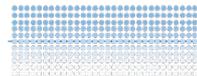
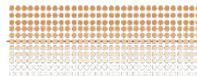
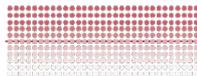
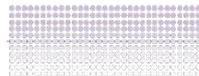


Git/GitHub Basics



New Zealand eScience Infrastructure

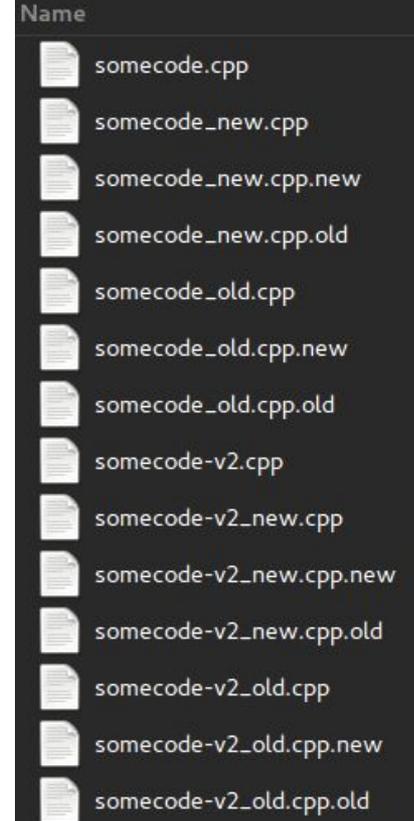


Why should researchers care about Git/Github?

Have you ever..

- Deleted an important file?
- Needed to manage multiple versions of something?
- Needed to host your work somewhere easily accessible?
- Needed to collaborate with anyone on anything ever?

Bad version control



Focus will be on the *minimum knowledge* necessary to use Git/GitHub effectively.

We will cover...

Basics of Git version control

- Making a Git repo.
- Staging files.
- Committing files.

Simple Git branching

- Making branches
- Switching branches

Use of GitHub

- Making a GitHub repo.
- Adding a git remote.
- Pushing, Pulling.

Collaboration on GitHub

- Pull requests.
- Forking

Focus will be on the minimum knowledge necessary to use Git/GitHub effectively.

We **will not** be covering...

Merge Strategies.

Cherry picking.

rebase, reset.

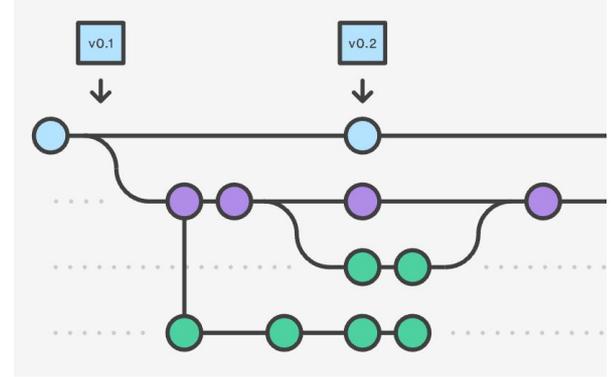
GUIs, tree visualisation.

Tagging.

Detached head.

Version Control (Source Control)

A system for recording and managing changes to a file or set of files.



Git is a tool, *GitHub* is a service.

Git

- A version control system.
- Free, OpenSource.
- Distributed (as opposed to centralised).
- Originally created for the development of the Linux kernel.



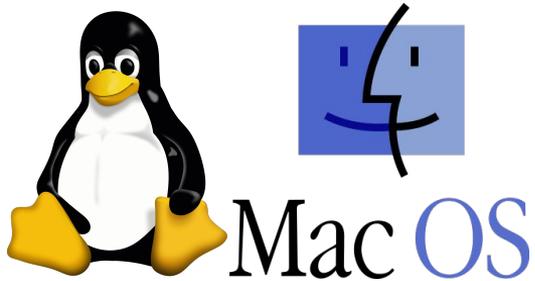
GitHub

- A website that hosts repositories.
- Free.
- Largest host of source code in the world.
- Provides many additional tools, Web hosting, access control, documentation, task management, etc.



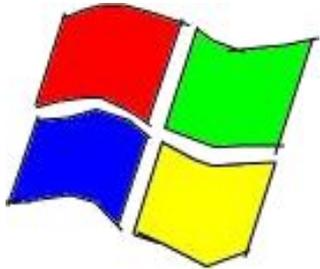
You can use Git without GitHub, and vice versa. But it is most useful when used together.

Where do I get Git?



Linux or MacOS

Should already be installed, check using `git --version`, otherwise install with whatever package manager you use.



Windows

Google 'download Git'.

Make sure you install Git Bash, as this allows you to use Git from command line.

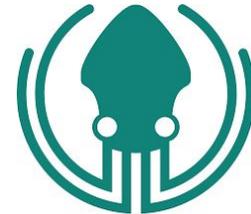


Git Bash

Git is *primarily a command line tool*, and this is all we will cover here.

However there are many different GUIs and programs with git integration.

It is worth checking if you are already using software with these features.



You may find it most useful to use a mix of command line, GUIs and integration.



git Basics

Terminology

- Repo

What is a Repository? (Repo)

A history of file changes. (Lab Notebook).

