add a new remote, we need to provide both to Git.

```
$ git remote  add <name> <url>
```

```
$ git remote add pb https://github.com/paulboone/ticgit
```

Renaming and Removing Remotes

```
$ git remote  rename <old> <new>
```

```
$ git remote  remove <name>
```

Origin

- Origin is a conventional Git remote name, but it is not at all special. It's just a name for a URL.
- When we clone a Github repo, the default remote name setup for us is called origin. You can change it. Most people leave it

```
$ git remote add origin git@github.com:devnetciscovn/git101.git
```

Pushing

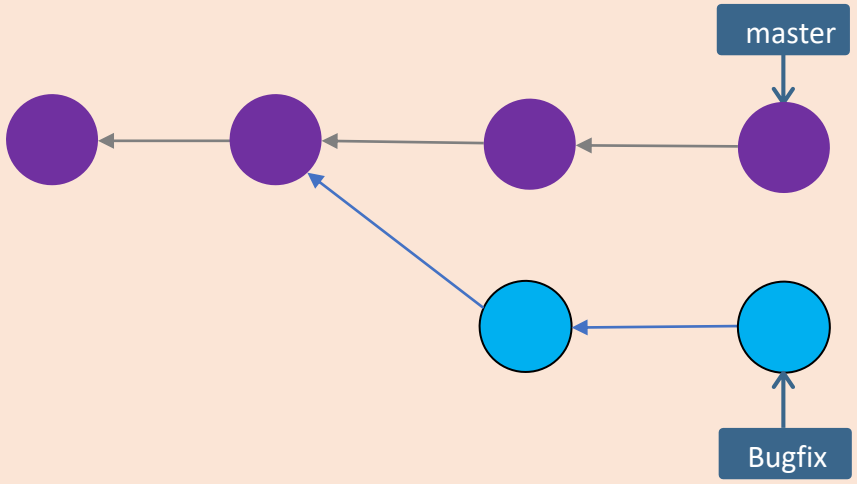
- Now that we have a remote set up, let's push some work up to Github! To do this, we need to use the git push command.

- We need to specify the remote we want to push up to AND the specific local branch we want to push up to that remote

```
$ git push <remote> <local-branch>
```

Pushing

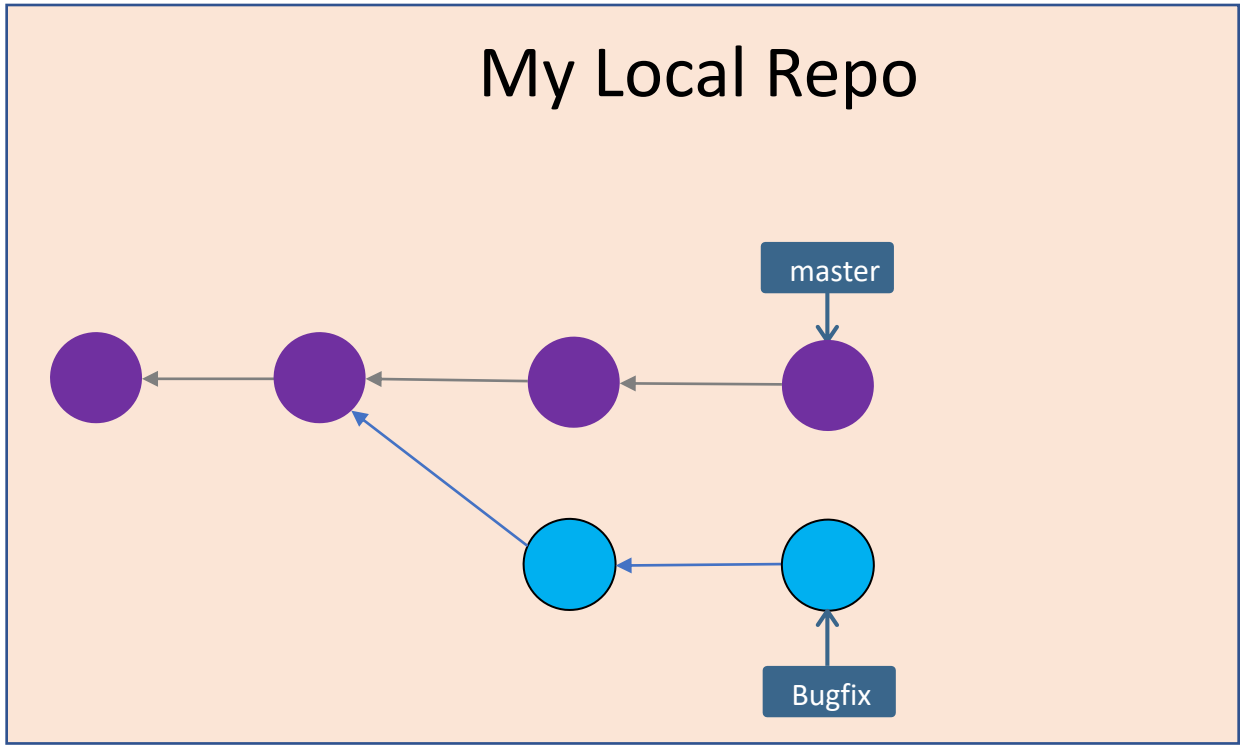
My Local Repo

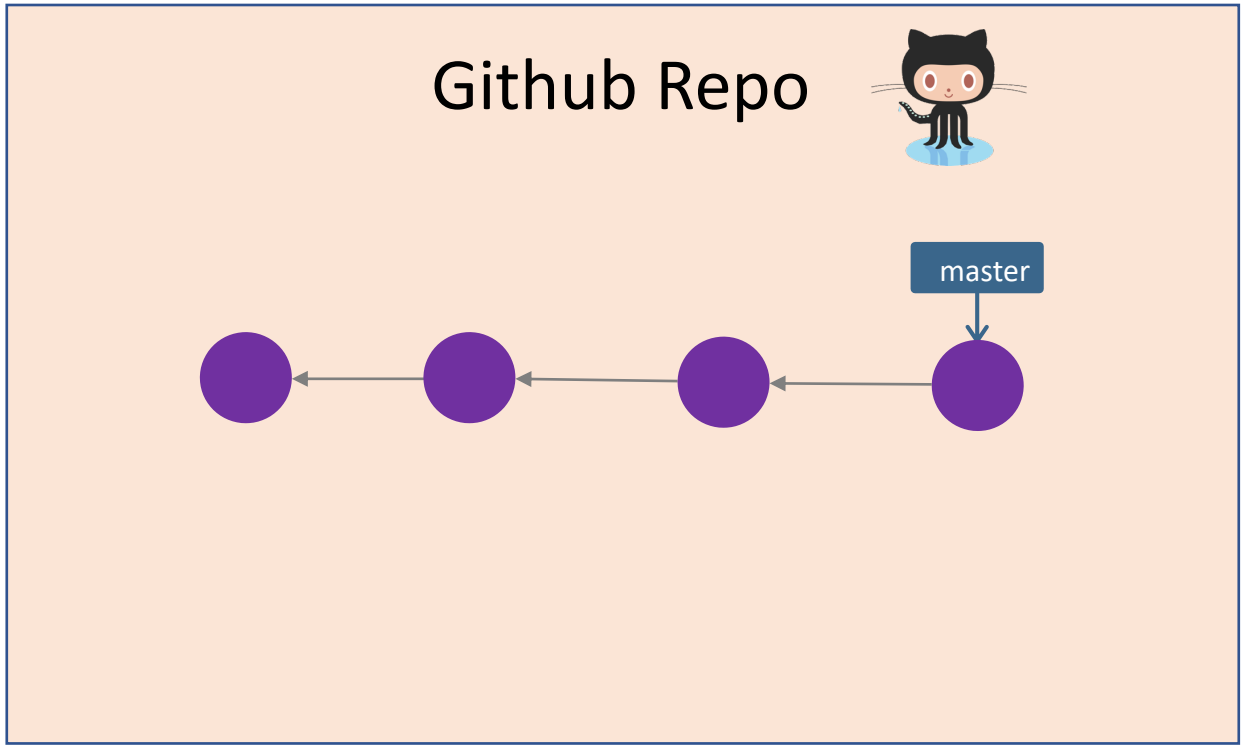


Github Repo

nothing to see here...

