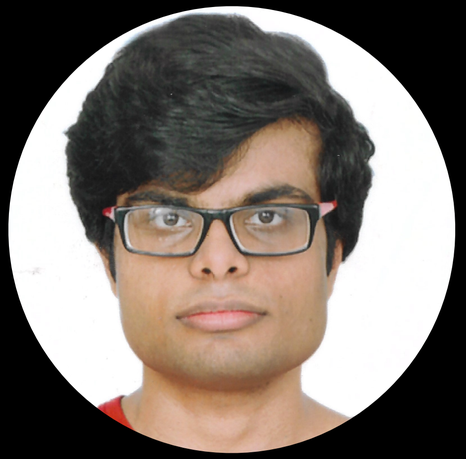
Git Workflows and Best Practices for Open Source Projects



Harikrishnan Balagopal

github.com/HarikrishnanBalagopal

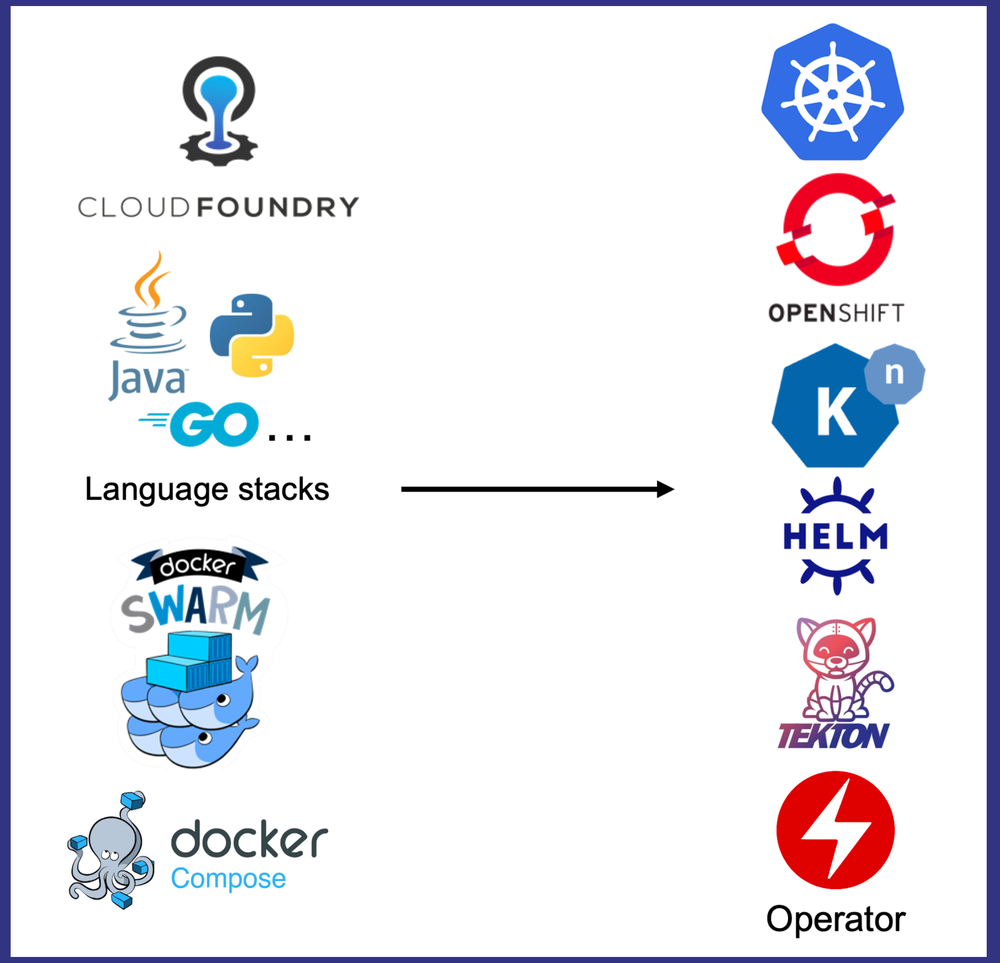
Completed M.Tech in CSE from IIT Kanpur. Thesis was on text generation and NLP.

Joined IBM IRL as a software engineer in August 2020.

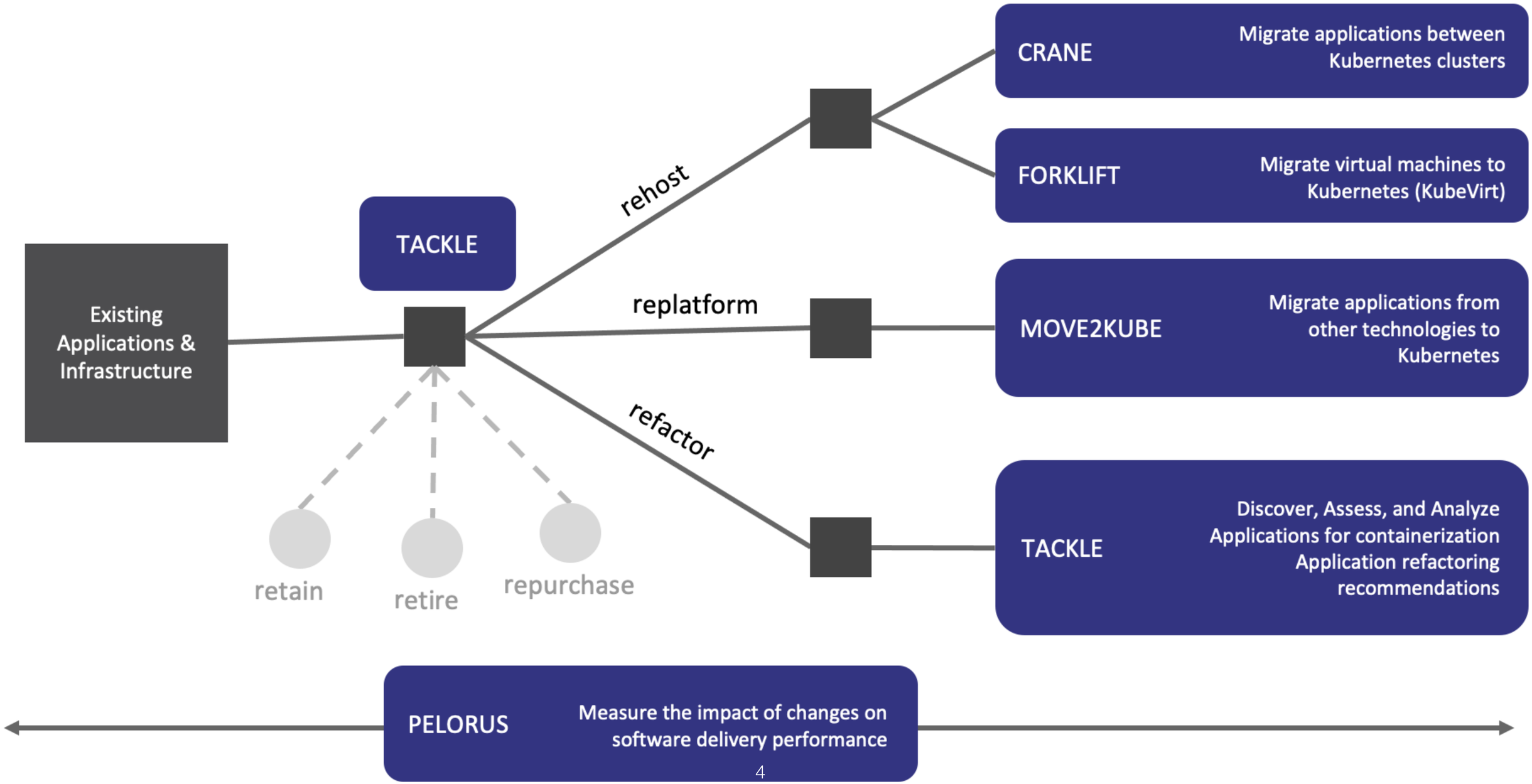
Currently maintaining the open source project Move2Kube.

KONVEYOR
MOVE2KUBE

github.com/konveyor/move2kube



Konveyor Community Projects

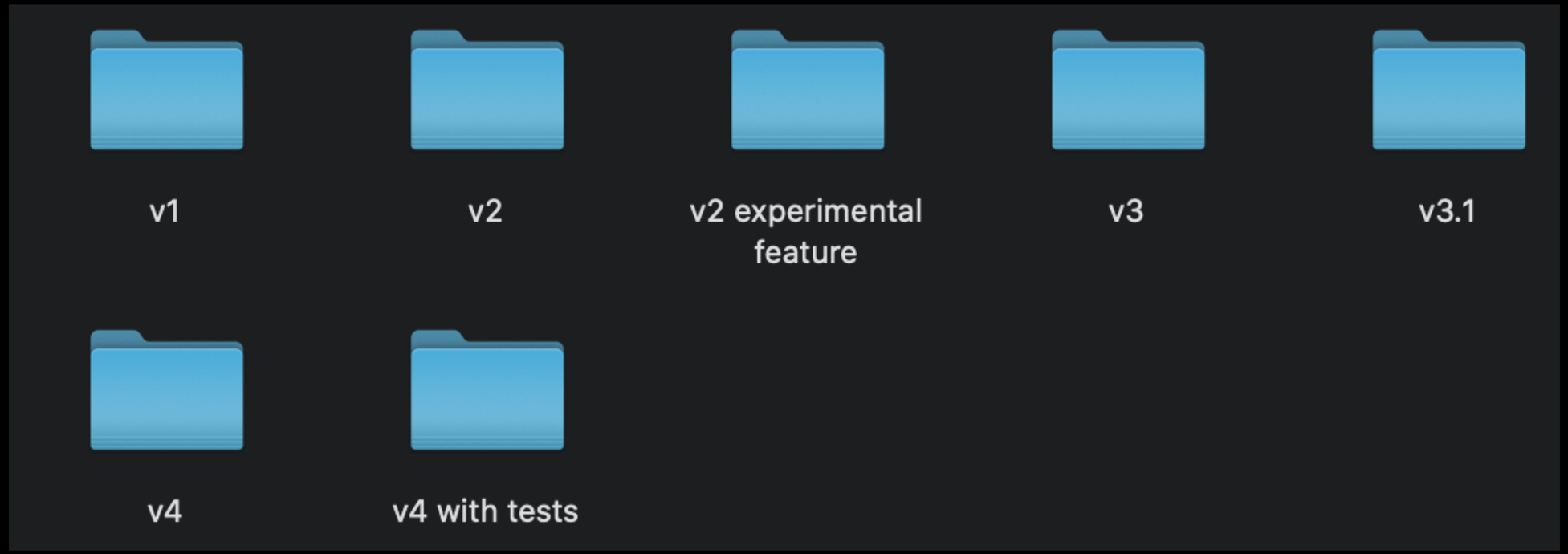


Fundamentals of Git

What is Git?

- Git is by far, the most widely used modern version control system in the world today.
- Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds.
- Git is distributed and is designed with performance, security and flexibility in mind.