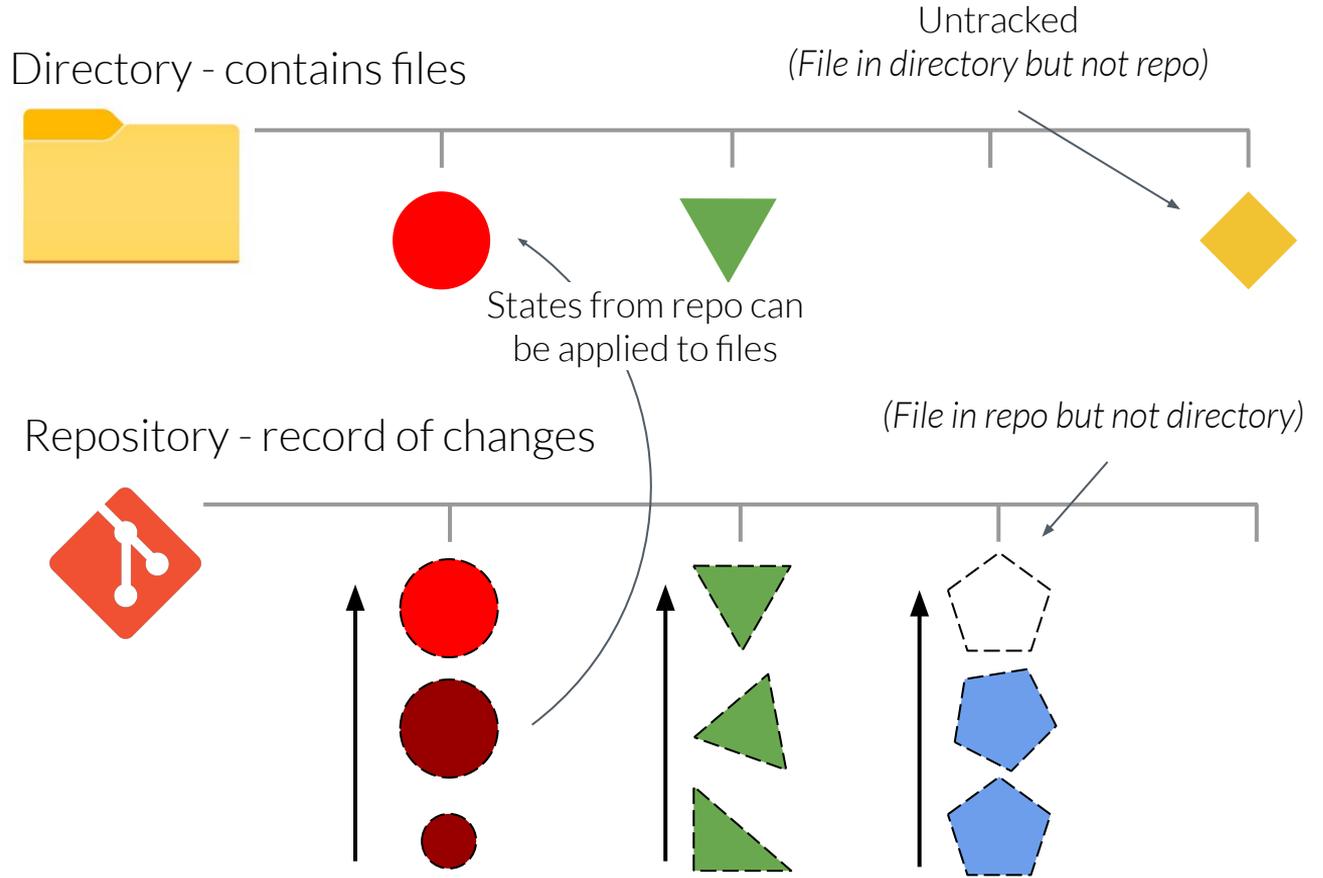
In a directory.

Files in that directory **are not** automatically included in the repo. (Not like gdrive / onedrive).

Name	Date modified	Type	Size
.git	15/05/2020 5:11 PM	File folder	
Data_4s	30/08/2019 12:05 ...	File folder	
01library.tcl	4/06/2019 3:17 PM	TCL File	9 KB
01LogFile.txt	30/08/2019 12:19 ...	TXT File	578 KB
02nodes.txt	30/08/2019 12:05 ...	TXT File	7 KB
04model.txt	30/08/2019 12:05 ...	TXT File	68 KB
10_storey_DBE_4s.tcl	16/08/2019 11:08 ...	TCL File	47 KB

Stored in a hidden directory (.git) - Don't touch!