Pushing



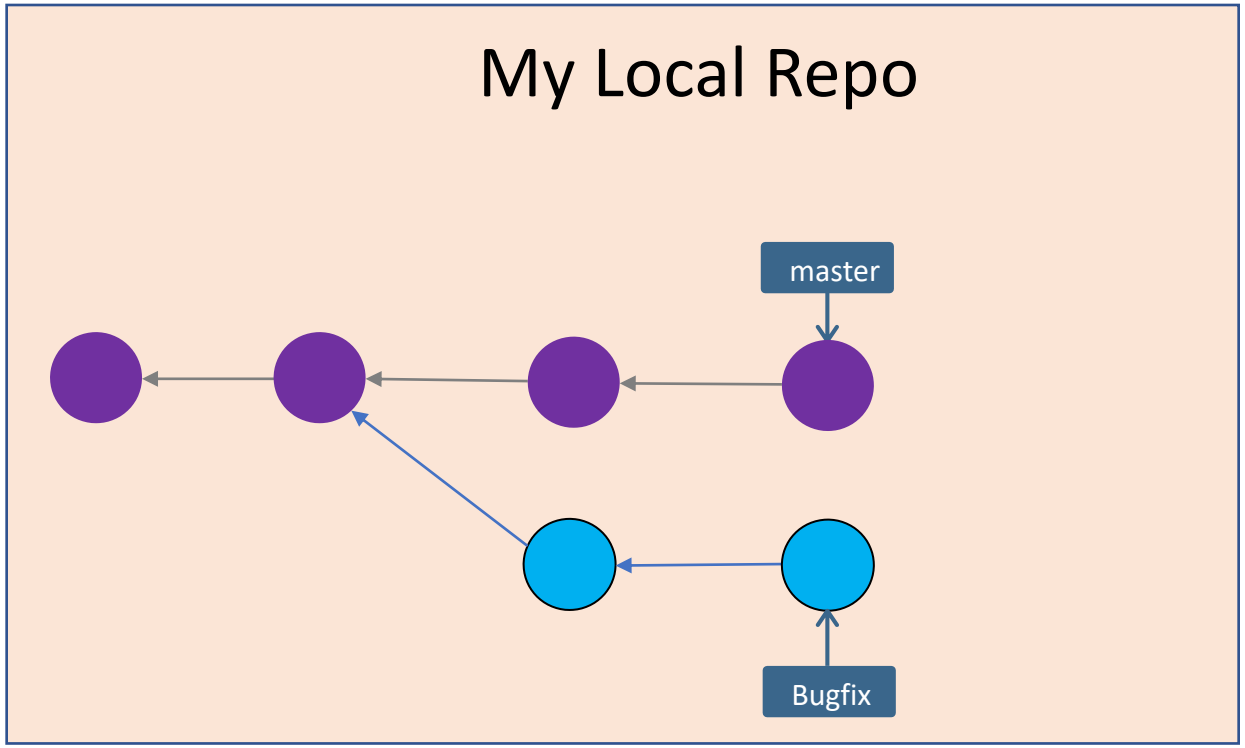


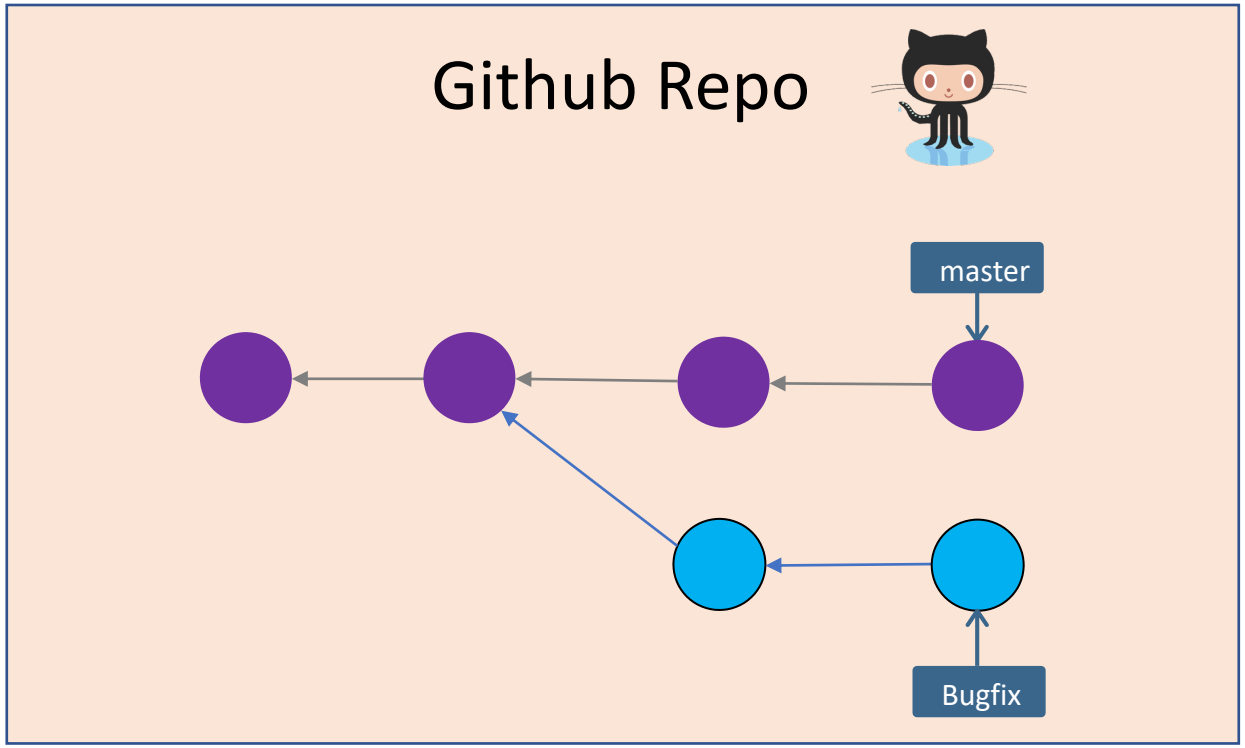

```
$ git push origin master
```

To push up my master branch to my Github repo
(assuming my remote is named origin