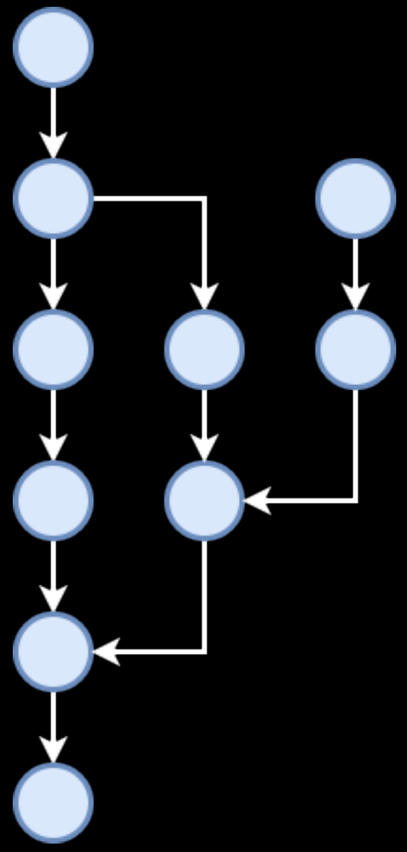
Who has done this before?



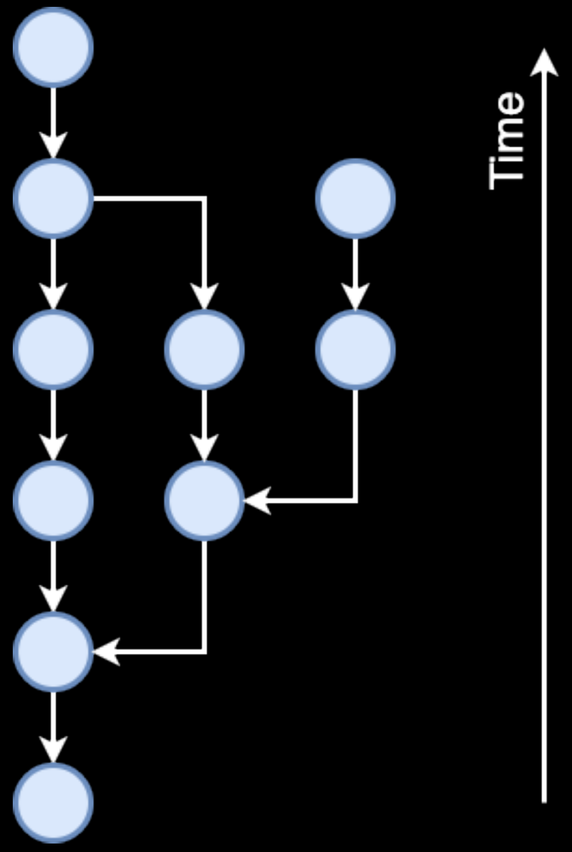
Git lets you maintain multiple versions of your software

- Fast.
- Memory efficient. No duplicate files or folders.
- No need to manually copy and paste files and folders.
- No need to be online, or connect to a server just to work on your code.
- Tries its best to never lose any of your data EVEN when you use the wrong Git commands.
- Easy to quickly switch to a different version of your code and work on some experimental feature or bug fix.
- Battle tested daily by the most demanding projects in the world.

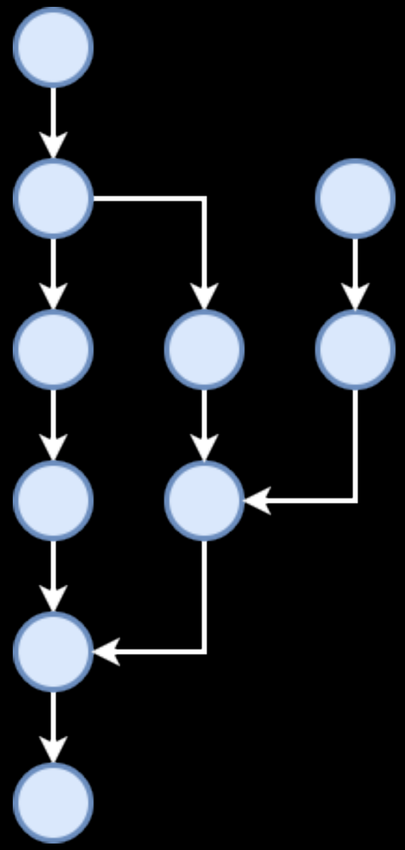
How Git stores your code



How Git stores your code

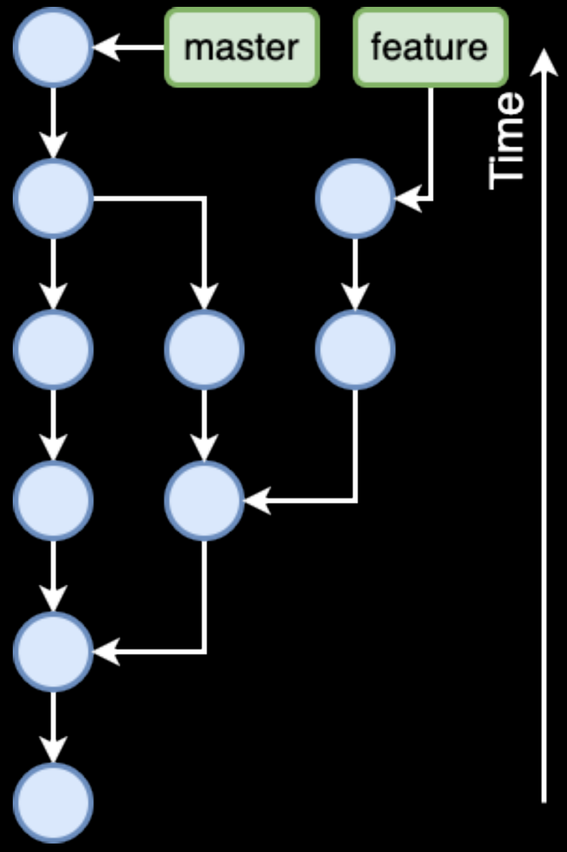


How Git stores your code



- A commit is a snapshot of the entire repository (not a diff).
- Commits are immutable.
- Commits point back in time to their parent commits.

How Git stores your code



Commit

- A commit is a snapshot of the entire repository (not a diff).
- Commits are immutable.
- Commits point back in time to their parent commits.

Branch

- A branch is a pointer to a commit.
- If you are on a branch and you make a new commit then the branch will update to point to the new commit.
- The default branch is usually called `master` or `main` but you can change it to whatever you want.
- You can think of branches as a way to put a pin 📌 in something you are working on and want to return to later.

Git terminology

- Repository (repo): Git stores the code in repositories.
- Local repo: A repo on our local machine (laptop/desktop).
- Remote repo: A repo hosted on some server somewhere.
- upstream: The remote repo we want to contribute to.
  Example: https://github.com/torvalds/linux.git
- origin: Our copy of the upstream repo. This is also a remote repository, usually created by forking the upstream repo on GitHub, GitLab, etc.
  Example: https://github.com/myusername/linux.git

How to get started

2 ways to get started:

- Create a new repo on your local machine (laptop/desktop).
  Go to the directory containing the source code and do `git init`

```
$ pwd
/Users/harikrishnanbalagopal/demo
$ mkdir mylocalrepo
$ cd mylocalrepo/
$ git init
Initialized empty Git repository in /Users/harikrishnanbalagopal/demo/mylocalrepo/.git/
```

- Clone a remote repo (from GitHub, GitLab, etc.)

```
$ git clone git@github.ibm.com:Harikrishnan-Balagopal/git-exercises.git
Cloning into 'git-exercises'...
remote: Enumerating objects: 39, done.
remote: Total 39 (delta 0), reused 0 (delta 0), pack-reused 39
Receiving objects: 100% (39/39), 338.49 KiB | 370.00 KiB/s, done.
Resolving deltas: 100% (8/8), done.
$ cd git-exercises/
$ ls
README.md images
```

Creating a local repo

- `git init` creates a hidden directory called `.git` containing everything that Git uses.
- Any files/directories we add to Git will be hashed and stored under the `objects` directory.
- The file called HEAD contains the branch/tag/commit SHA we have checked out.
- When we create new branches or tags, their metadata will be stored under the `refs` directory.

```
$ tree -a
.
└── .git
    ├── HEAD
    ├── config
    ├── description
    ├── hooks
    │   ├── applypatch-msg.sample
    │   ├── commit-msg.sample
    │   ├── fsmonitor-watchman.sample
    │   ├── post-update.sample
    │   ├── pre-applypatch.sample
    │   ├── pre-commit.sample
    │   ├── pre-merge-commit.sample
    │   ├── pre-push.sample
    │   ├── pre-rebase.sample
    │   ├── pre-receive.sample
    │   ├── prepare-commit-msg.sample
    │   └── update.sample
    ├── info
    │   └── exclude
    ├── objects
    │   ├── info
    │   └── pack
    └── refs
        ├── heads
        └── tags

9 directories, 16 files
```

Git is a content addressable key value store

- We can insert any kind of content into a Git repository, and Git will hand us back a unique key that we can use to retrieve that content later.
- This key is just the SHA-1 hash of the content. So if the content changes the key will also change (hence “content addressable”).
- To turn a folder into a git repository we can use the command `git init`. This creates a hidden folder called `.git` that contains all the information that Git uses.

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git is a content addressable key value store

- We can insert an object using the `git hash-object` command.
- Git will return the 40 character SHA-1 hash.

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

- The object is stored in the `.git/objects` folder.

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

- We can see the content of the object using the `git cat-file` command.

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

- This type of object is called a blob (Binary Large OBject) since it stores the actual content of files.

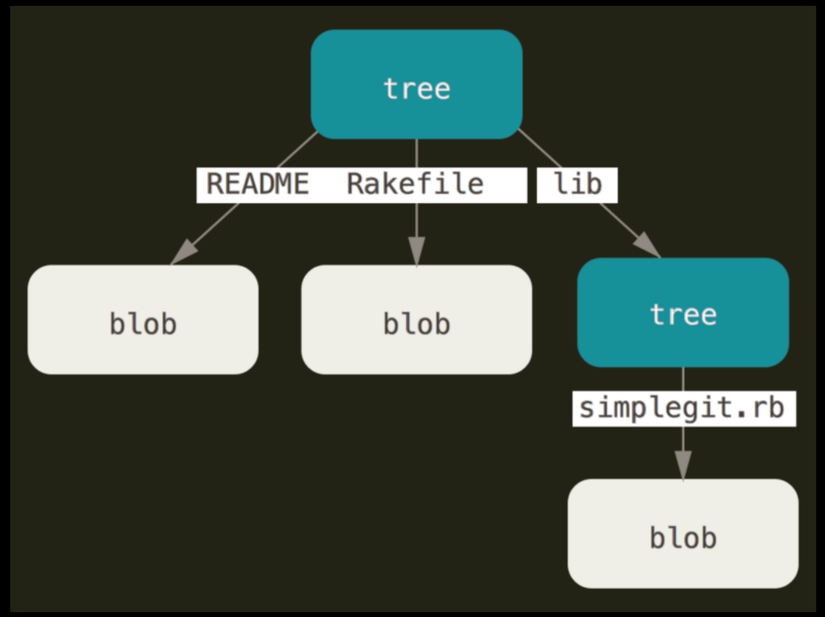
- Git has 2 other types of objects called Trees and Commits.