Terminology



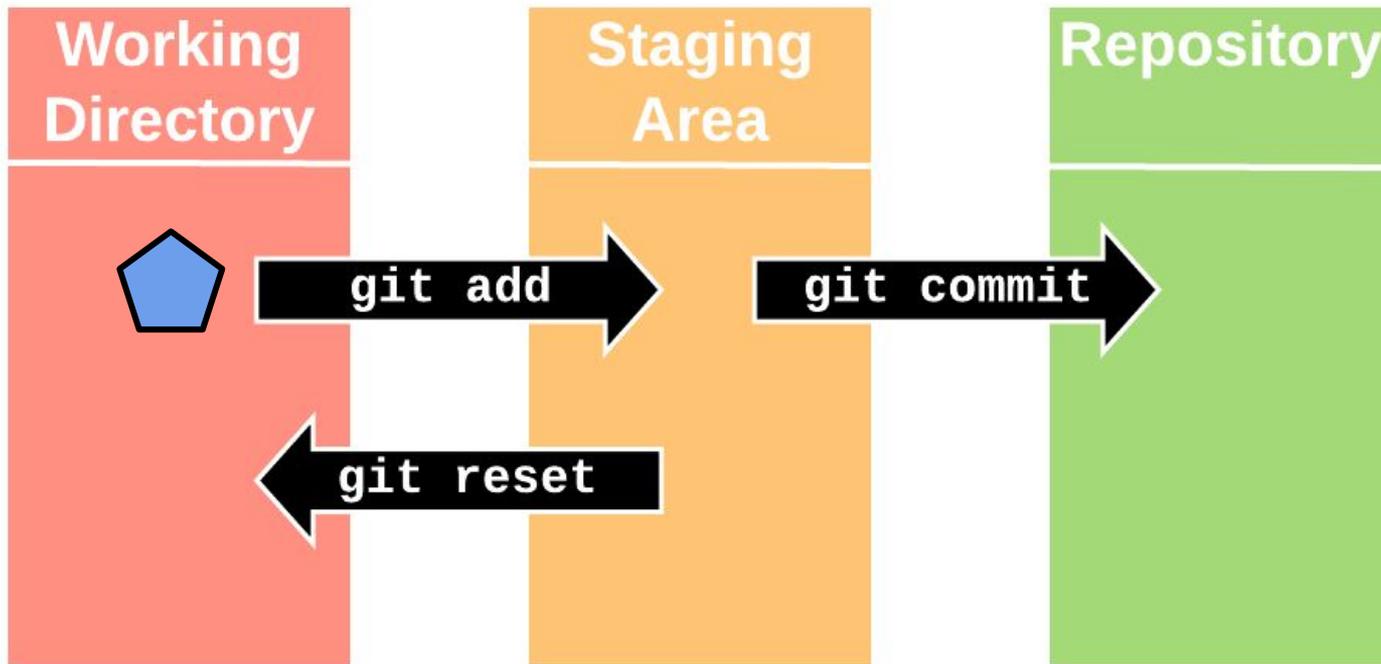
Managing

- init

```
git init
```

Will create an empty git repo in current directory.

Directory can already have files in it (They won't be in the *repo* until added).



Managing

- add

```
git add <file>
```

Will add file <file> to your **staging area**. <file> isn't in repo yet.

The staging area is meant for working changes.

```
git status
```

Will show you...

- Branch name.
 - Staged changes.
 - Changes that haven't been staged. (e.g. files you have edited but not *added*)
 - Some other things.
-

Managing

- mv

```
git rm <path>
```

Will `rm <path>` *and* stage `<path>` for removal.

Else `<path>` will return when you next pull.

```
git mv <path1> <path2>
```

Will `mv <path1> <path2>` *and* record change in repo.

Else `<path2>` will be treated as a new file, and `<path1>` will return when you pull.

Managing

- gitignore

A '.gitignore' file can be used to prevent files from being added

A text file named '.gitignore' (it starts with a dot, so is hidden).

Applies to directory and all child directories.

Specifies (glob) patterns to be excluded from any future `add` commands.

Useful for ...

- Protecting incriminating information.

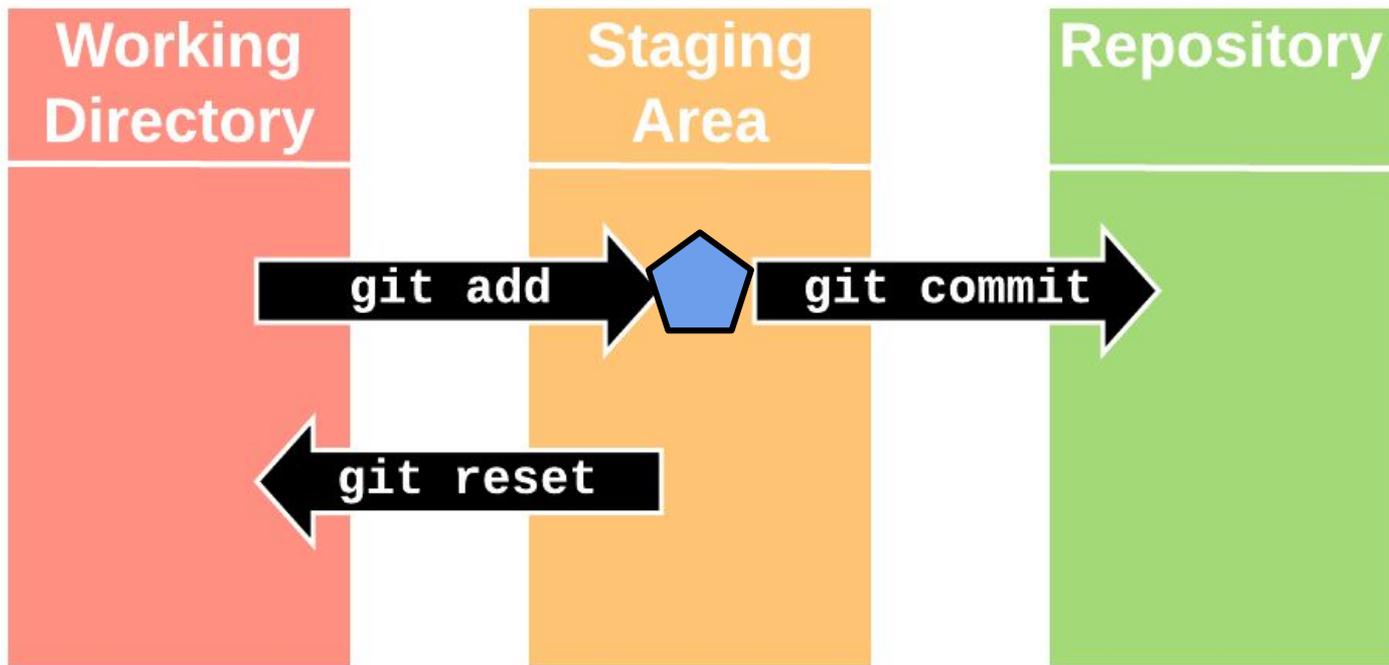
- Avoid filling your repo with trash (temp files etc).

- Exclude datasets, outputs or other non relevant files .

.gitignore

```
_pycache_/*      # Everything in folder
*.swp            # Any file ending in .swp
*keys*          # Any file with the word 'keys' in name.
#*.pyc          # This bit is commented out.
```

You probably do want to add your .gitignore to the repo.



Managing

- commit

```
git commit
```

A commit takes all staged changes and records them in the repo.

Each commit should have a single purpose.

Will open your default text editor for a 'commit message'.