TRAINOCATE

Pushing

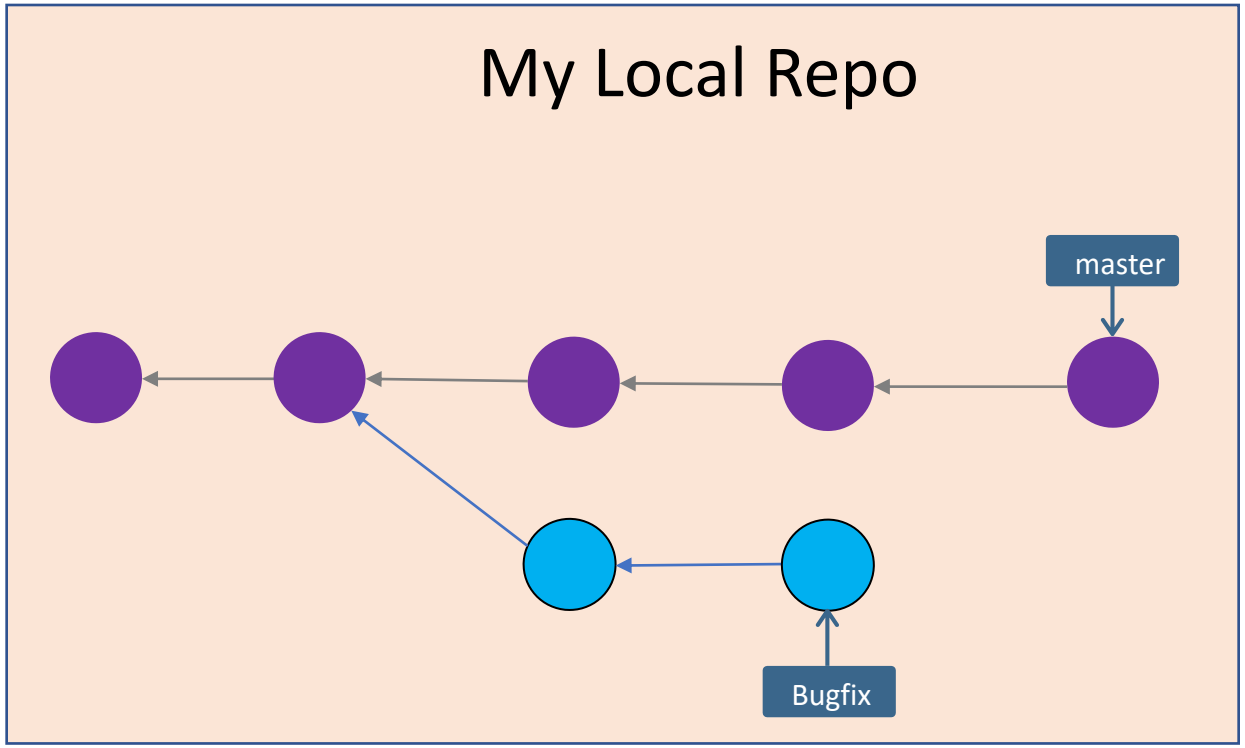


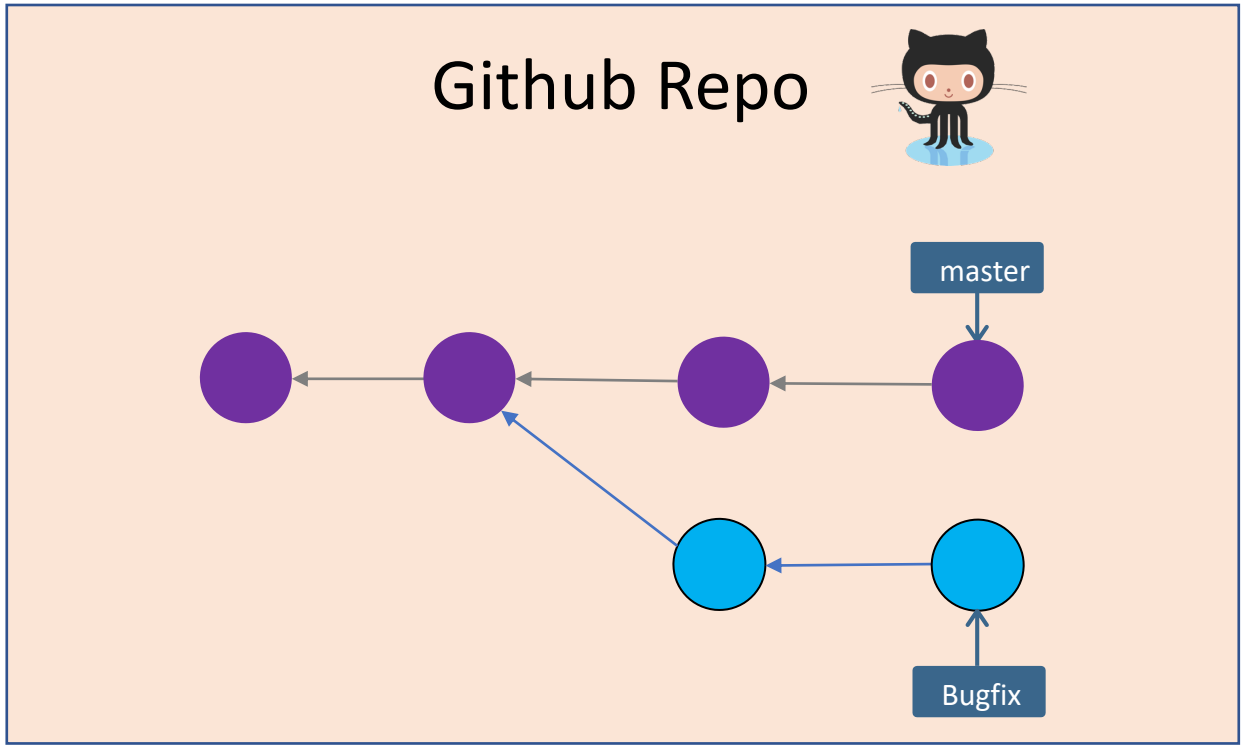



```
$ git push origin Bugfix
```

To push up the Bugfix branch to Github

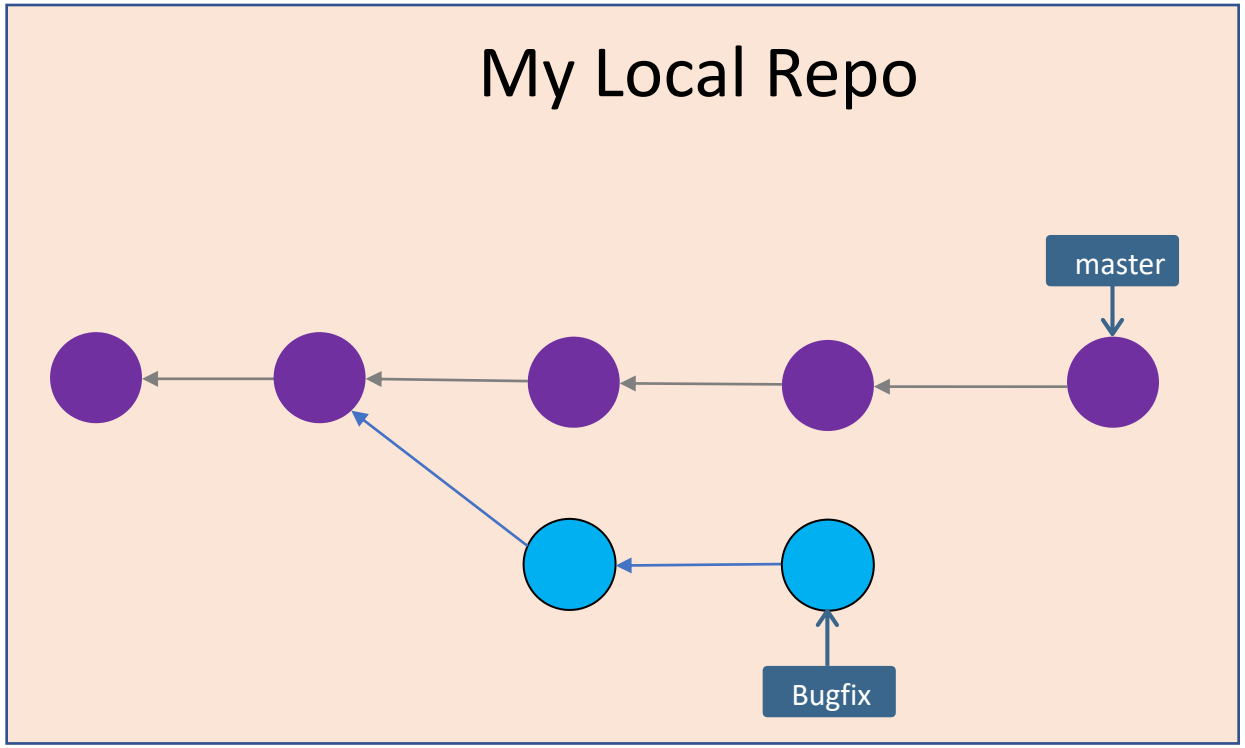
Pushing

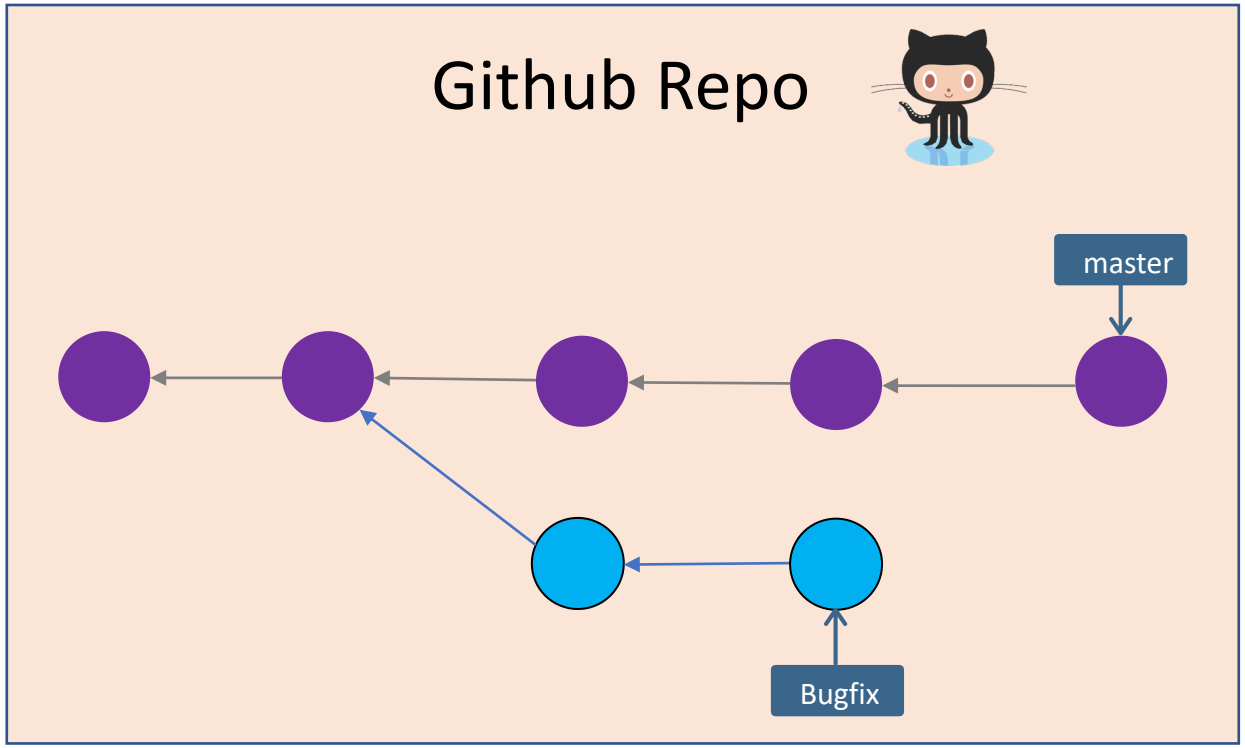




I make some new commits locally.
My Github repo has no idea!

Pushing

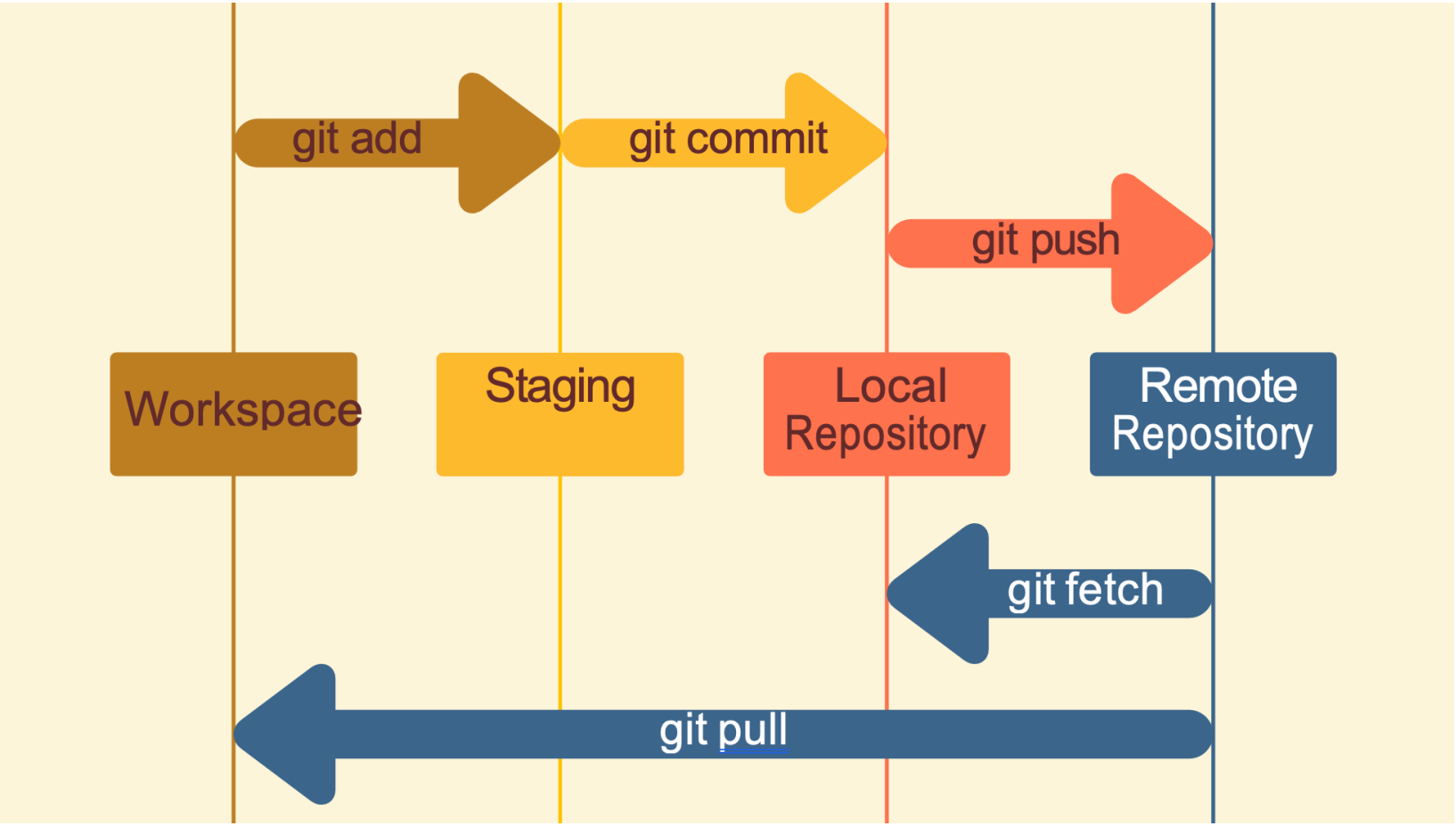





```
$ git push origin master
```

Push up the master branch again, to make sure the Github repo has the new commits

Workflow Recap



Option 2: Start From Scratch

If you haven't begun work on your local repo, you can...

- Create a brand-new repo on Github Clone it down to your machine
- Do some work locally
- Push up your changes to Github