Git tree

Trees are made up of subtrees and blobs.
Trees correspond to directories and blobs correspond to files.
Trees are also hashed just like everything else in Git.

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```



Git commit

- A tree represent a snapshot of the repo at a point in time. However it doesn't have any information about who saved the snapshot, when it was saved, or why it was saved.

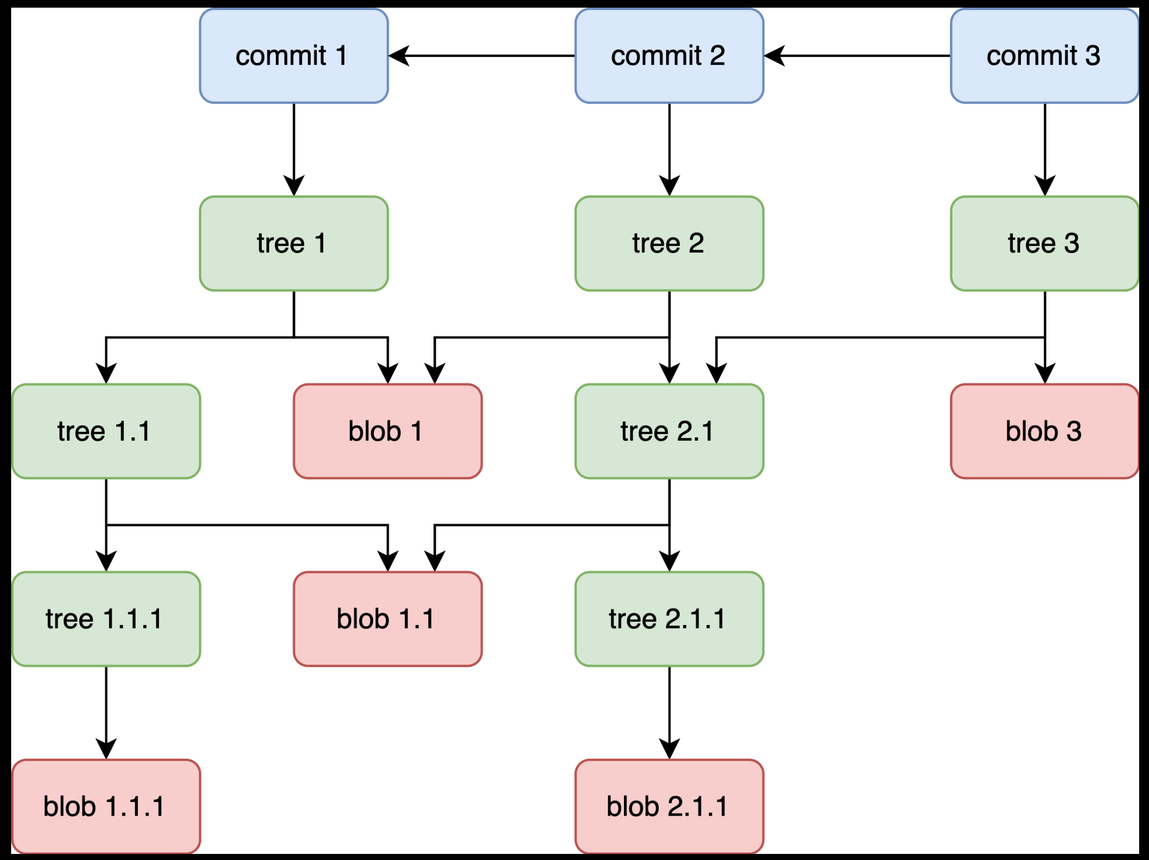
- A commit contains all this extra information. Author, committer, date and time, a commit message giving the reason for the commit, etc.

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

First commit
```

- A commit points to a tree.

Git internal graph

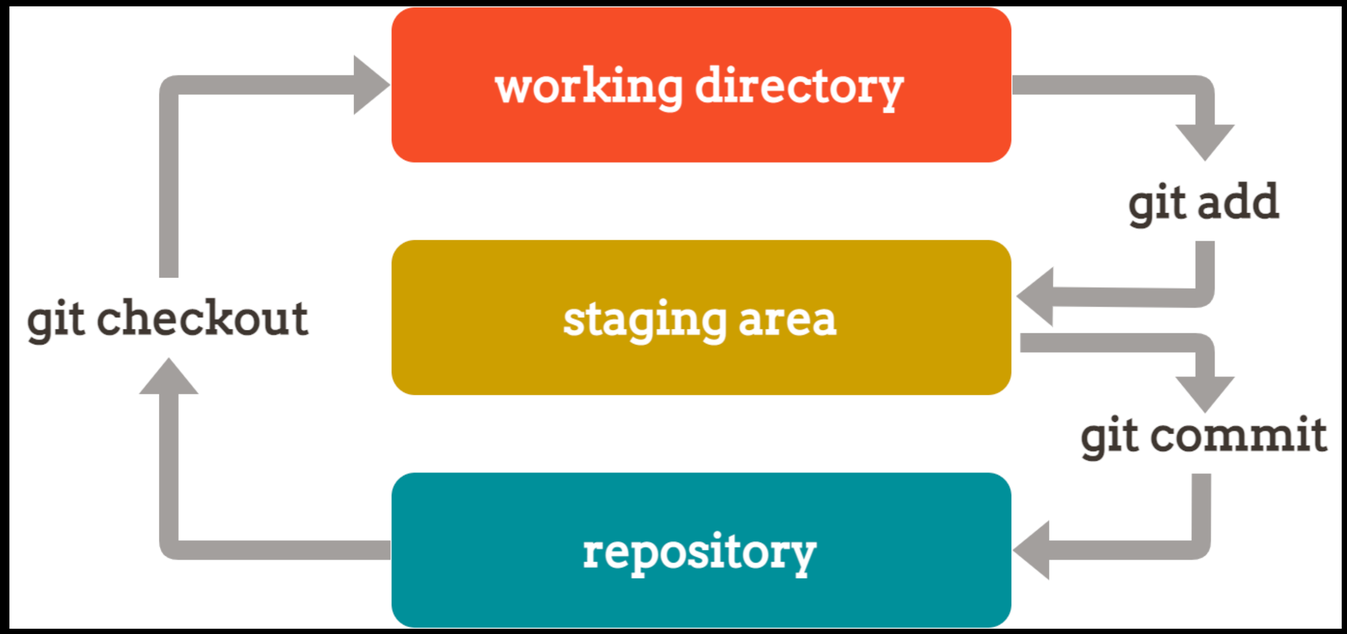
- Putting it all together we get a graph made of commits, trees and blobs.
- There are no cycles in this graph.
- The trees pointed to by later commits can still reference trees and blobs from earlier commits.
- This gives us a persistent data structure without redundancy.



Git commands

3 important areas in Git

- Working directory: This contains the files and directories you have checked out currently on your filesystem (excluding the `.git` directory).
- Staging area (also called Index): This provides a temporary space where you can put changes that you want in the next commit.
- Repository: This is the graph of commits (snapshots) that Git maintains. It contains all the blobs, trees, commits, branches, tags, etc.



Making a commit

- To make a new commit first we do `git add <filename1> <filename2>` to specify the changes we want in the commit.
- These changes are now in the staging area. We can see this by running `git status`
- Then we do `git commit -m <commit message>` to finalize and make the commit itself.

```
$ echo 'just some python code' > foo.py
$ git add foo.py
$ git status
On branch master

No commits yet

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

$ git commit -m 'add new feature foo'
[master (root-commit) 06e6377] add new feature foo
 1 file changed, 1 insertion(+)
 create mode 100644 foo.py
$ git status
On branch master
nothing to commit, working tree clean
$ git log
commit 06e63779b93e795939e999d90db3e0f3110e5c82 (HEAD -> master)
Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
Date:   Thu Dec 17 23:39:20 2020 +0530

    add new feature foo
$
```

Debugging Git

- `git status` tells us what branch we have checked out. It also tells us the differences between the working directory and the last commit.
- `git log —graph —all` shows the entire graph containing the commits, branches, tags, etc.
- Use `git log —graph —all —oneline` to only show the first line of the commit messages.

```
* commit 01ab707786e3a451a8ef531b2f8220475be257f2 (HEAD -> chore/releaseplugin, origin/chore/releaseplugin)
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Thu Dec 17 00:39:53 2020 +0530
|
|     chore: release the plugin when we make a non prerelease
|
|     Signed-off-by: Harikrishnan Balagopal <harikrishmenon@gmail.com>
|
* commit 2719cca3e93f71da0016b07850666f1538fa8822 (upstream/main, origin/main, main)
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Thu Dec 17 00:05:51 2020 +0530
|
|     feat: create the translate plugin for kubectl (#332)
|
|     feat: create the translate plugin for kubectl
|
|     The translate plugin is a much simpler version
|     of the translate command in move2kube.
|
|     Also refactored move2kube commands to
|     allow it to be used as a plugin.
|
|     Signed-off-by: Harikrishnan Balagopal <harikrishmenon@gmail.com>
|
* commit 0c28344efc1ee3fa63e20b4628fd2472dcc274bf
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Tue Dec 15 22:37:51 2020 +0530
|
|     chore: add each valid PR type to a group in release notes (#331)
|
|     Signed-off-by: Harikrishnan Balagopal <harikrishmenon@gmail.com>
|
* commit 17d7e484555501050473f851ba895efcf4eff3534
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Tue Dec 15 21:55:53 2020 +0530
|
|     ci: added support for specifying commit and previous tag while releasing (#330)
|
|     For major releases we might want to have the changes from all
```

Creating and deleting a branch

- `git branch` lists the branches
- `git branch <branch name>` creates a new branch pointing to the commit we are currently on.
- `git branch -d <branch name>` to delete a branch in your local repo.
- `git branch -rd <branch name>` to delete a branch we fetched from a remote. Note: This doesn't delete it on the remote repo. To delete it on the remote repo: `git push -d <remote_name> <branch_name>`
- To create a new branch and immediately check it out we can use `git checkout -b <branch name>`

```
$ git branch
* master
$ git branch my-feature
$ git branch
* master
  my-feature
```

Creating a commit on a branch

- In order to create a commit on a branch we need to checkout that branch using `git checkout <branch name>`
- When you create a new commit, Git will automatically move the branch you have checked out to point to the new commit.