Commit message should be a short summary of your changes.

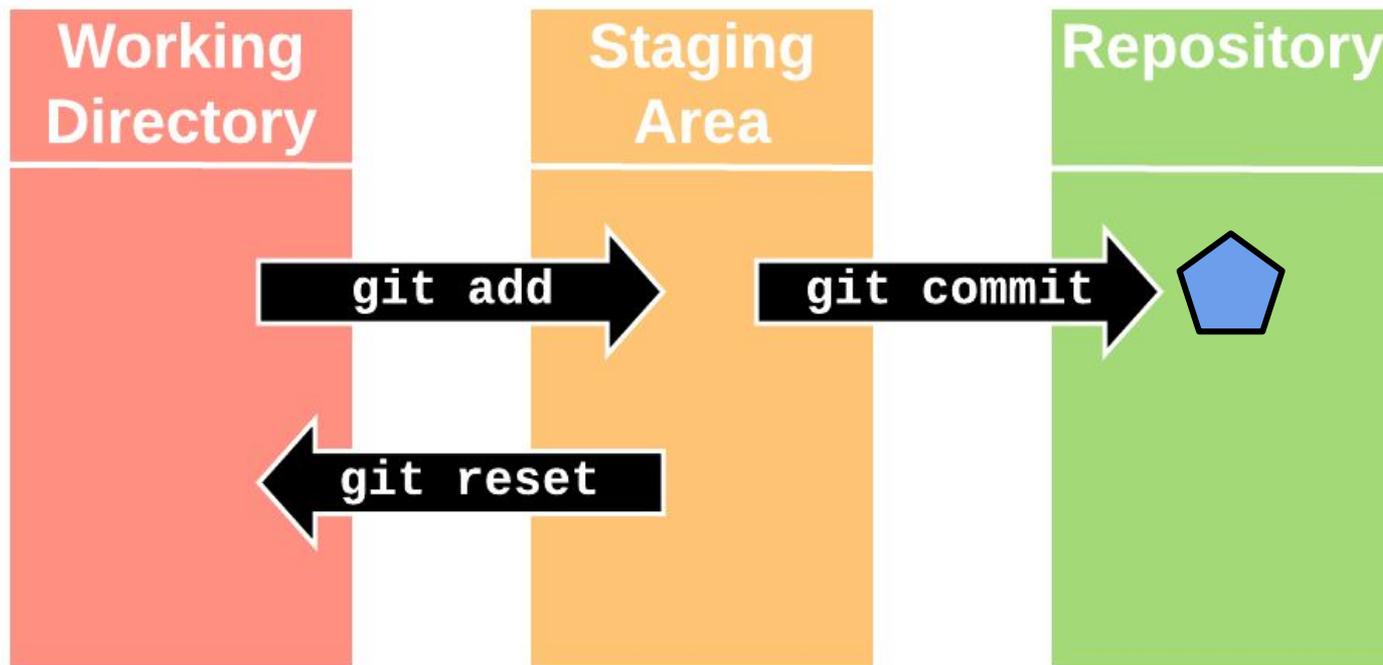
```
git commit -m <message>
```

For when you can't spare the extra seconds to open a text editor.

```
git commit --amend
```

Sneakily add your changes to your last commit.

Good for fixing mistakes and keeping your commit history uncluttered.





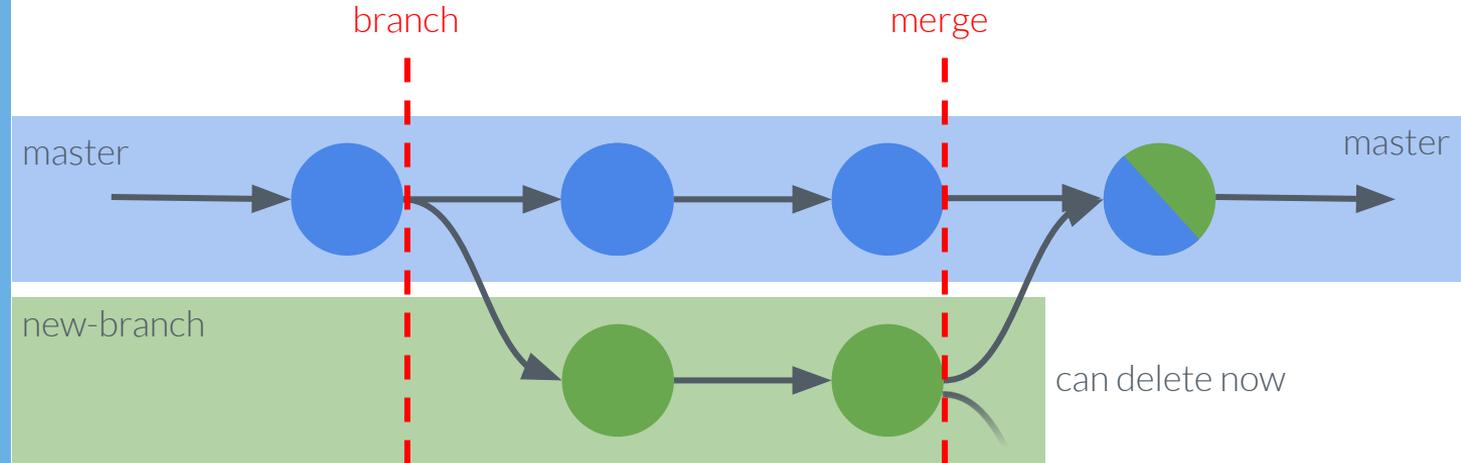
Branches

Branches

- Terminology

To **branch** is to create a separately tracked copy of your files (*within the same repo*).

By convention the main/default branch is called **master**



To **merge** is to incorporate *changes* from one branch into another.

It's good practice to make potentially dangerous changes in a new branch.