Fetching & Puling

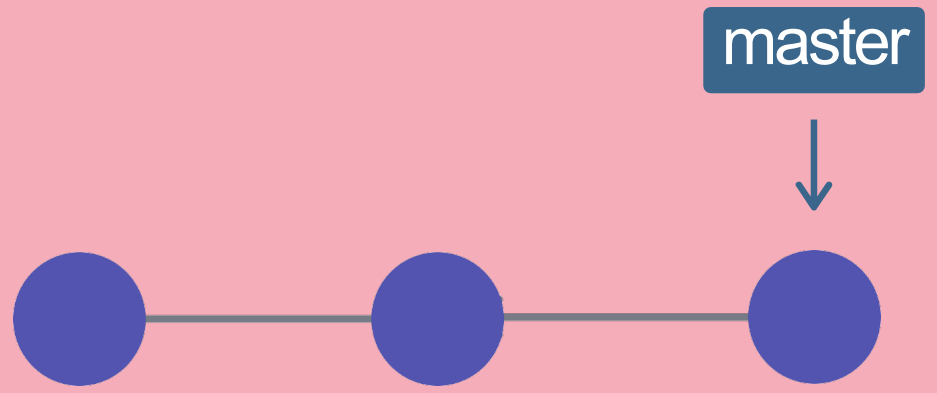
TRAINOCATE

A Closer Look At Cloning

TRAINOCATE

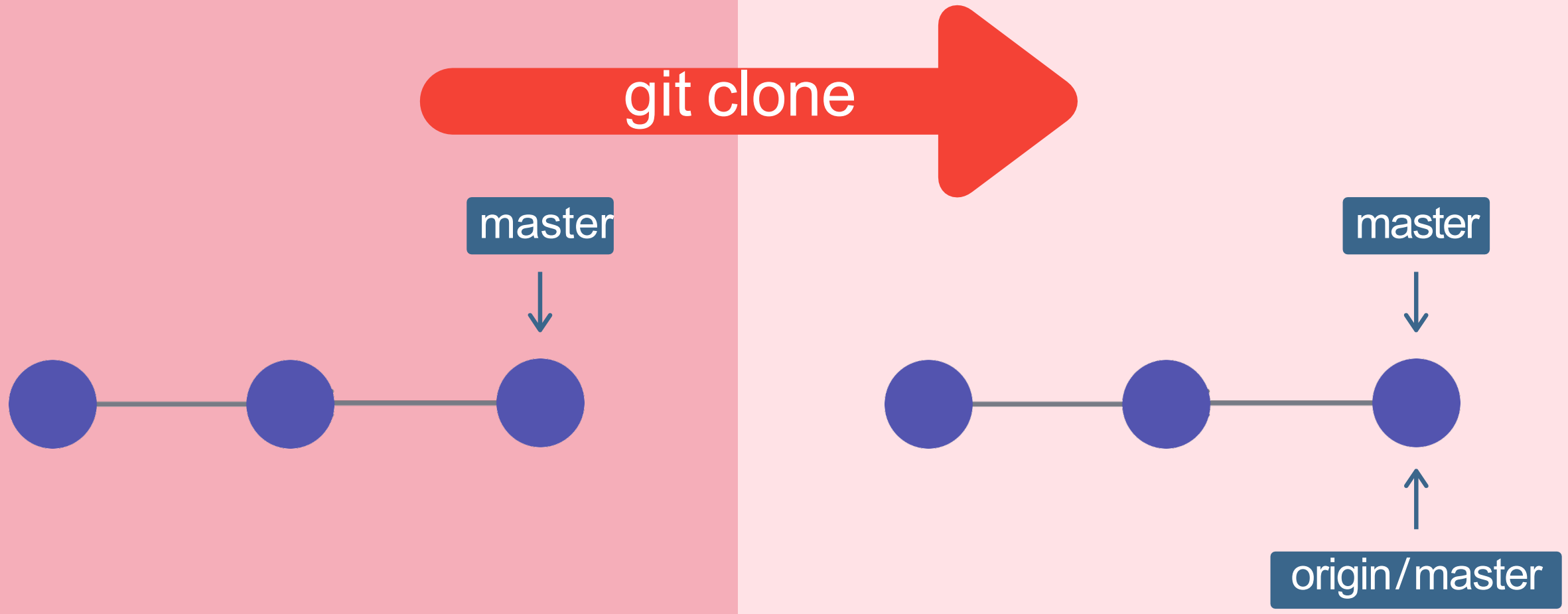
Github Repo

My Computer

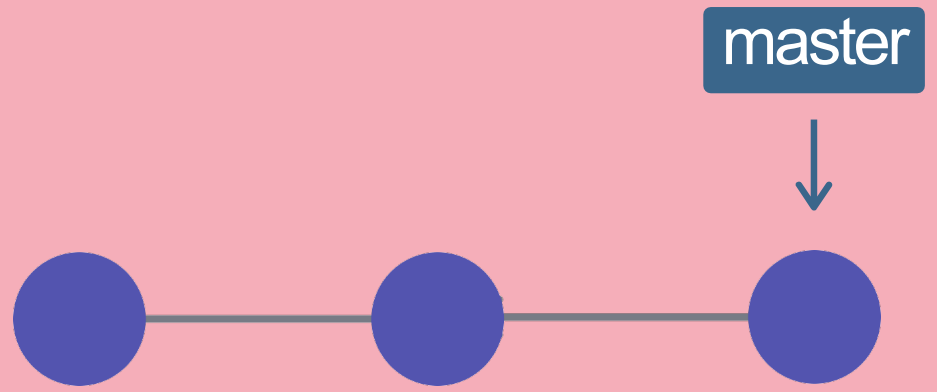


Github Repo

My Computer

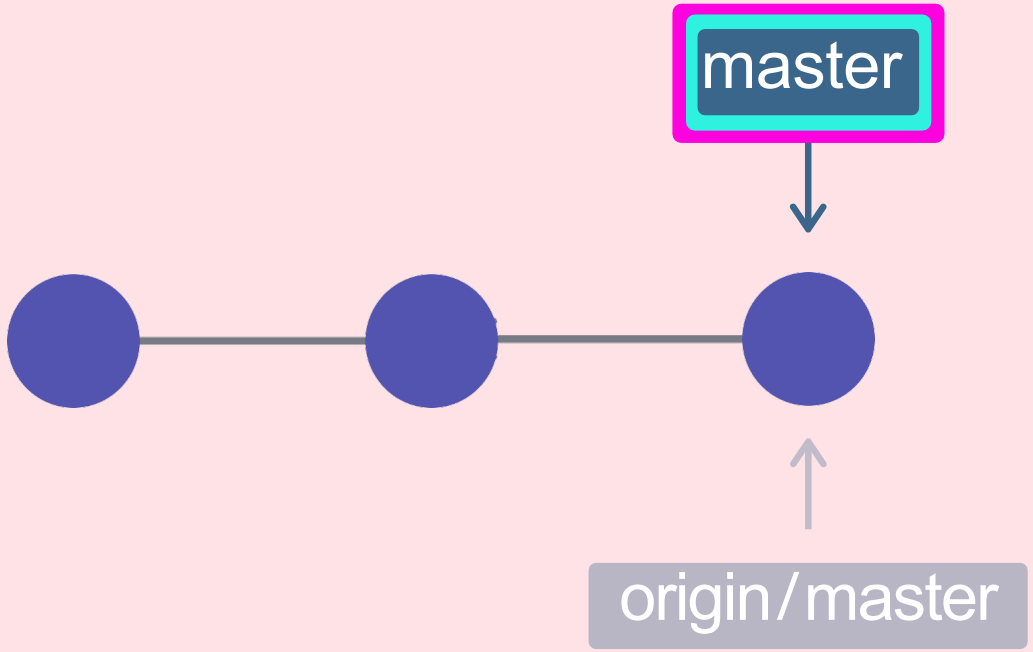


Github Repo



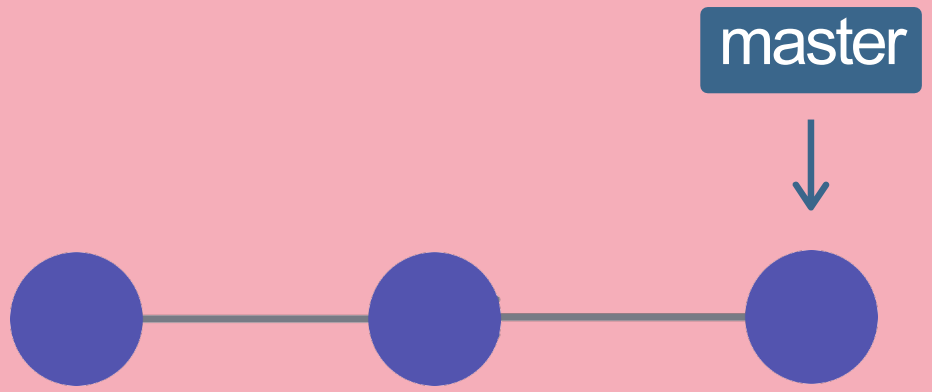
My Computer

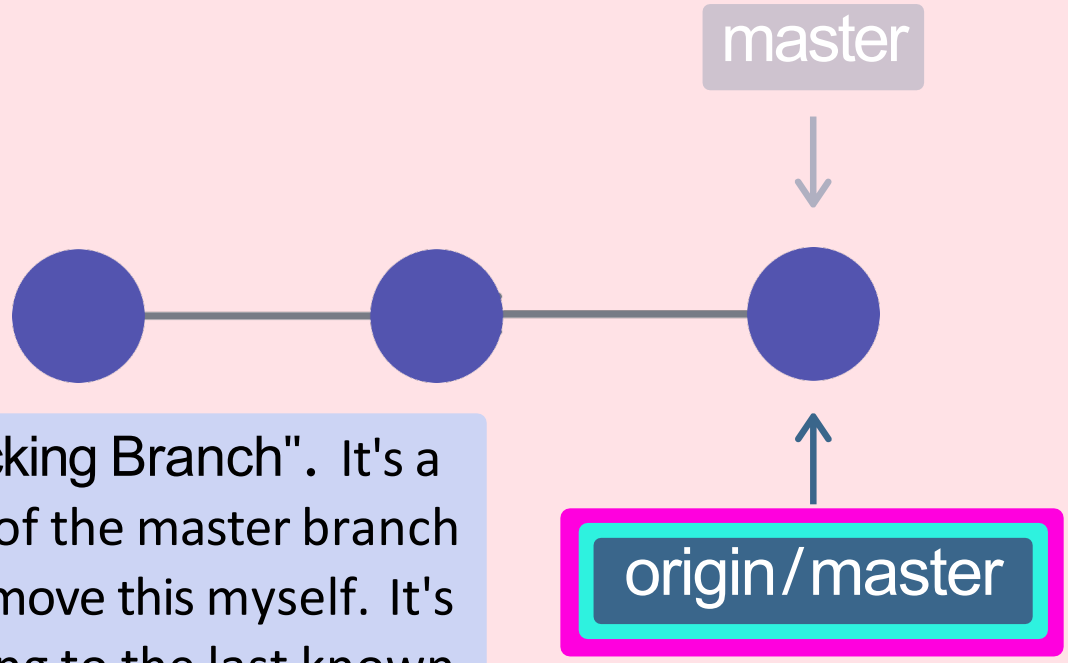
A regular branch reference.
I can move this around myself.



Github Repo

My Computer





This is a "Remote Tracking Branch". It's a reference to the state of the master branch on the remote. I can't move this myself. It's like a bookmark pointing to the last known commit on the master branch on origin

Remote Tracking Branches

At the time you last communicated with this remote repository, here is where x branch was pointing"

- They follow this pattern <remote>/<branch>. origin/master references the state of the master branch on the remote repo named origin.
- upstream/logoRedesign references the state of the logoRedesign branch on the remote named upstream (a common remote name)

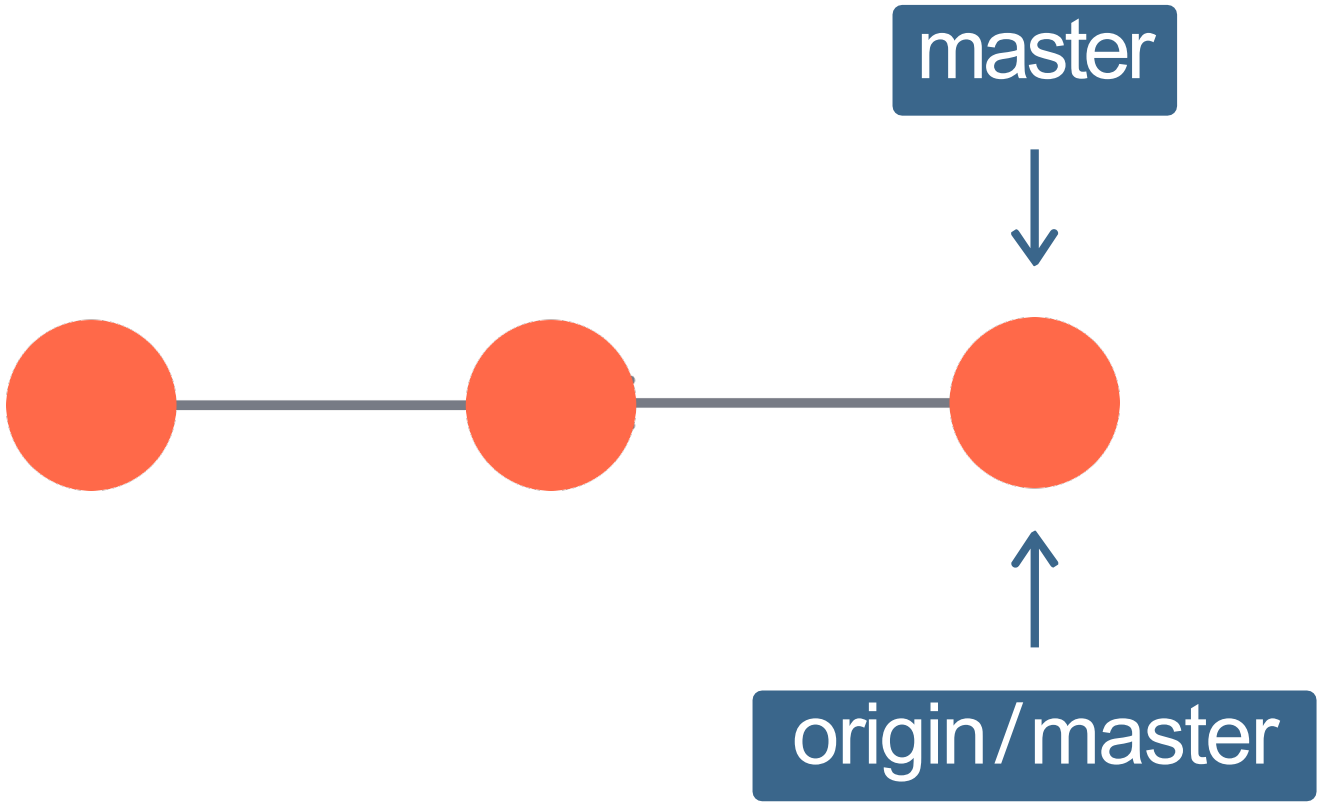
TRAINOCATE

Remote Branches

Run git branch -r to view the remote branches our local repository knows about.