```
$ git branch
* master
$ git branch my-feature
$ git branch
* master
  my-feature
$ git checkout my-feature
D       foo.py
Switched to branch 'my-feature'
$ git branch
  master
* my-feature
$ git log --graph --all
* commit 06e63779b93e795939e999d90db3e0f3110e5c82 (HEAD -> my-feature, master)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Dec 17 23:39:20 2020 +0530

      add new feature foo
$ echo 'some more code' > bar.py
$ git add bar.py && git commit -m 'add code for feature bar'
[my-feature dd9b391] add code for feature bar
 1 file changed, 1 insertion(+)
 create mode 100644 bar.py
$ git log --graph --all
* commit dd9b39149852ead8a42669926ac7d0e49eadd51c (HEAD -> my-feature)
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Fri Dec 18 11:51:58 2020 +0530
|
|     add code for feature bar
|
* commit 06e63779b93e795939e999d90db3e0f3110e5c82 (master)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Dec 17 23:39:20 2020 +0530

      add new feature foo
```

Ignoring files and folders

- Usually we have some files and folders that we don’t want to commit (build output, `node_modules`, etc.).
- Create a `.gitignore` file in the base of the repo containing all the paths that should be ignored.
- It is also possible to have multiple `.gitignore` files (one per folder). Each `.gitignore` file can have paths relative to the file itself.

```
$ ls
index.js          node_modules      package-lock.json package.json
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.js
        node_modules/
        package-lock.json
        package.json

nothing added to commit but untracked files present (use "git add" to track)
$ echo 'node_modules/' > .gitignore
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        index.js
        package-lock.json
        package.json

nothing added to commit but untracked files present (use "git add" to track)
$ git add .gitignore index.js package.json package-lock.json
$ git commit -m 'some commit message'
```

Creating and deleting a tag

- A tag is a pointer to a commit.
- `git tag` lists all the tags.
- `git tag <tag name>` creates a new tag pointing to the commit we are currently on. `git tag -d <tag name>` deletes the tag.

```
$ git tag
$ git tag my-tag-1
$ git tag
my-tag-1
$ git log --graph --all
* commit dd9b39149852ead8a42669926ac7d0e49eadd51c (HEAD -> my-feature, tag: my-tag-1)
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Fri Dec 18 11:51:58 2020 +0530
|
|     add code for feature bar
|
* commit 06e63779b93e795939e999d90db3e0f3110e5c82 (master)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Dec 17 23:39:20 2020 +0530

      add new feature foo
```

Difference between a branch and a tag

- Similarities: Both are pointers to a commit.
- When you checkout a tag and make a commit the tag will not move.
- When you checkout a branch and make a commit the branch will move to point to the new commit.
- Note: Only the branch you have checked out will move. All other branches stay in place.

```
$ git log --graph --all --oneline
* dd9b391 (HEAD -> branch1, tag: tag2, tag: tag1, branch2) add code for feature bar
* 06e6377 (master) add new feature foo
$ echo 'even more code' >> foo.py
$ git add foo.py && git commit -m 'add more code to foo.py'
[branch1 9aeb1c5] add more code to foo.py
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --graph --all --oneline
* 9aeb1c5 (HEAD -> branch1) add more code to foo.py
* dd9b391 (tag: tag2, tag: tag1, branch2) add code for feature bar
* 06e6377 (master) add new feature foo
```

Difference between a branch and a tag

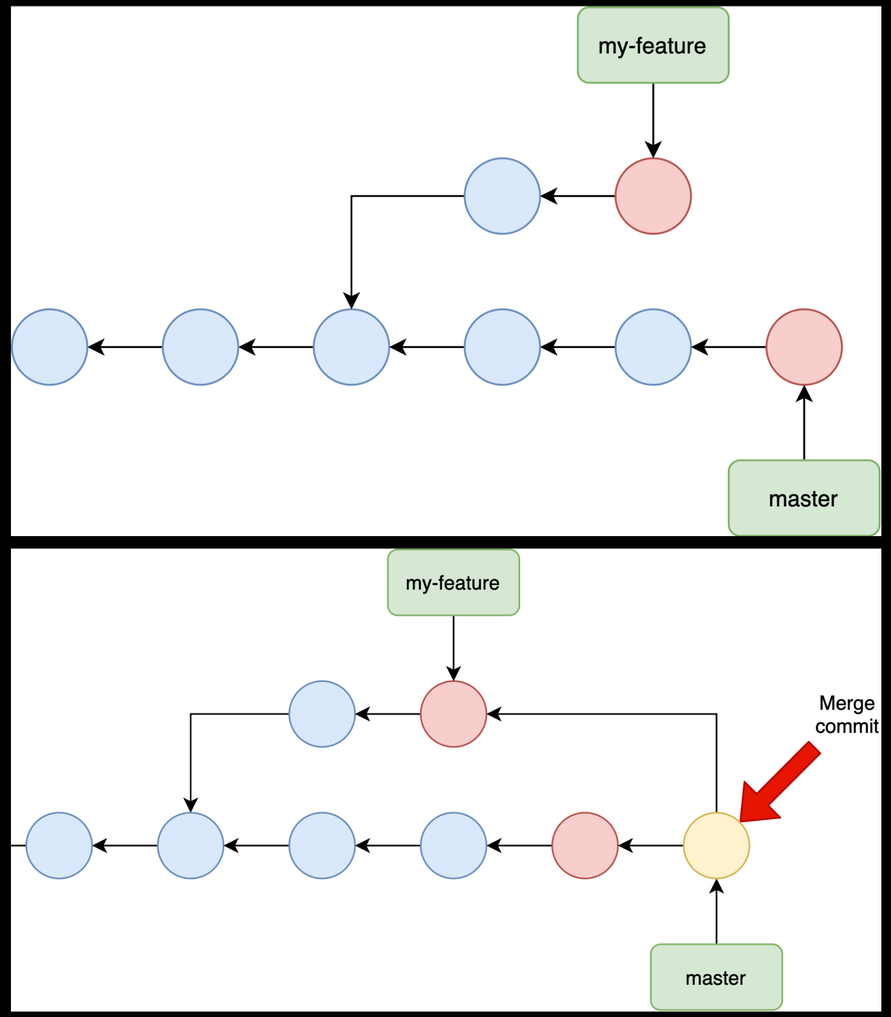
- Tags can be lightweight or annotated.
- Annotated tags have a tagger, time, and a message associated with them (Similar to commits).
- `git tag -a <tag name> -m <message>` to create an annotated tag.

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

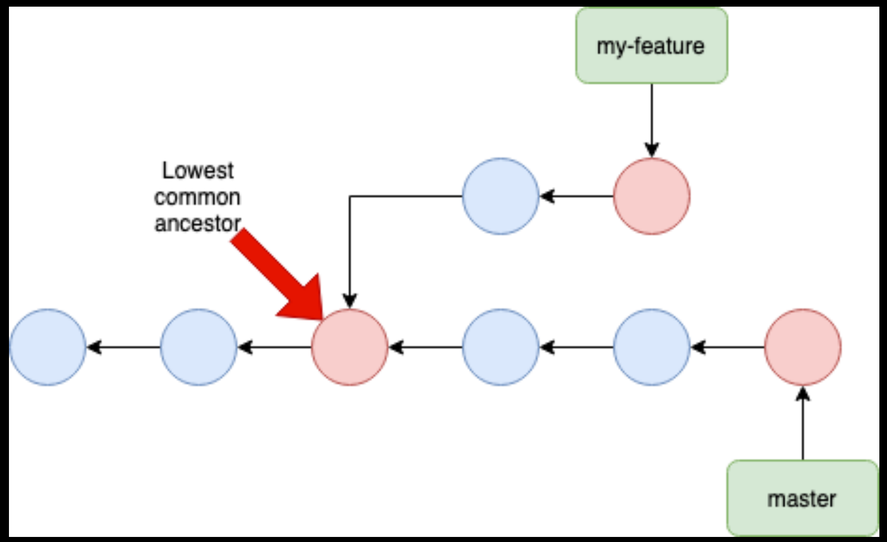
Merging in Git

- When we start working on a new feature, we create a branch and make some commits on it.
- When we are done working on our feature and want to add it to the master branch, we can do a merge.
- To do a merge:
  1. `git checkout <branch you want to merge into>`
  2. `git merge <branch you want to merge from>`
- This will create a new merge commit that contains the changes from both branches.



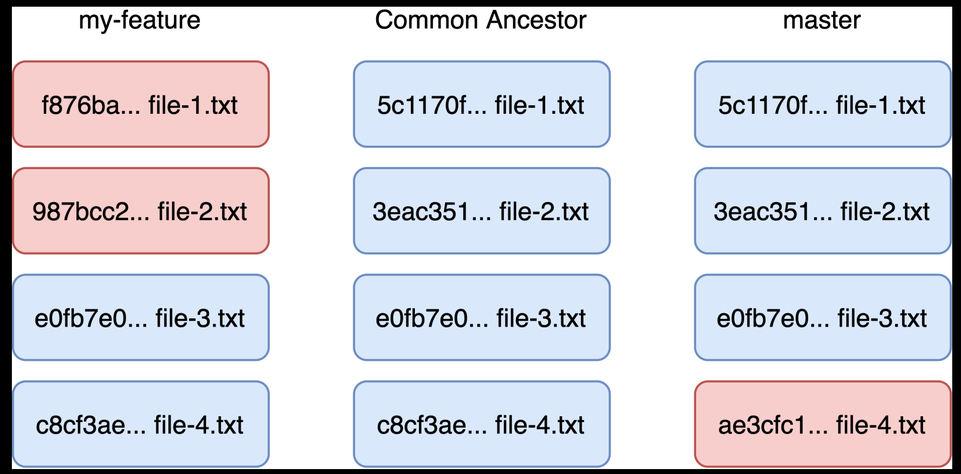
Merging in Git

- Git does what is known as a 3-way merge.
- Given 2 commits to merge, git finds a 3rd commit that is the lowest common ancestor of the 2 commits.
- Then it compares all 3 commits to each other to decide what should go in the final merged commit.



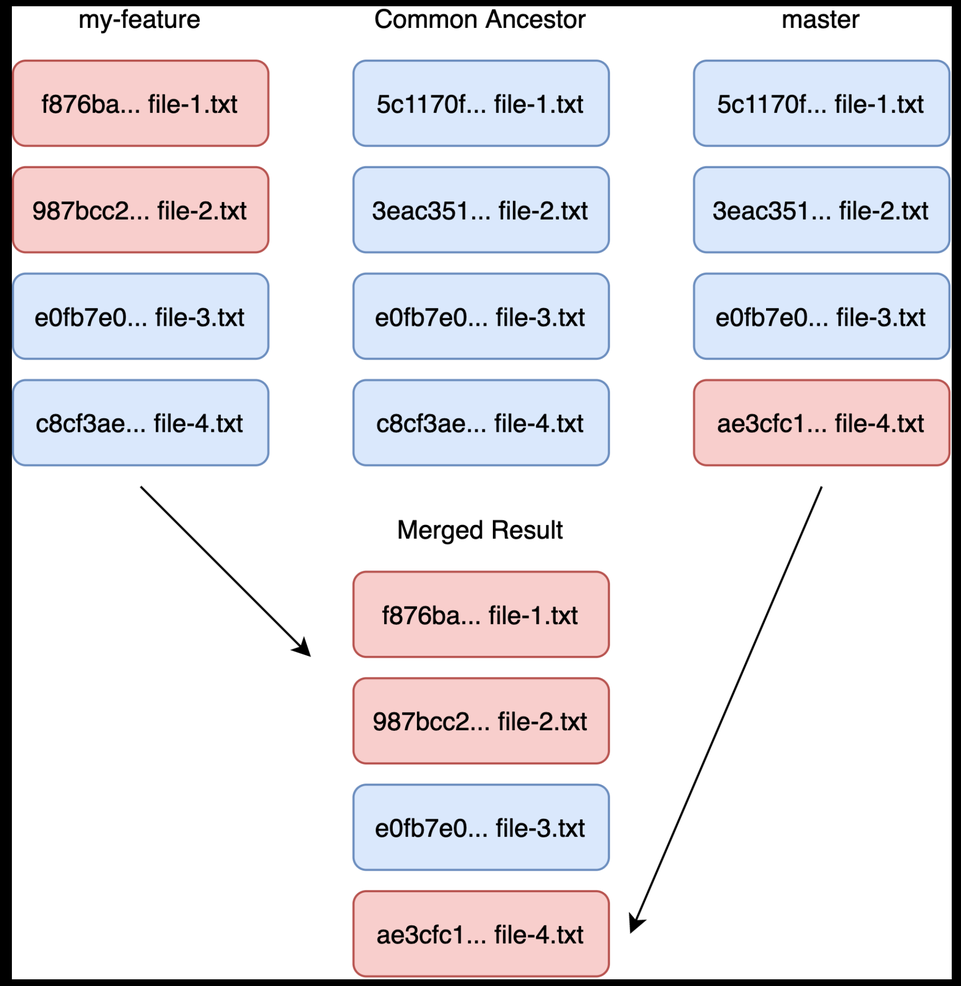
Merging in Git

- We go through each file in the common ancestor commit and compare it to the corresponding file in the other 2 commits to see if it was changed.
- The files that haven't changed can be added to the merge commit as is.



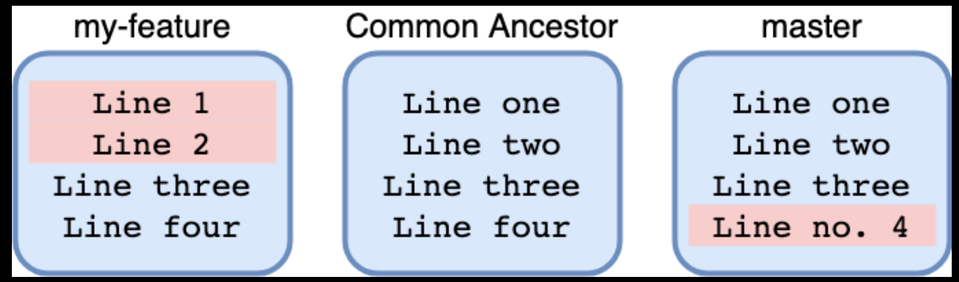
Merging in Git

- In case the file was changed in exactly one branch, then we add the changed version to the merge commit.
- This makes sense since we want to keep as many of the changes that were made in each branch.
- If a new file was created in exactly one branch then that file also gets added to the merge commit. Example: If the `my-feature` branch created a file called `foo.py` and the other branch didn't, then `foo.py` can be added to the merge commit as is.



Merging in Git

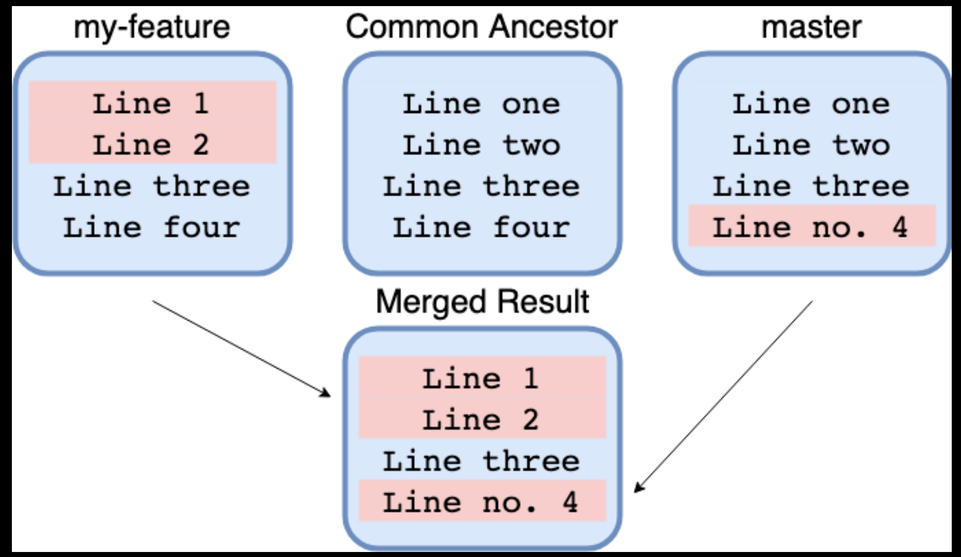
- For the files that were changed in both branches we need to look at their contents and go line by line.
- Git compares each file to the same file in the common ancestor.
- Git uses a diffing algorithm to compare the files and come up with a list of changes.
- Git has built-in support for 4 different diffing algorithms:
  patience, minimal, histogram, and myers.
- By default Git uses the Myers algorithm.



Merging in Git

- The lines that haven't changed can be added to the final merged file as is.

- In case the line was changed in exactly one branch then we add the changed version to the merge commit.
  This includes the addition and deletion of lines.



Merging in Git

- In case the same line was changed in both branches then we have a merge conflict.
- Git will add the changes from both branches into the final file with some additional markers: <<<<<<<, =======, >>>>>>>