Branches

- branching

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

Creates a new branch called `<branch-name>` (try pick a descriptive name).

You are still working on the original until you `git checkout <branch-name>`

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

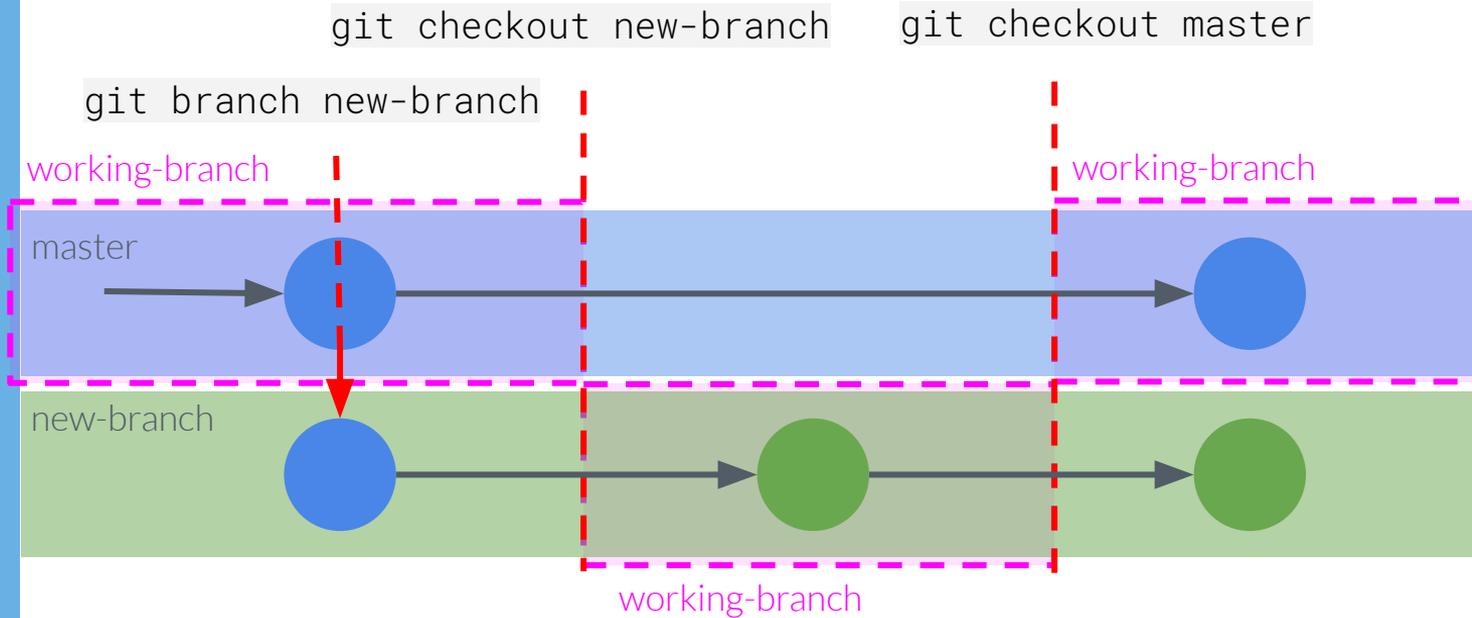
Changes files in your working directory to match `<branch-name>`

Makes `<branch-name>` the **current branch** future changes will be recorded here.

Branches

- checkout

When you **checkout** a branch, your working directory will change to match it.
Any changes committed will be *to that branch*.



You *cannot* checkout while there are unrecorded changes.

First you must `git commit` or `git stash` changes.



GitHub

GitHub

- Remote

Local Repo...

Is the repository you are working on right now.
Where you are changing stuff.

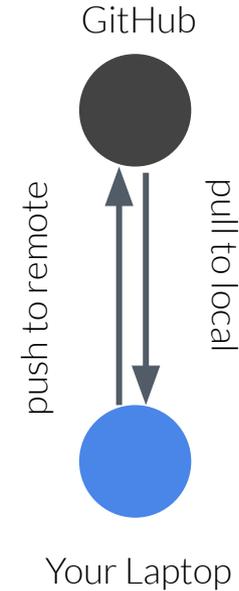
Remote Repo...

Is a repository somewhere else.
Generally GitHub.
Generally upstream.

You **pull** *from* (get their changes)

You **push** *to* (impose your changes)

I always make sure to pull the latest version from my GitHub repo before making changes.



GitHub

- Upstream

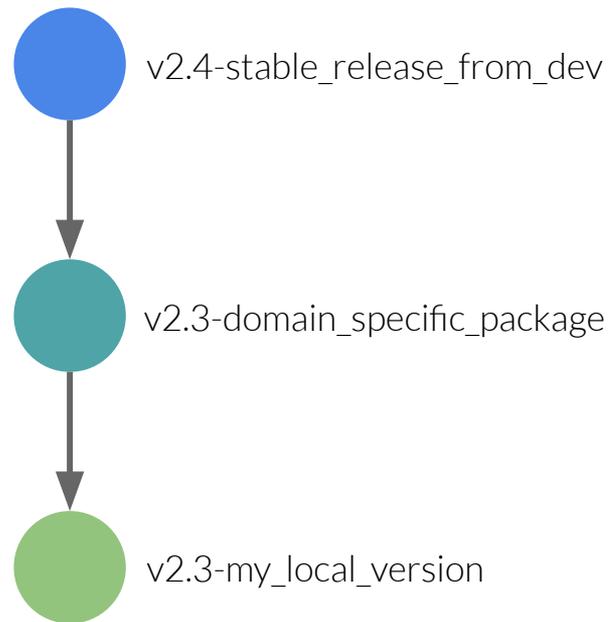
Upstream / Downstream describes a relationship

A repo is **upstream** of you if

- You are 'tracking' it.
- More up to date.
- More official-er.
- Probably not going to adopt your changes.