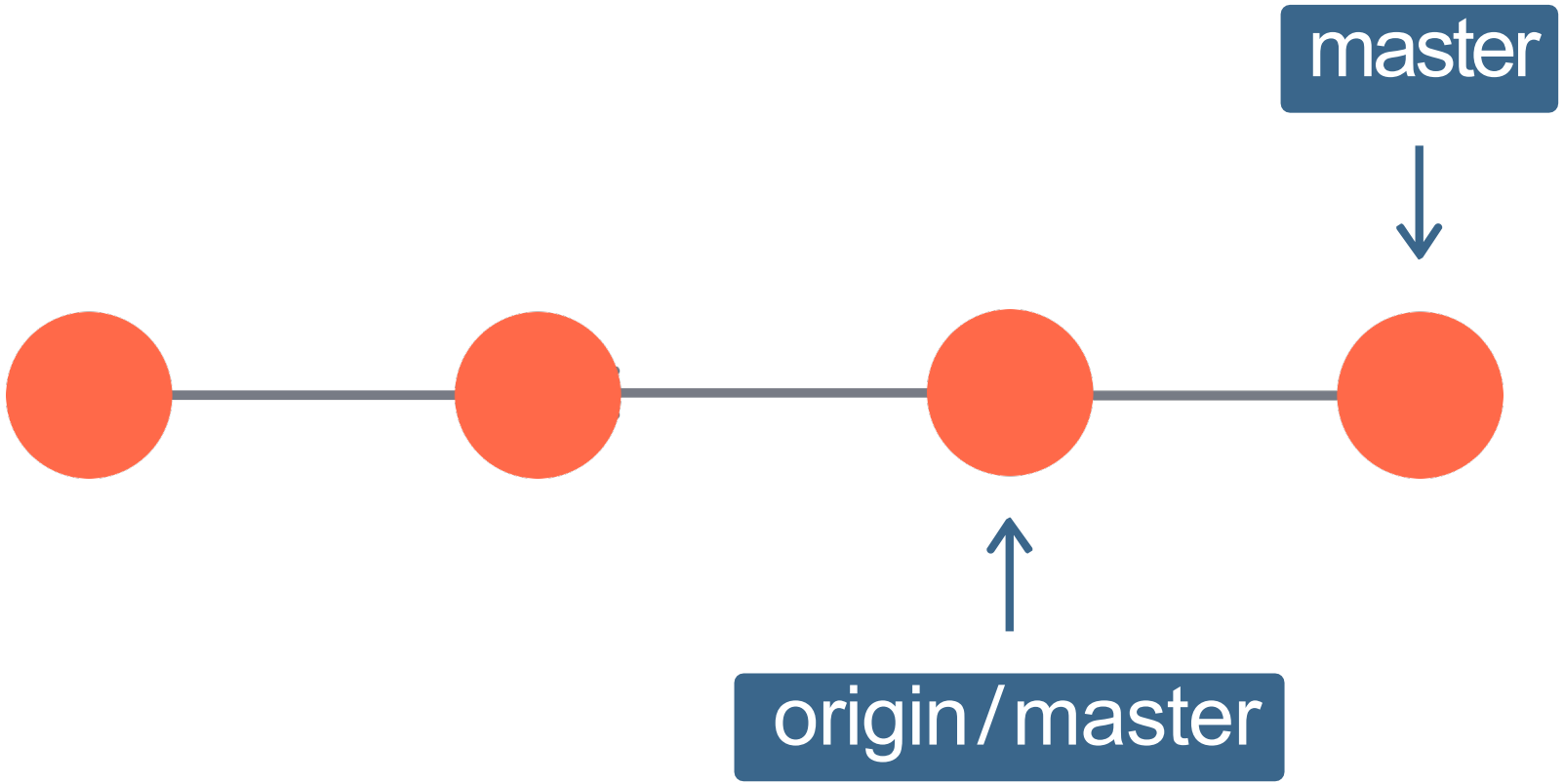
```
$ git branch –r origin/master
```

My Computer



My Computer

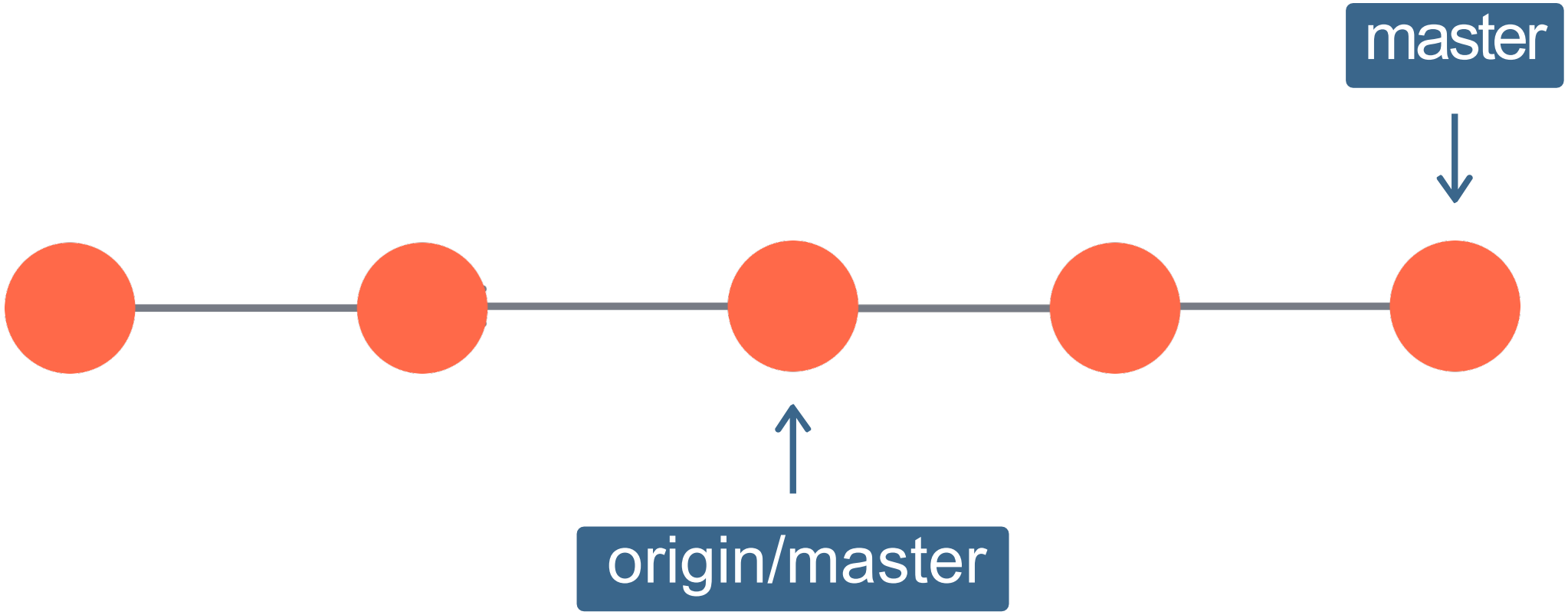
I make a new commit locally. My master branch reference updates, like always.



The remote reference stays the same

My Computer

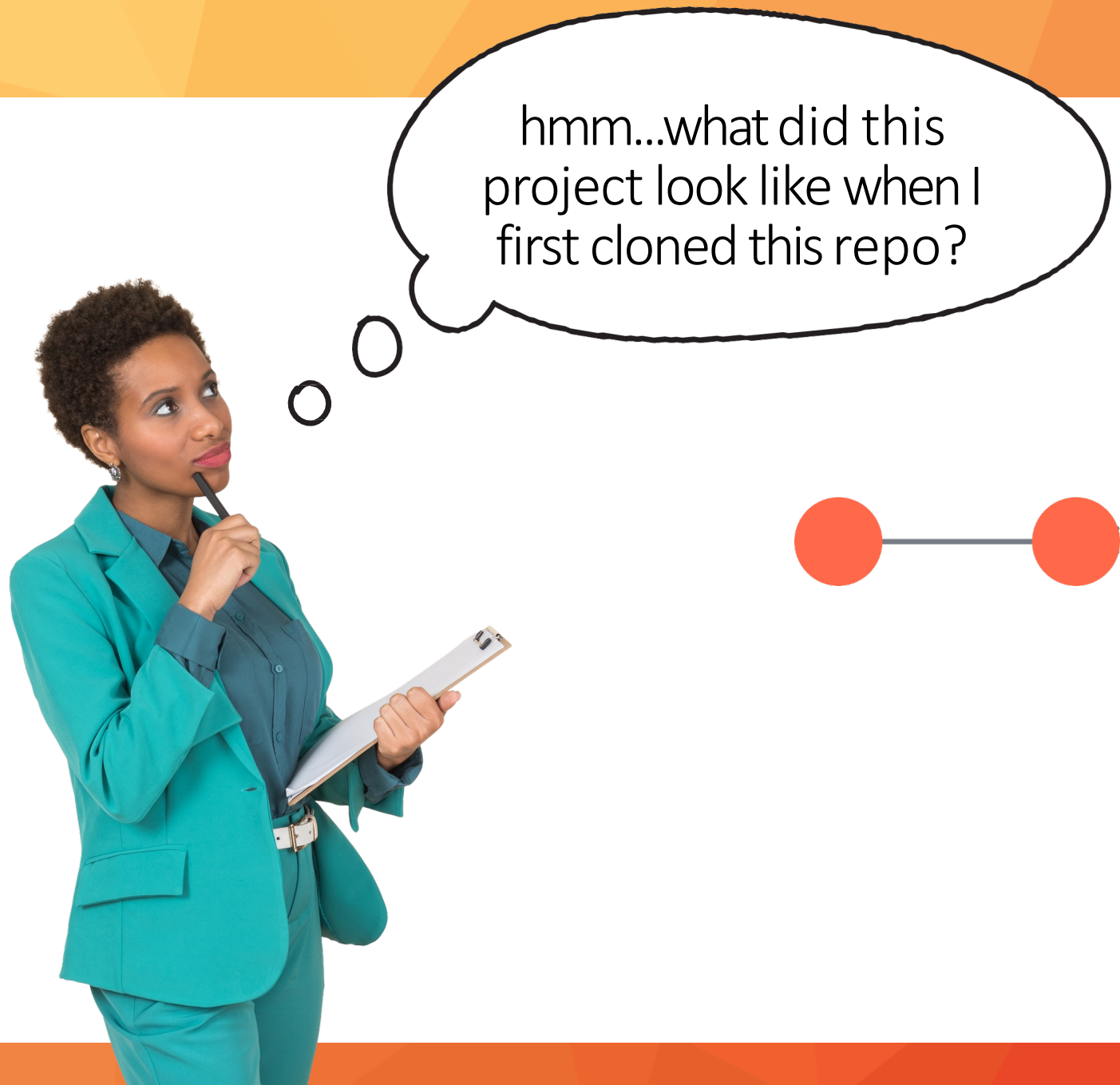
I make another commit, and the local branch reference moves again.