```
<<<<<<< HEAD
This is the change in master
=======
This is the change in my-feature
>>>>>>> my-feature
```

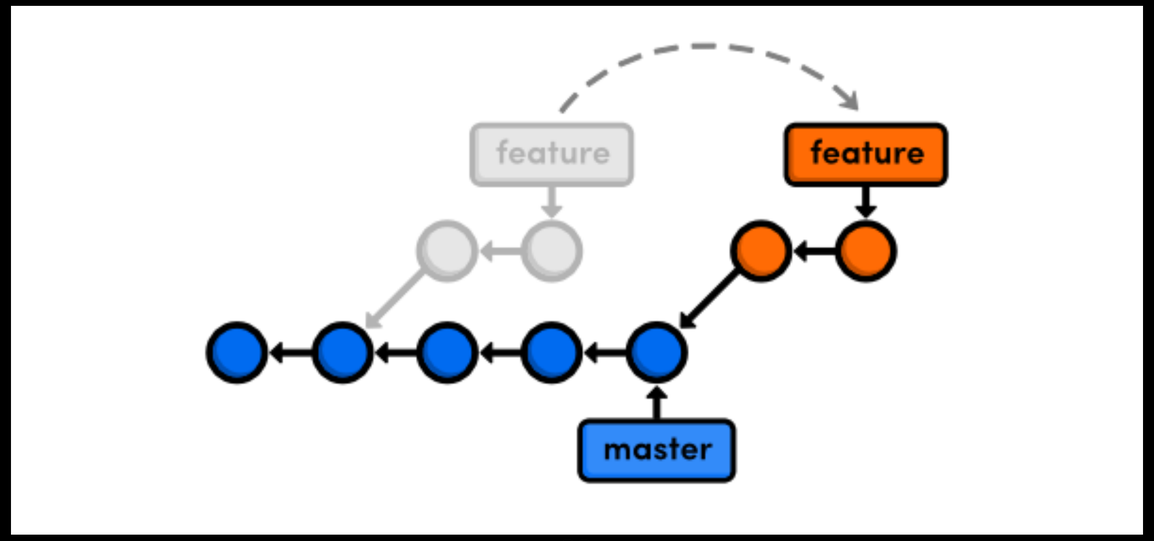
Merging in Git

- In order to proceed with the merge the user must fix all the conflicts by editing the parts of the files that are in conflict, saving the file, and adding it to the staging area with `git add`
- Once all the conflicts have been resolved, the merge can proceed by creating the merge commit containing all the changes that were added in the previous steps.

```
<<<<<<< HEAD
This is the change in master
=======
This is the change in my-feature
>>>>>>> my-feature
```

Rebasing commits

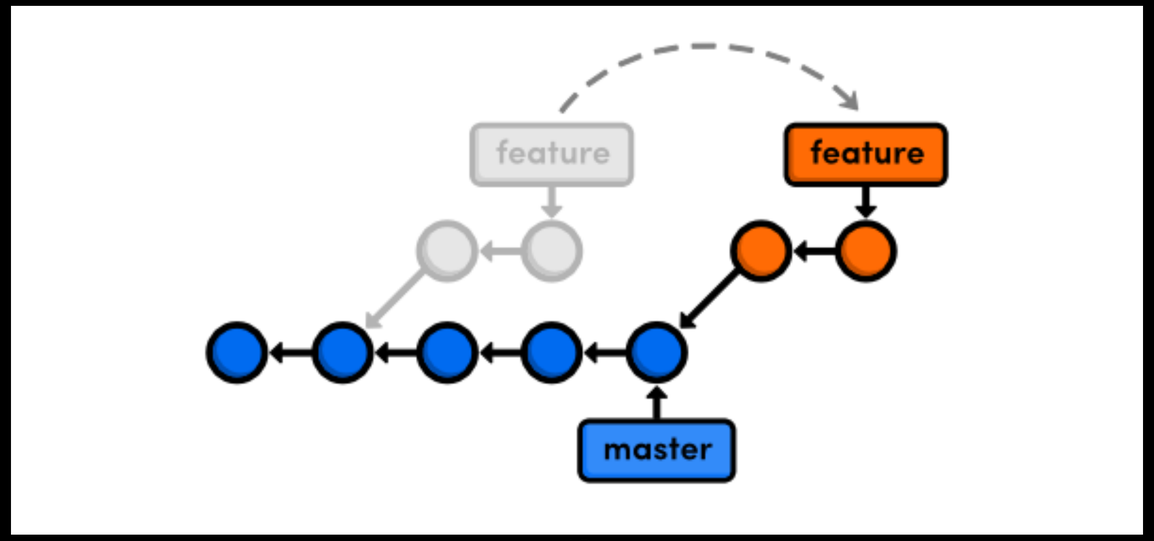
- Rebasing in Git is a very powerful operation that can do a lot of different things.
- It can be done interactively and non interactively.
- Here we will focus on rebasing in the sense of changing the root commit that a set of commits are based on. This is an alternative to the merging we saw earlier.



Rebasing commits

- To rebase a branch:
  1. `git checkout <branch name>`
  2. `git rebase <new base branch>`

- Example:
  `git checkout feature`
  `git rebase master`



Rebase	Merge
Rebasing gives a linear commit history.	Merging results in a complicated graph with many diverging and converging branches.
Rebasing does not require extra commits.	Merging leads to the creation of extra merge commits that clutter the history.
In the worst case, rebasing might require a merge conflict resolution session per commit.	Worst case only a single merge conflict resolution session is required.
Rebasing rewrites history and destroys chronological order.	Preserves complete history and chronological order.

Squashing commits

- Let’s say we are in the situation shown on the right.
- We added a new feature.
- We made some fixes to it.
- We have 3 new commits but, it would be nice if there was only 1 new commit containing the feature and all the fixes.