A repo is **downstream** of you if

- Less up to date.
- Less 'official'.
- Probably going to adopt your changes.



Changes *usually* flow from upstream to downstream.

The developer has made some bugfixes upstream.

Creating a new GitHub repository

Repositories



By default all GitHub Repos are public. (can be *read* and copied by anyone).

No limit to number of public repos.

Private repos have some restrictions (this is how GitHub is monetised).

GitHub

- pull/push

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

Adds a remote called <name> at <url>. Conventionally this is named 'origin'

```
git push <remote> <branch>
```

Apply changes to remote.

You will need the right permissions on the remote.

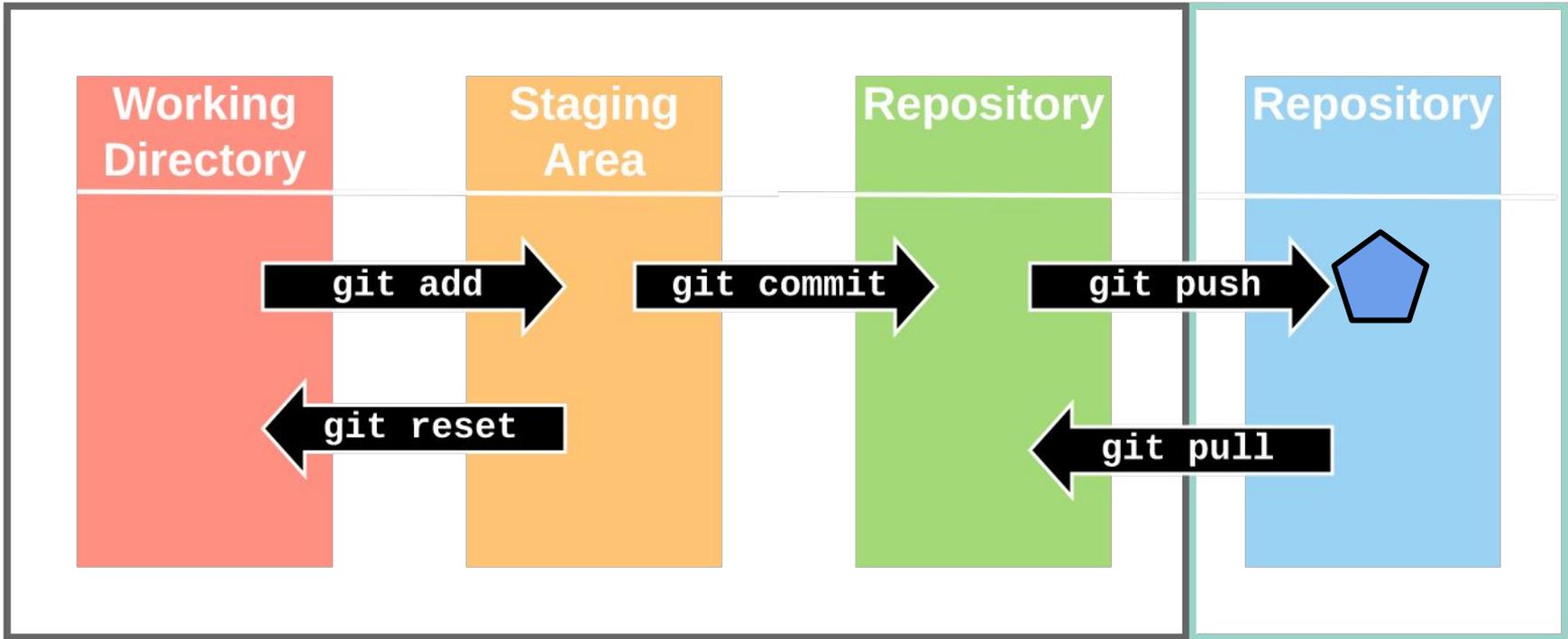
```
git pull <remote> <branch>
```

Get changes from remote (and merge into your current branch).

*A cloned repo will already have a remote called **origin**.*

LOCAL

REMOTE



GitHub

- Playing safe

If you are working by yourself (or team non-concurrently), merge conflicts can be avoided.

`git pull` - Always start with the most current version from upstream.

`git add` - All changes you want to keep.

`git commit` - When finished, with a useful message.

`git push` - To update your remote.