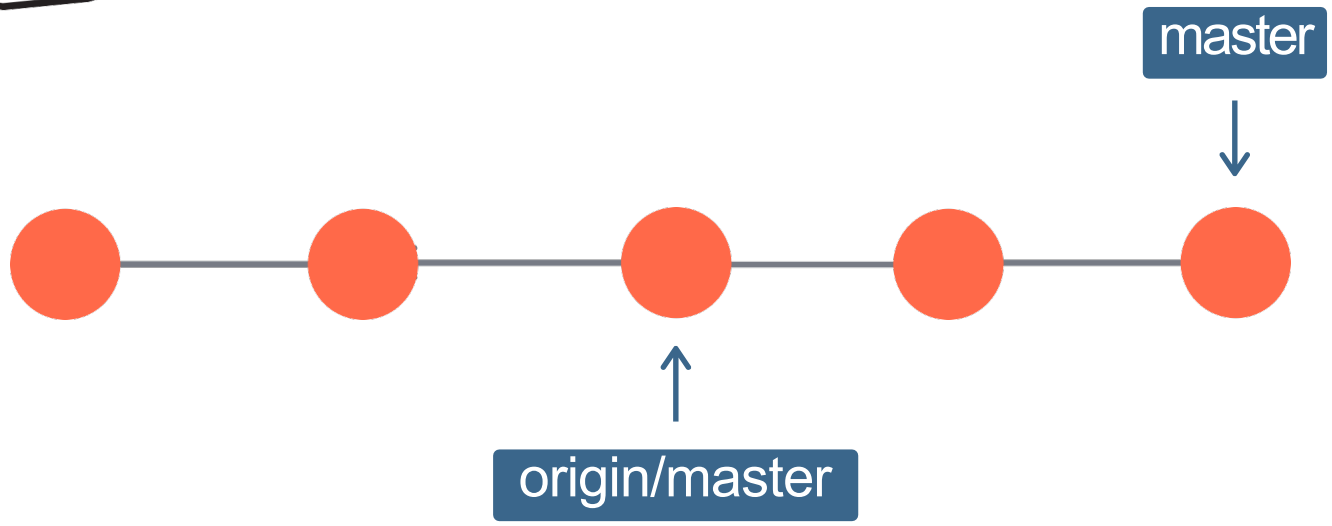


Remote reference doesn't move!

When I run git status

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)
```


hmm...what did this project look like when I first cloned this repo?


master
origin/master

TRAINOCATE

You can checkout these remote branch pointers

```
$ git checkout origin/master
Note: switching to 'origin/master'.
You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this .....
```

Fetching

- Fetching allows us to download changes from a remote repository, BUT those changes will not be automatically integrated into our working files.
- It lets you see what others have been working on, without having to merge those changes into your local repo.
- Think of it as "please go and get the latest information from Github, but don't screw up my working directory."

TRAINOCATE

Git Fetch

- The git fetch <remote> command fetches branches and history from a specific remote repository. It only updates remote tracking branches.

- git fetch origin would fetch all changes from the origin remote repository.

```
$ git fetch <remote>
```

If not specified, <remote> defaults to origin

Git Fetch

- We can also fetch a specific branch from a remote using git fetch <remote> <branch>

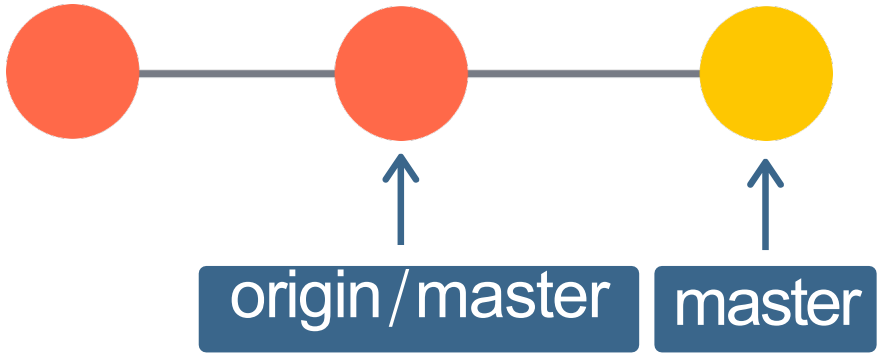
- For example, git fetch origin master would retrieve the latest information from the master branch on the origin remote repository.