```
$ echo 'some foo code' > foo.py
$ git add foo.py && git commit -m 'new feature'
[master c2f0c73] new feature
 1 file changed, 1 insertion(+)
 create mode 100644 foo.py
$ echo 'some fixes' >> foo.py
$ git add foo.py && git commit -m 'fix some bugs'
[master dd10eb5] fix some bugs
 1 file changed, 1 insertion(+)
$ echo 'more fixes' >> foo.py
$ git add foo.py && git commit -m 'fix formatting'
[master f6ad28a] fix formatting
 1 file changed, 1 insertion(+)
$ git log --graph --all --oneline
* f6ad28a (HEAD -> master) fix formatting
* dd10eb5 fix some bugs
* c2f0c73 new feature
* 45d7b3c intial commit
```

Squashing commits

- `git rebase -i <new base or parent commit>` can be used to squash a bunch of commits into a single commit.
- In this case we will do: `git rebase -i 45d7b3c` since we want the new squashed commit to be based on `45d7b3c`
- The `-i` flag means interactive.

```
$ echo 'some foo code' > foo.py
$ git add foo.py && git commit -m 'new feature'
[master c2f0c73] new feature
 1 file changed, 1 insertion(+)
 create mode 100644 foo.py
$ echo 'some fixes' >> foo.py
$ git add foo.py && git commit -m 'fix some bugs'
[master dd10eb5] fix some bugs
 1 file changed, 1 insertion(+)
$ echo 'more fixes' >> foo.py
$ git add foo.py && git commit -m 'fix formatting'
[master f6ad28a] fix formatting
 1 file changed, 1 insertion(+)
$ git log --graph --all --oneline
* f6ad28a (HEAD -> master) fix formatting
* dd10eb5 fix some bugs
* c2f0c73 new feature
* 45d7b3c intial commit
$ git rebase -i 45d7b3c
```

Squashing commits

- This opens up the editor since we chose to rebase interactively.
- At the top, we can see the commits involved in the rebase.
- Below that, there are some comments. The commands that we can use are listed here.
- For each commit we need to specify which command to use.

```
pick c2f0c73 new feature
pick dd10eb5 fix some bugs
pick f6ad28a fix formatting
# Rebase 45d7b3c..f6ad28a onto 45d7b3c (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Squashing commits

- There are 2 commands related to squashing: `squash` and `fixup`
- `squash` will squash the commit keeping the commit message.
- `fixup` will squash getting rid of the commit message.
- Each command also has a shorthand: s for `squash`, f for `fixup`, etc.

```
pick c2f0c73 new feature
pick dd10eb5 fix some bugs
pick f6ad28a fix formatting

# Rebase 45d7b3c..f6ad28a onto 45d7b3c (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Squashing commits

- Here we choose `fixup` for the 2nd and 3rd commit.
- We will leave the 1st commit as `pick` since we want to keep its commit message.
- Note: we could also have chosen `reword` for the 1st commit if we wanted to change the final commit message.

```
pick c2f0c73 new feature
f dd10eb5 fix some bugs
f f6ad28a fix formatting

# Rebase 45d7b3c..f6ad28a onto 45d7b3c (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Squashing commits

- Save and quit the editor.
- We can see that Git created a new squashed commit that contains the changes from all 3 commits.
- The commit message is same as the first commit since we used `fixup` to throw away the commit messages of the other 2 commits.

```
$ git rebase -i 45d7b3c
Successfully rebased and updated refs/heads/master.
$ git log --graph --all --oneline
* 152b4f8 (HEAD -> master) new feature
* 45d7b3c intial commit
```

Adding remote repos

- `git remote -v` lists the remote repos you have added.
- `git remote add <remote_name> <repo_url>` adds a new remote.
- The convention is to use the name “upstream” for the original repo and “origin” for your fork of that repo on GitHub, GitLab, etc.
- `git remote remove <remote_name>` removes a remote.

```
$ git remote -v
$ git remote add upstream git@github.com:torvalds/linux.git
$ git remote -v
upstream        git@github.com:torvalds/linux.git (fetch)
upstream        git@github.com:torvalds/linux.git (push)
$ git remote add origin git@github.com:HarikrishnanBalagopal/linux.git
$ git remote -v
origin  git@github.com:HarikrishnanBalagopal/linux.git (fetch)
origin  git@github.com:HarikrishnanBalagopal/linux.git (push)
upstream        git@github.com:torvalds/linux.git (fetch)
upstream        git@github.com:torvalds/linux.git (push)
$ git remote remove origin
$ git remote -v
upstream        git@github.com:torvalds/linux.git (fetch)
upstream        git@github.com:torvalds/linux.git (push)
$
```

Fetching from remote repos

- Simply adding a remote doesn’t fetch any data from the remote.
- `git fetch <remote_name>` fetches commits and branches from the remote.
- `git fetch —all` to fetch data from all the remotes.

```
$ git log --graph --all
* commit 63091a713cd0b4d2ddafbf78d925e4a093ca31c5 (HEAD -> main, origin/main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Sat Feb 20 22:30:19 2021 +0530

      initial commit
$ git remote -v
origin  git@github.com:HarikrishnanBalagopal/myapp.git (fetch)
origin  git@github.com:HarikrishnanBalagopal/myapp.git (push)
$ git fetch --all
Fetching origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 4.49 KiB | 1.12 MiB/s, done.
From github.com:HarikrishnanBalagopal/myapp
   63091a7..5eceb6a  main       -> origin/main
$ git log --graph --all
* commit 5eceb6a5a9cb46bd473154d65bc18808b0ffc81c (origin/main)
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Sat Feb 20 22:32:54 2021 +0530
|
|     added a license
|
* commit 63091a713cd0b4d2ddafbf78d925e4a093ca31c5 (HEAD -> main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Sat Feb 20 22:30:19 2021 +0530

      initial commit
$
```

Syncing with remote repos

- After fetching, the new commits and branches show up in our local repo's history. However all the local branches remain where they were.
- We can move our local branch to point to the same commit as the remote branch with a fast forward merge:
  `git merge —ff-only <remote_name>/<branch_name>'
- We can do the same with a rebase:
  `git rebase <remote_name>/<branch_name>'
- Alternatively `git pull` does both a fetch and a merge in one go.

```
$ git log --graph --all
* commit 5eceb6a5a9cb46bd473154d65bc18808b0ffc81c (origin/main)
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Sat Feb 20 22:32:54 2021 +0530
|
|     added a license
|
* commit 63091a713cd0b4d2ddafbf78d925e4a093ca31c5 (HEAD -> main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Sat Feb 20 22:30:19 2021 +0530

      initial commit
$ git merge --ff-only origin/main
Updating 63091a7..5eceb6a
Fast-forward
 LICENSE | 201 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 201 insertions(+)
 create mode 100644 LICENSE
$ git log --graph --all
* commit 5eceb6a5a9cb46bd473154d65bc18808b0ffc81c (HEAD -> main, origin/main)
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Sat Feb 20 22:32:54 2021 +0530
|
|     added a license
|
* commit 63091a713cd0b4d2ddafbf78d925e4a093ca31c5
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Sat Feb 20 22:30:19 2021 +0530

      initial commit
$
```

Pushing changes to remotes

- `git push` pushes the new commits on the local branch to the corresponding branch on the remote.
- In order for `git push` to work we need to configure a remote branch:
  `git branch —set-upstream-to=<remote_name>/<branch_name`
- You can also configure the remote while pushing:
  `git push —set-upstream <remote_name> <branch_name>`

```
$ git checkout -b add-gitignore
Switched to a new branch 'add-gitignore'
$ echo '.vscode/' > .gitignore
$ git add -A && git commit -m 'added gitignore'
[add-gitignore cae0e44] added gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
$ git log --graph --all --oneline
* cae0e44 (HEAD -> add-gitignore) added gitignore
* 5eceb6a (origin/main, main) added a license
* 63091a7 initial commit
$ git push
fatal: The current branch add-gitignore has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin add-gitignore

$ git push --set-upstream origin add-gitignore
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 989 bytes | 989.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'add-gitignore' on GitHub by visiting:
remote:      https://github.com/HarikrishnanBalagopal/myapp/pull/new/add-gitignore
remote:
To github.com:HarikrishnanBalagopal/myapp.git
 * [new branch]      add-gitignore -> add-gitignore
Branch 'add-gitignore' set up to track remote branch 'add-gitignore' from 'origin'
$ git log --graph --all --oneline
* cae0e44 (HEAD -> add-gitignore, origin/add-gitignore) added gitignore
* 5eceb6a (origin/main, main) added a license
* 63091a7 initial commit
```

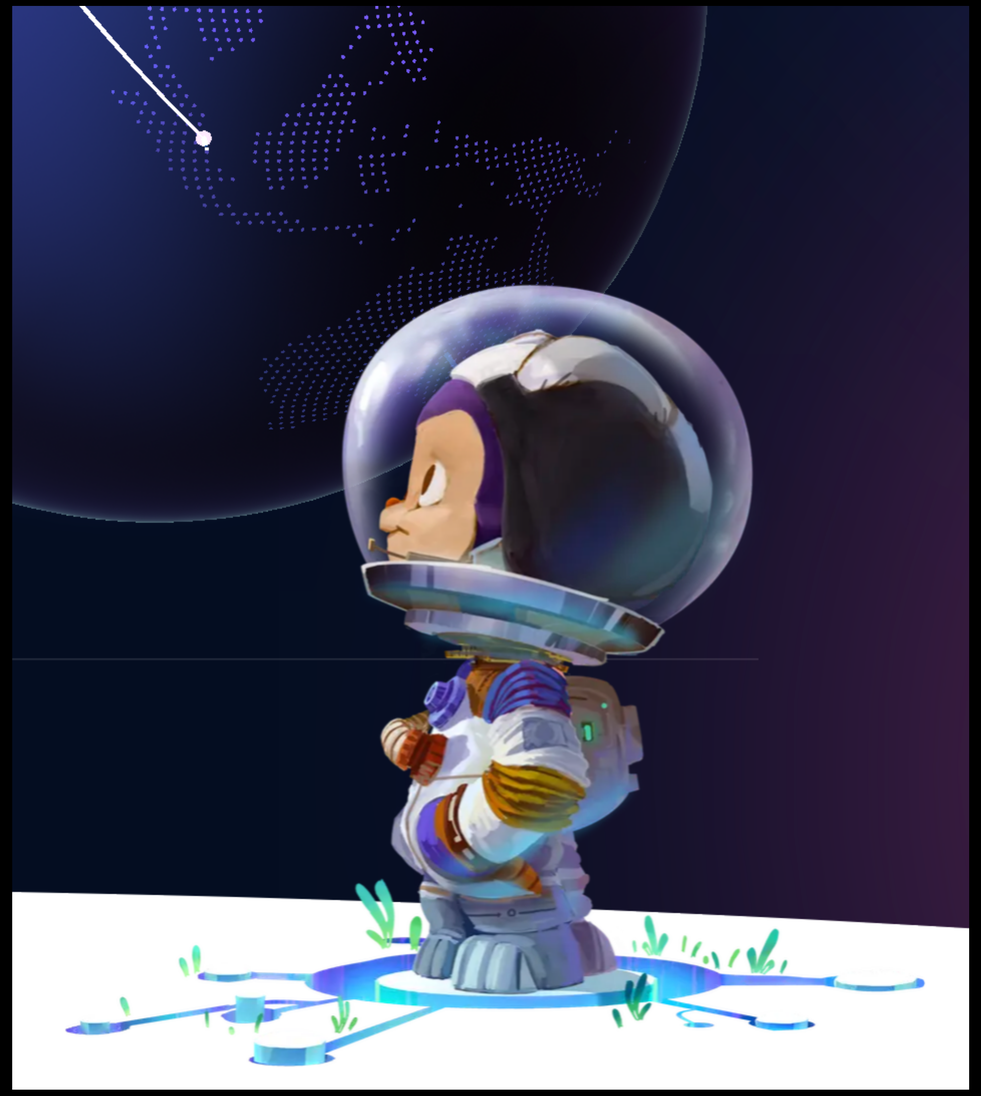
Git hosting and workflows

Git hosting services

- Git is NOT GitHub.
- GitHub, GitLab, BitBucket, etc. are websites that offers free hosting for remote git repositories.
- GitHub Enterprise and GitLab are software that can be hosted on your own servers.
  For example: IBM runs a GitHub Enterprise instance on github.ibm.com for code internal to the company.
- GitLab is open source software. GitHub Enterprise is licensed software.