Collaborating

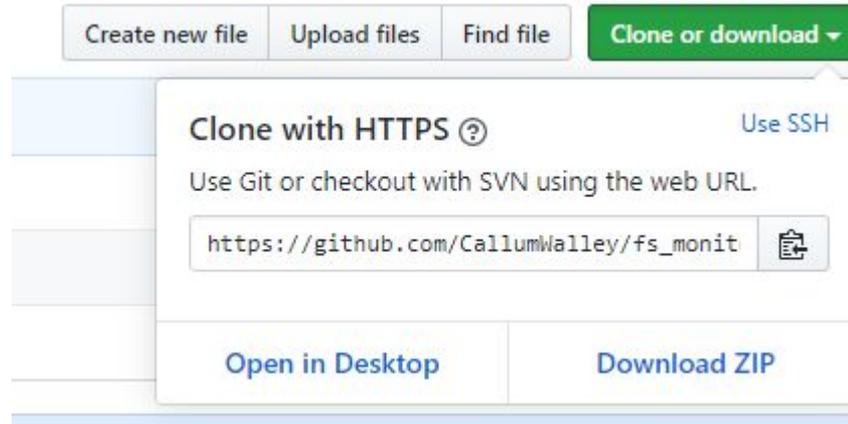
Collaborating

- clone

```
git clone <remote>
```

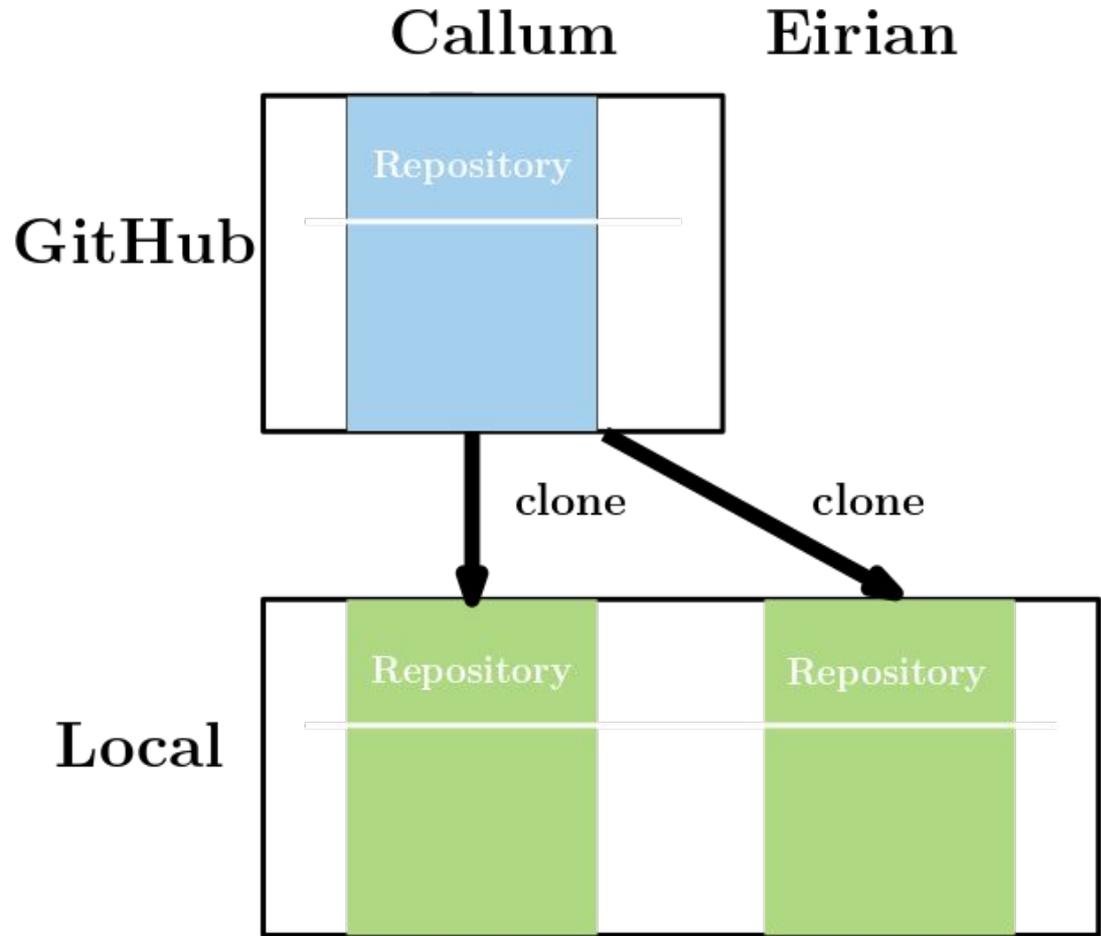
Creates a new directory and copies the repo (and files) from <remote> into it.

When cloning from GitHub you can get the path here.



Collaborating

- clone



Collaborating

- Push

It is *bad practice* to push changes straight to a (remote) master branch.

Extra important if collaborating.

- Keeps untested changes separate and easier to roll back.
- Clearly tracks added features.
- Reduces chances of a merge conflict.

Instead, from a your local push to a *branch* of the remote,

```
git push <remote> <not-master>
```

then go to GitHub and *create a pull request*.

test (2 minutes ago)

Compare & pull request

Pull request is not the same as a push. You are *asking* the *repo* owner to *pull* from you.

Your push to master broke everything! Make a pull request next time please.

Collaborating

- Fork

If you don't have permission to push straight to someone's GitHub repo, use *fork*.



orking a repo creates a new identical GitHub repo.

Clone: **Git** terminology for copying another **Git** repo.