```
$ git fetch <remote> <remot—branch>
```

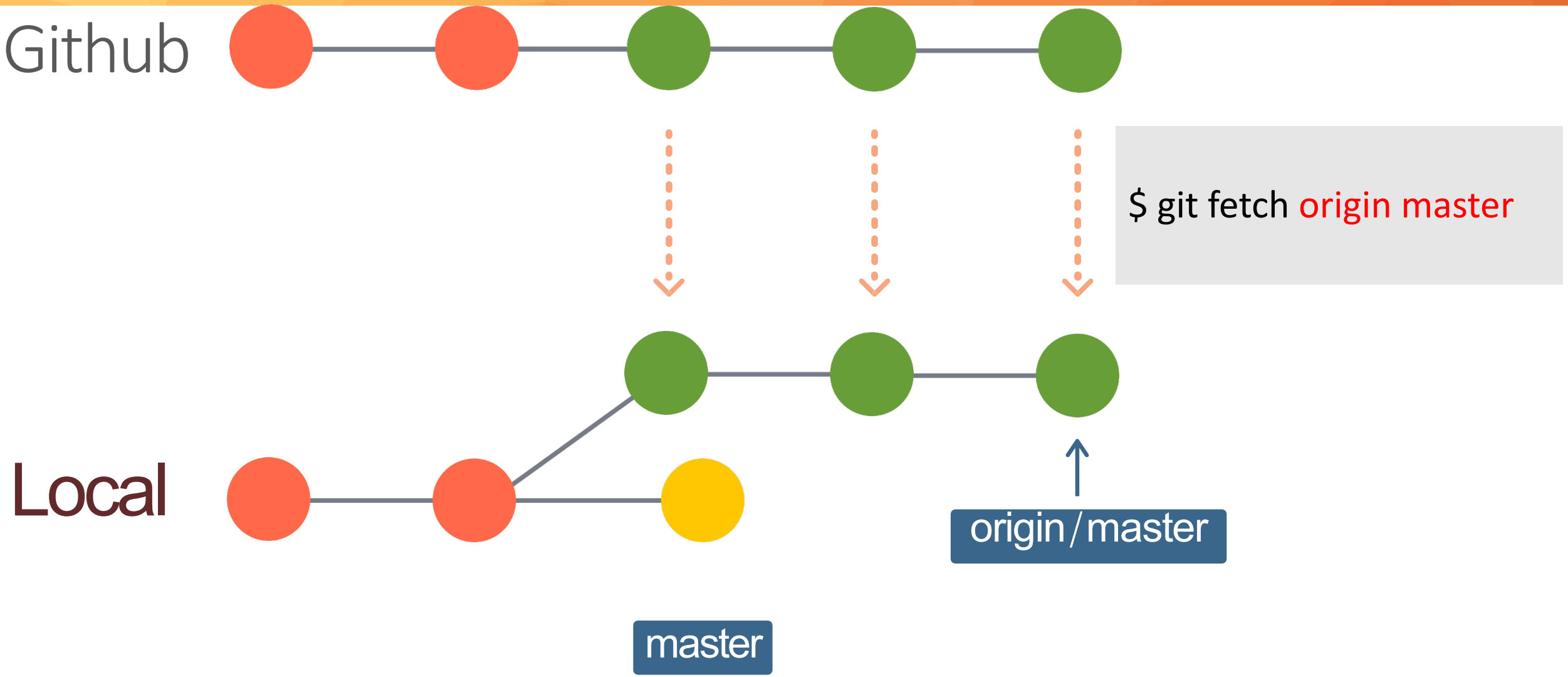
Github

- Uh oh! The remote repo has changed! A teammate has pushed up changes to the master branch, but my local repo doesn't know!

Local



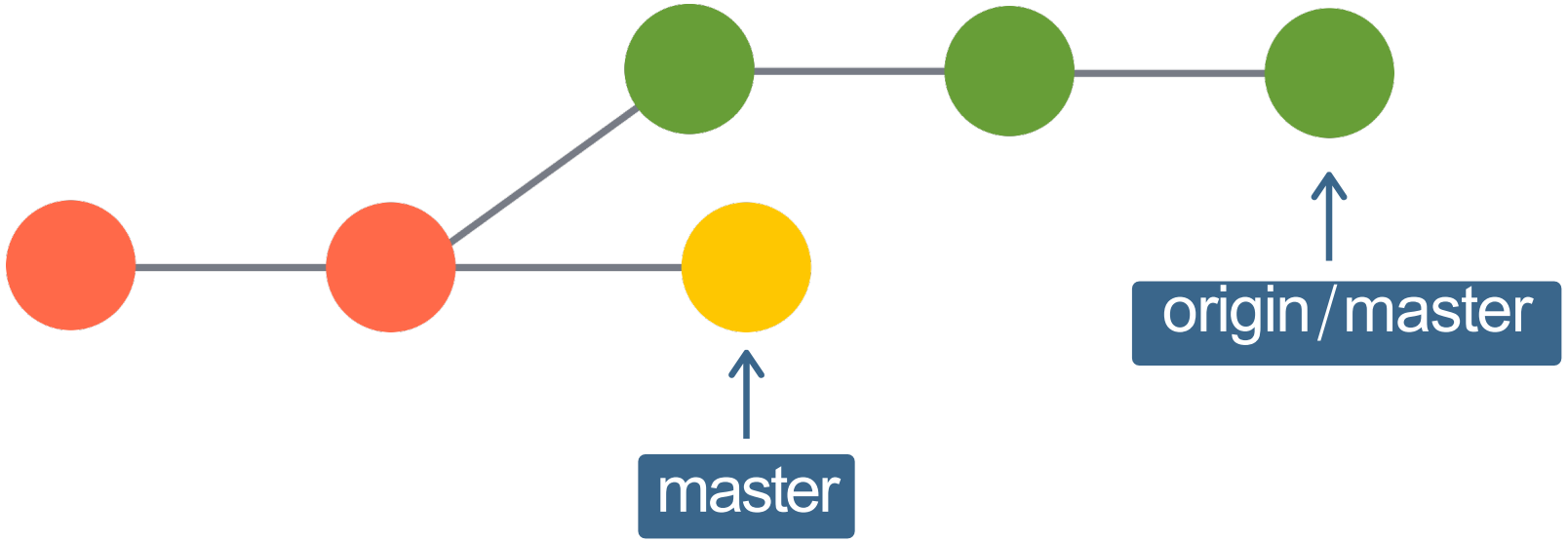
How do I get those changes???

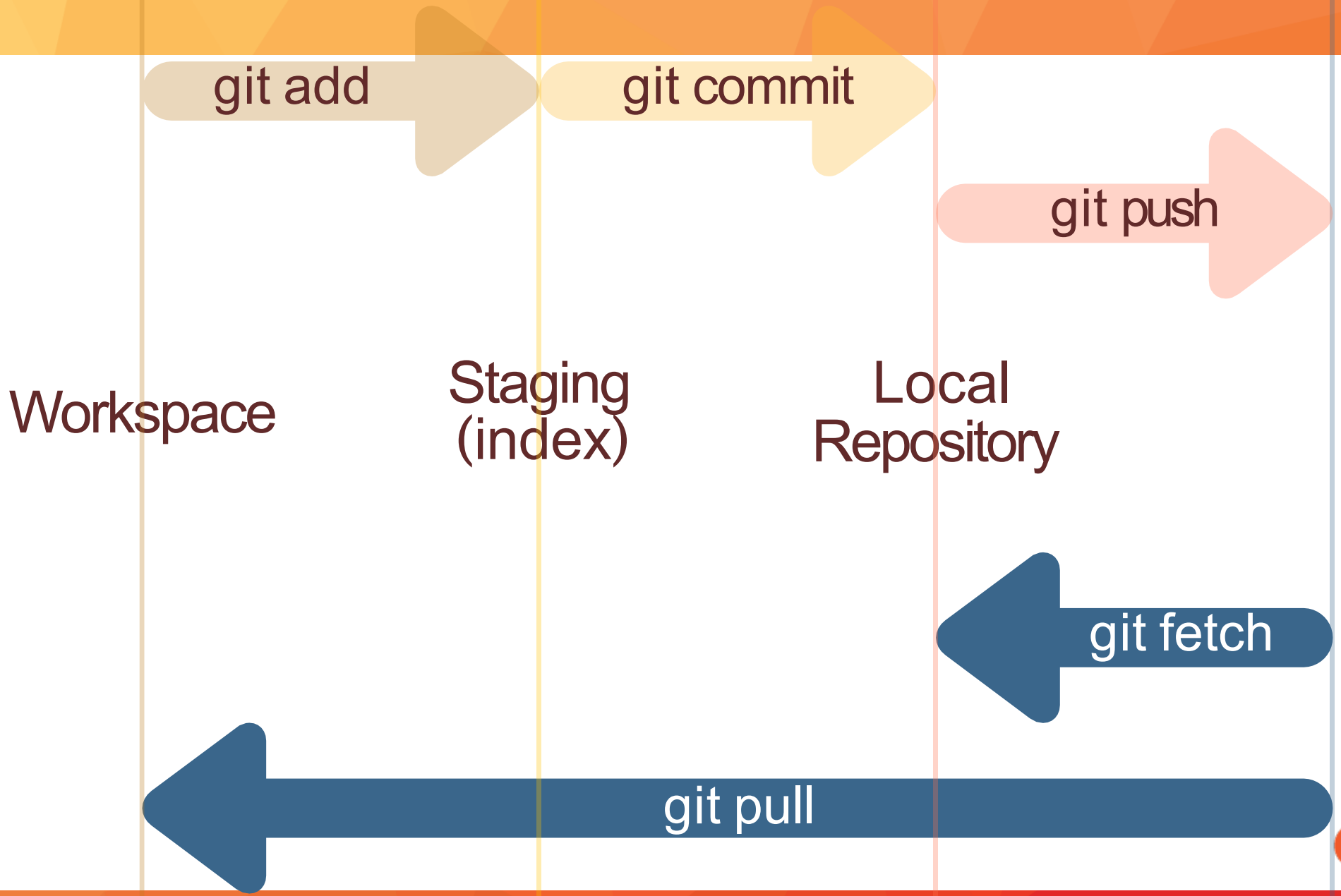


Github

I now have those changes on my machine, but if I want to see them I have to checkout origin/master. My master branch is untouched!

Local




git add
git commit
git push
Workspace
Staging
(index)
Local
Repository
git fetch
git pull

Puling

- git pull is another command we can use to retrieve changes from a remote repository. Unlike fetch, pull actually updates our HEAD branch with whatever changes are retrieved from the remote.

- "go and download data from Github AND immediately update my local repo with those changes"

TRAINOCATE

git pull = git fetch + git merge

update the remote tracking branch with the latest changes from the remote repository

update my current branch with whatever changes are on the remote tracking branch

git pull

- To pull, we specify the particular remote and branch we want to pull using git pull <remote> <branch>.
  Just like

- with git merge, it matters WHERE we run this command from. Whatever branch we run it from is where the changes will be merged into.

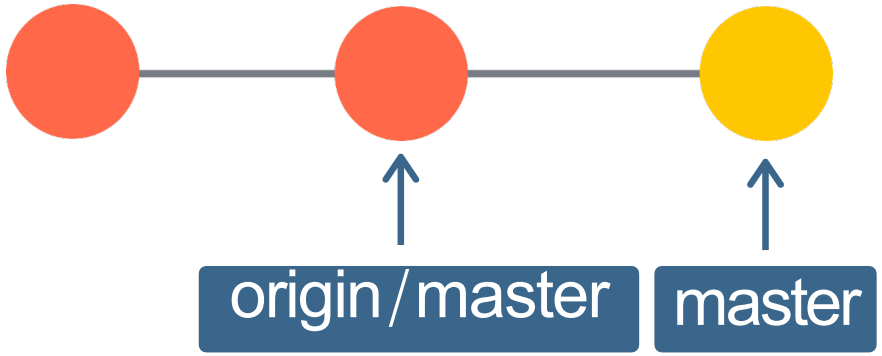
- git pull origin master would fetch the latest information from the origin's master branch and merge those changes into our current branch.

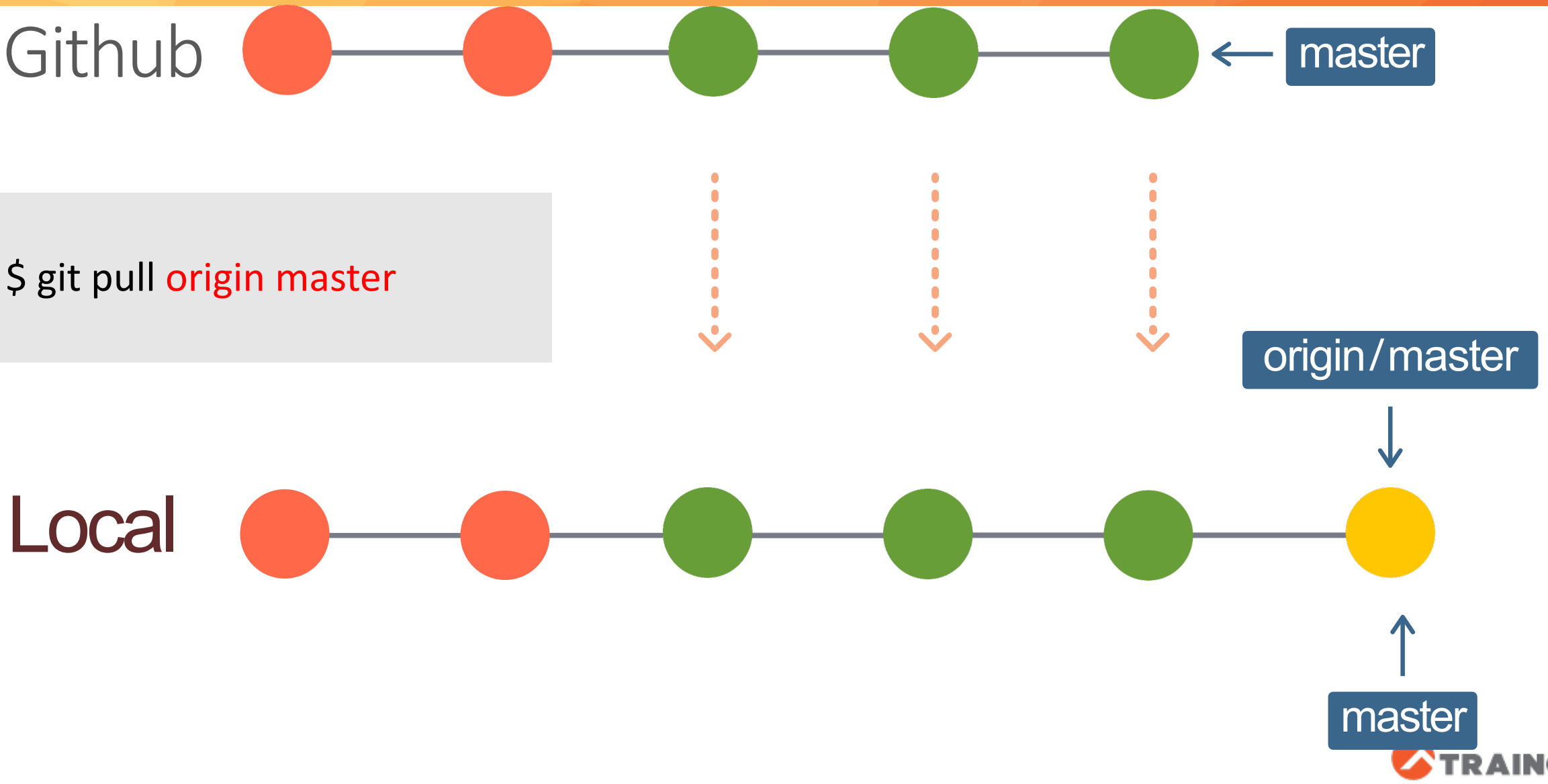
```
$ git pull <remote> <remot—branch>
```

pulls can result in merge conflicts !

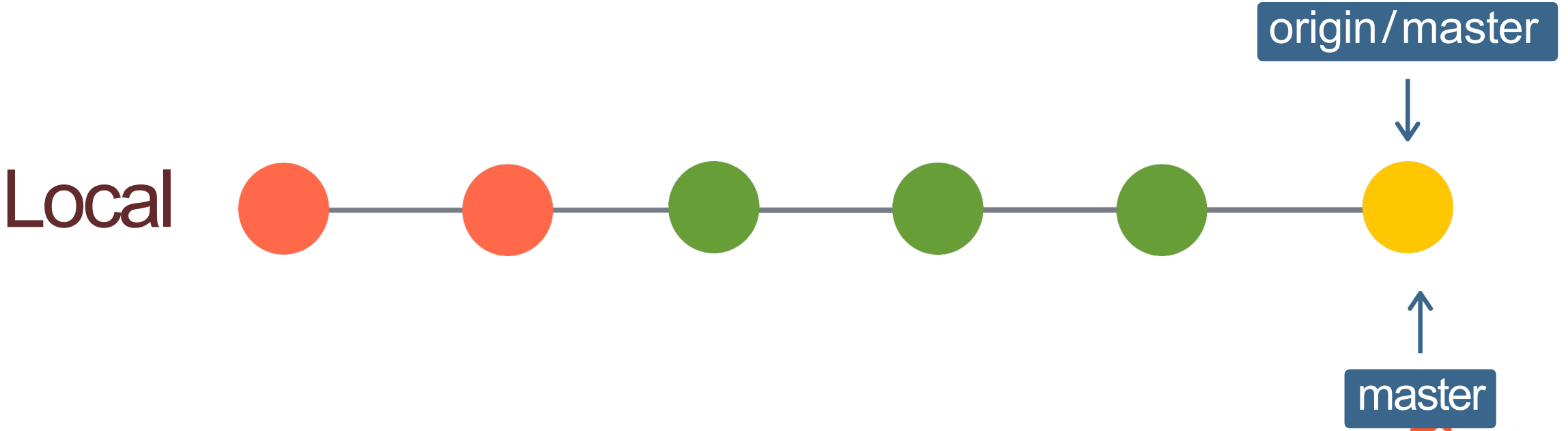
Github

Local




Github
master
$ git pull origin master
origin/master
Local
master

I now have the latest commits from origin/master on my local master branch (assuming I pulled while on my master branch)



Github

Local

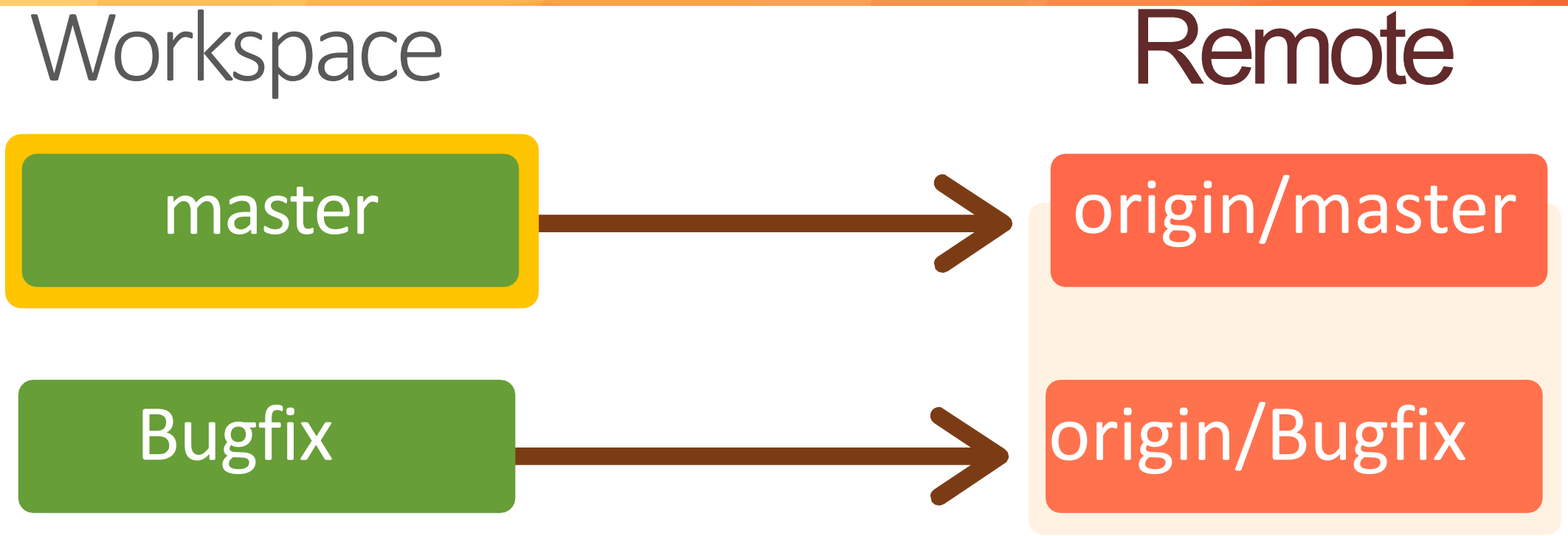
I have a commit locally that is not on Github.
When I pulled, it was merged with the new commits.
As with any other merge, this can result in merge conflicts.

An even easier syntax!

If we run git pull without specifying a particular remote or branch to pull from, git assumes the following:

- remote will default to origin
- branch will default to whatever tracking connection is configured for your current branch.