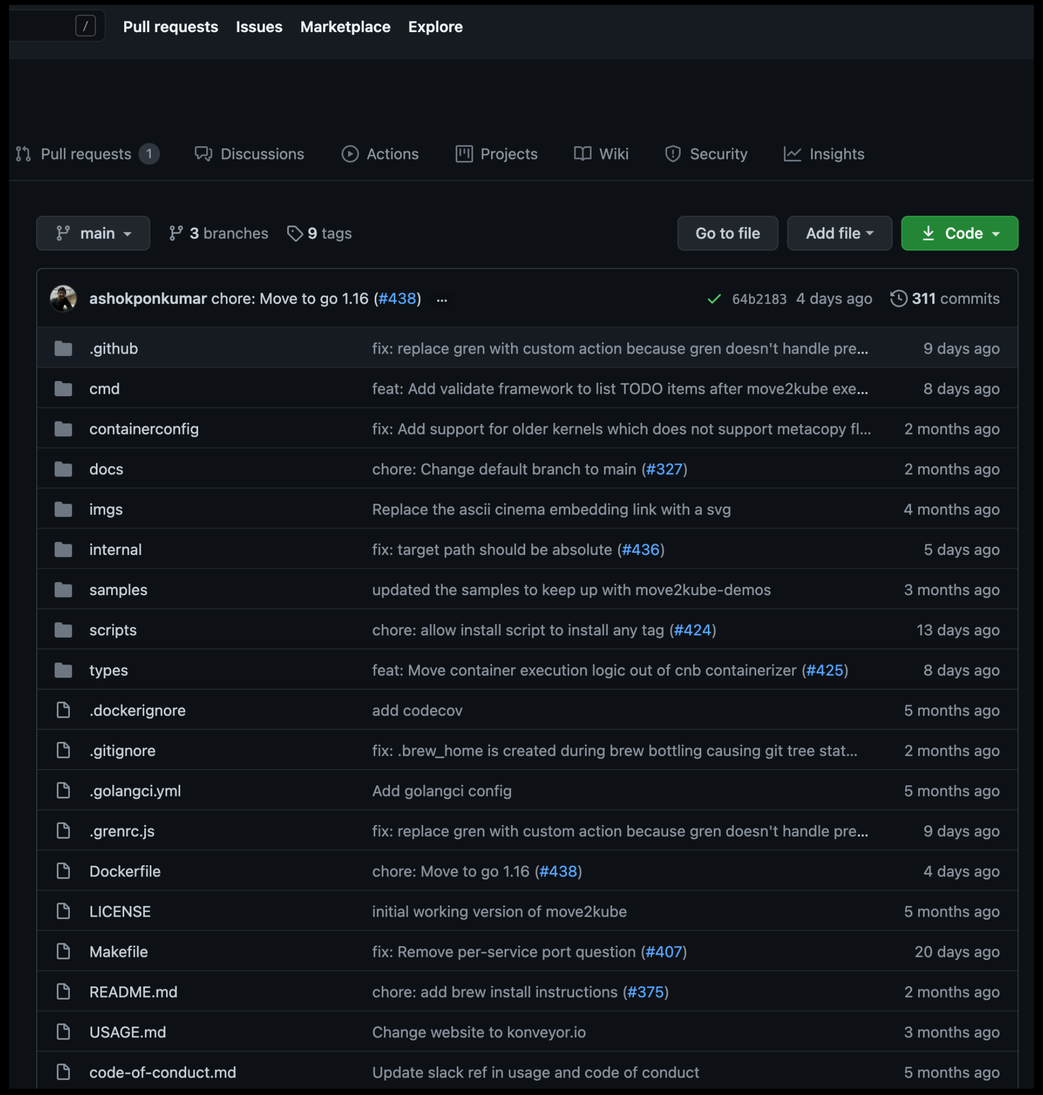
What is GitHub?

- GitHub is the most popular of all the Git hosting services.
- Launched on April 10, 2008.
- Acquired by Microsoft for $7.5 billion in June of 2018.
- Largest user base (over 56 million users). Some of the largest open sources projects (like Linux) are on GitHub.
- The fundamental software that underpins GitHub is Git.
  The GitHub user interface was written using Ruby on Rails and Erlang by GitHub developers Wanstrath, Hyett, and Preston-Werner.



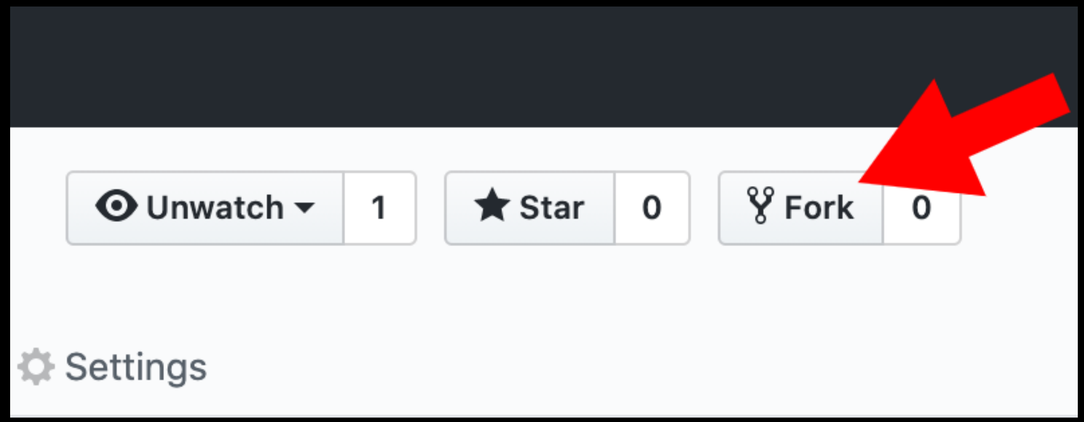
What is GitHub?

- GitHub exposes (almost) all of the functionality of Git through the website. So we can do everything we could do from the CLI, using the UI.
- We can edit a file, create a new file, delete files, etc. and make a new commit with the changes.
- The entire commit history, branches, tags, contributors, etc. are listed on the website.
- GitHub offers both public and private repositories.



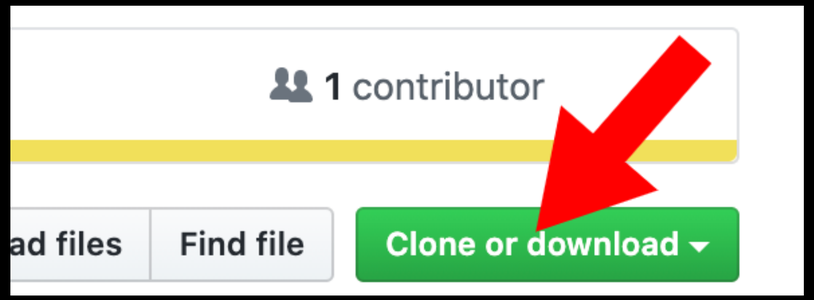
GitHub forking

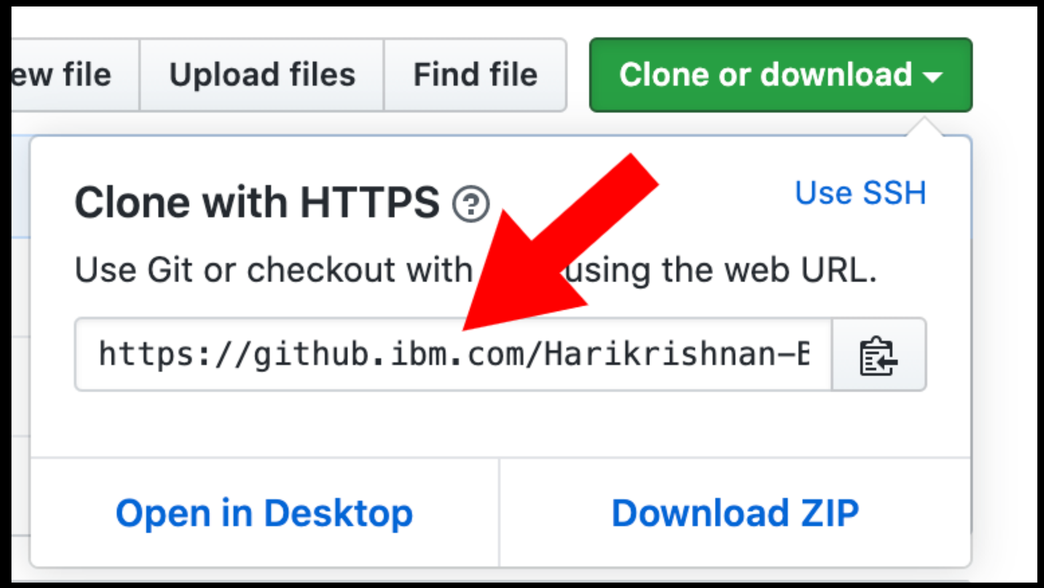
- Fork means to make a copy of a repo.
- To fork, we simply go to the repo we want to make a copy of and click the “fork” button.
- The copy is stored as a new repo in our account.
- Changes on the fork do not affect the original repo we forked.
- This gives us an easy way to mess around and make changes without messing up the history of the original repo.



Cloning a repo on GitHub

- `git clone <repo_url>`
- We can get the repo url by clicking the “Clone or download” button.





GitHub pull requests

- Pull request is a misnomer. Merge request is a better name.
(GitLab calls it merge requests.)

- GitHub allows us to a make request to merge one branch in a repo with another branch. We can also merge branches across forks.

- The request can be made on our own repos and on repos we don’t own/control.

- The maintainers of the repo will review the changes you want to merge and decide what to do. They might request that we make some changes to the PR before it can be merged.