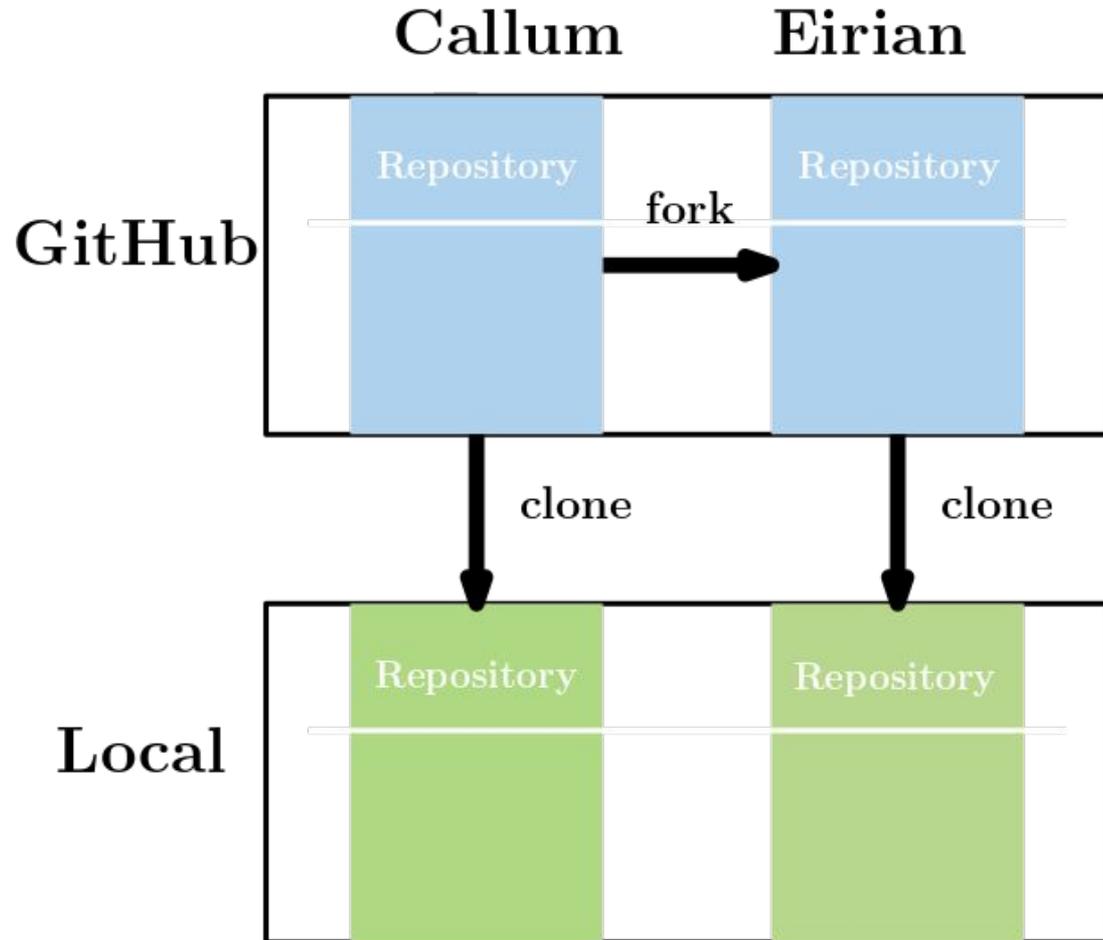
Fork: **GitHub** terminology for making a copy of a **GitHub** repo (copy is owned by you)

Branch: Creates a new branch **inside the same repo**.

I took a fork of your project and fixed all your typos. You're welcome.

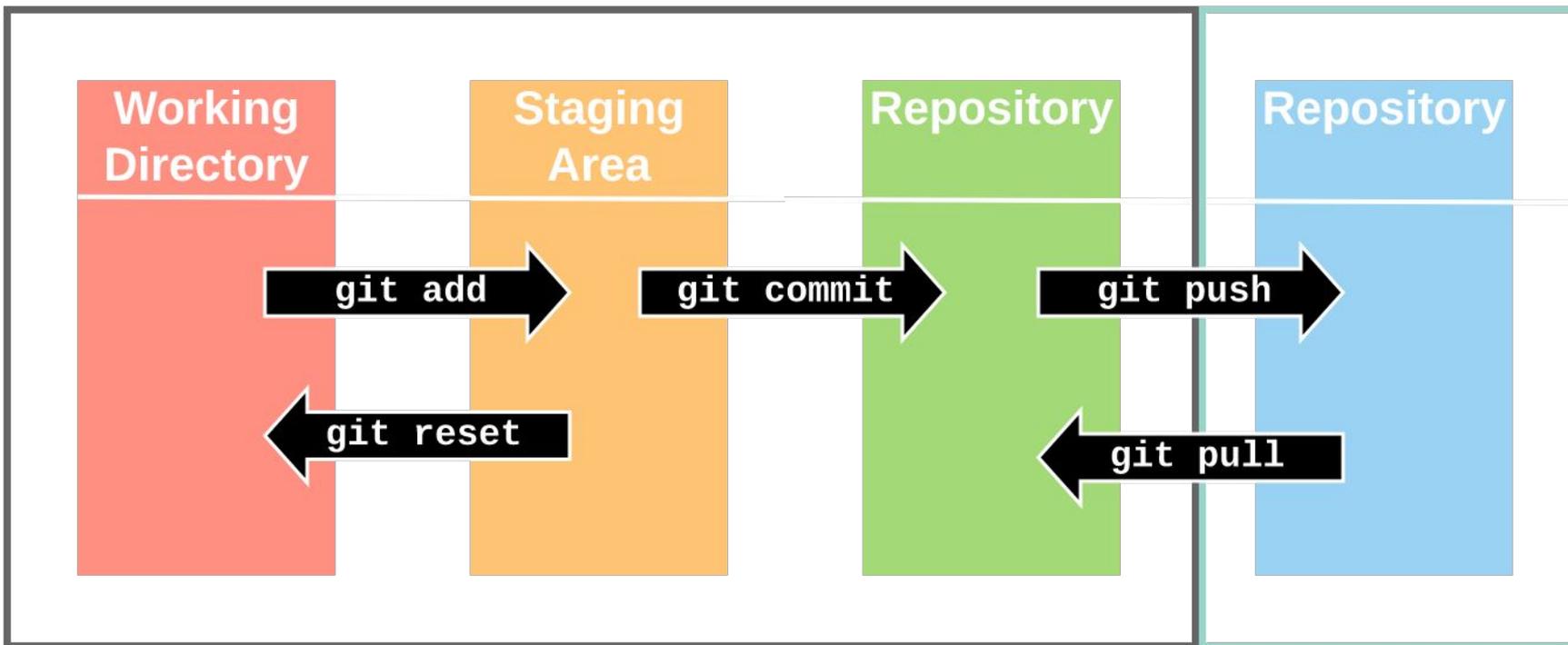
Collaborating

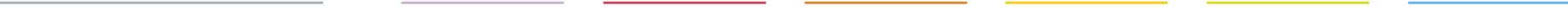
- Fork



LOCAL

REMOTE





Insert shill for CST

The first step of any NeSI consultancy is to set up a collaborative workflow.
Hey! You know how to do that now. You should sign up or whatever.

www.nesi.org.nz/services/consultancy



Fin