```
$ git pull
```



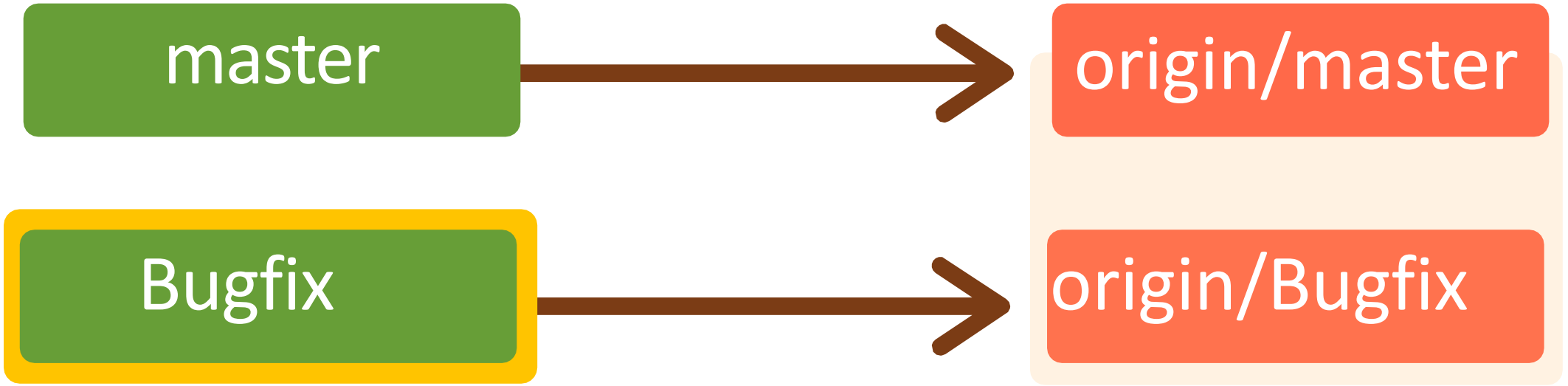
When I'm on my local master branch...

```
$ git pull
```

pulls from origin/master automatically

Workspace

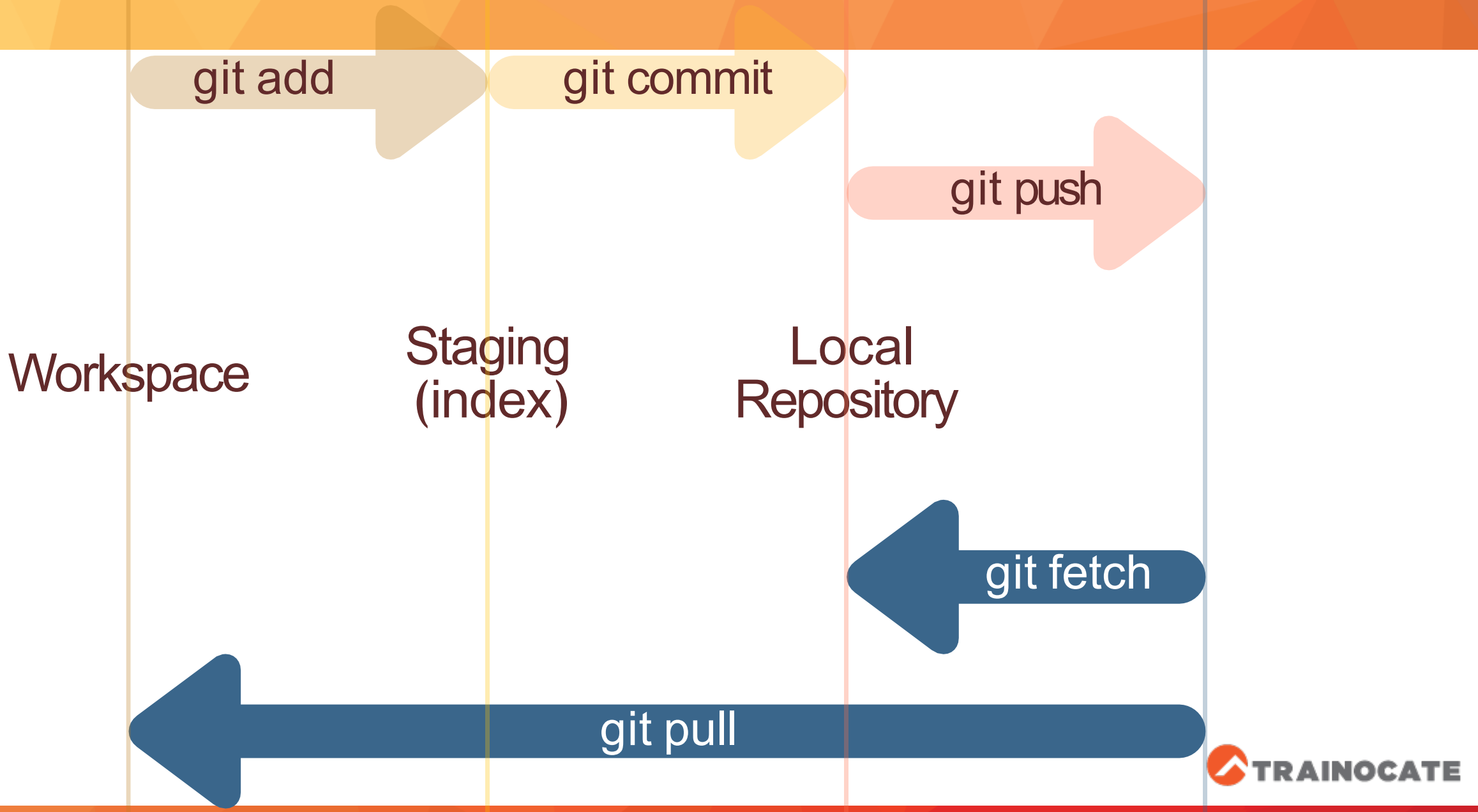
Remote



When I'm on my local Bugfix branch...

```
$ git pull
```

pulls from origin/Bugfix automatically


git add
git commit
git push
Workspace
Staging
(index)
Local
Repository
git fetch
git pull

git fetch

- Gets changes from remote branch(es) Updates the remote-tracking branches with the new changes
- Does not merge changes onto your current HEAD branch
- Safe to do at anytime

git pull

- Gets changes from remote branch(es) Updates the current branch with the new changes, merging them in
- Can result in merge conflicts Not recommended if you have uncommitted changes!


TRAINOCATE