- A pull request (PR) can be used to merge a feature or bug fix that we have implemented on our fork into the original repo.

GitHub PR based workflow

- Setup (we only need to do this one time):
  1. Fork the “upstream” repo. We will call our fork “origin”.

  2. Clone origin to your laptop/desktop and add upstream as a remote.

GitHub PR based workflow

- Workflow:
  1. Create a branch on our local repo.

  2. Make some commits.

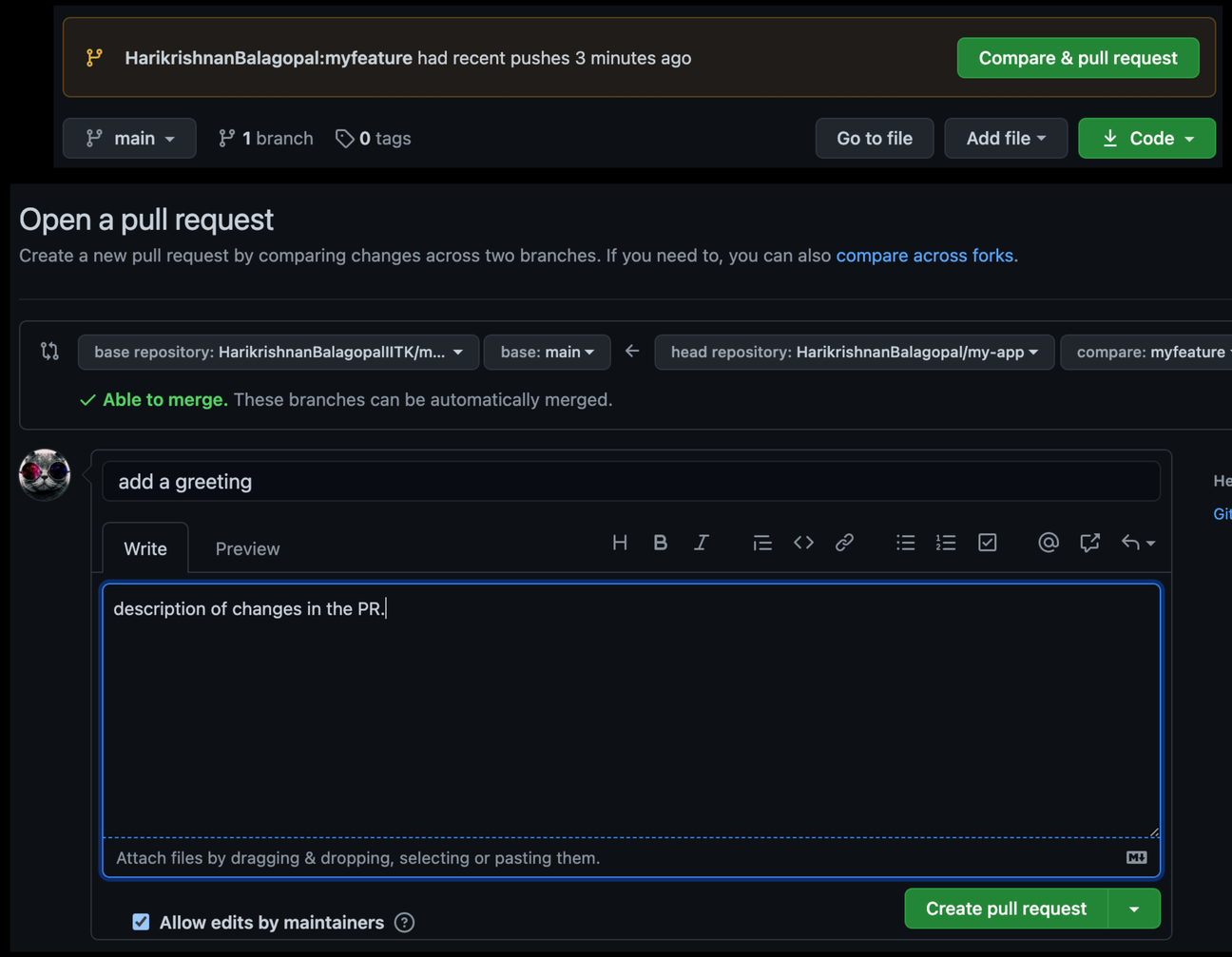
  3. Push the commits to origin.

```
$ x
* commit b811e9d958e7411caaf3a209f9ab0843a7a18535 (HEAD -> main, origin/main, origin/HEAD)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Feb 25 03:23:45 2021 +0530

      add readme and license
$ ls
LICENSE   README.md
$ git checkout -b myfeature
Switched to a new branch 'myfeature'
$ echo 'print("hello there!")' > main.py
$ git add -A && git commit -m 'add a greeting' && git push -u origin myfeature
[myfeature 6992d5c] add a greeting
 1 file changed, 1 insertion(+)
 create mode 100644 main.py
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 999 bytes | 999.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'myfeature' on GitHub by visiting:
remote:      https://github.com/HarikrishnanBalagopal/my-app/pull/new/myfeature
remote:
To github.com:HarikrishnanBalagopal/my-app.git
 * [new branch]      myfeature -> myfeature
Branch 'myfeature' set up to track remote branch 'myfeature' from 'origin'.
$ x
* commit 6992d5cbd6539fd6b41836ca38882563ff9c9cc7 (HEAD -> myfeature, origin/myfeature)
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Thu Feb 25 03:53:24 2021 +0530
|
|     add a greeting
|
* commit b811e9d958e7411caaf3a209f9ab0843a7a18535 (origin/main, origin/HEAD, main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Feb 25 03:23:45 2021 +0530

      add readme and license
$
```

GitHub PR based workflow

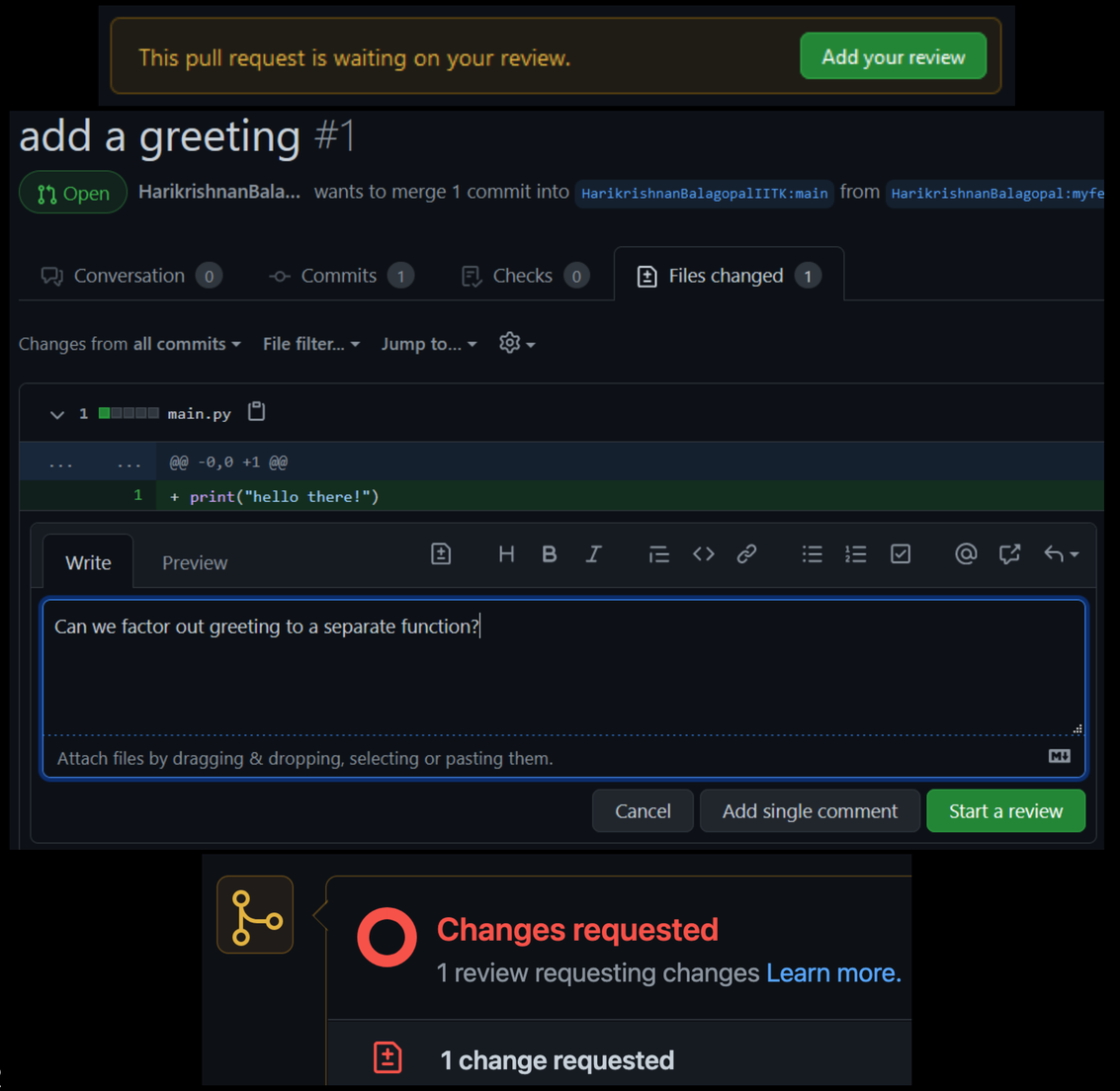
- Workflow:
  4. Go to the upstream repo and make a PR against the master/main branch.

  5. The maintainers will review the PR and request changes.



GitHub PR review

- Maintainers of a repo can review PRs on that repo.
- They can add comments on each line.
- We can reply to these comments individually and this often leads to multiple independent threads of conversation.
- Maintainers can finish the review with 3 different statuses:
  Approve, Comment or Request Changes.
- Don’t panic if they request changes since this is a VERY common thing to do.



GitHub PR based workflow

- Workflow:
  6. Make the necessary changes and push the commits to origin. The PR will update automatically. Now we can request another review.

  7. Repeat step 6 until they accept the PR. The PR then gets merged.

  8. We can now delete the feature branch from our local repo and origin.

```
$ git add -A && git commit -m 'refactor greeting' && git push
[myfeature e9d585b] refactor greeting
 1 file changed, 5 insertions(+), 1 deletion(-)
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 979 bytes | 979.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:HarikrishnanBalagopal/my-app.git
   6992d5c..e9d585b  myfeature -> myfeature
$ x
* commit e9d585b59ea3cd6232300677f394e55b9885d7fc (HEAD -> myfeature, origin/myfeature)
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Thu Feb 25 04:26:11 2021 +0530
|
|     refactor greeting
|
* commit 6992d5cbd6539fd6b41836ca38882563ff9c9cc7
| Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| Date:   Thu Feb 25 03:53:24 2021 +0530
|
|     add a greeting
|
* commit b811e9d958e7411caaf3a209f9ab0843a7a18535 (origin/main, origin/HEAD, main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Feb 25 03:23:45 2021 +0530

      add readme and license
$
```

GitHub PR based workflow

- As an additional step we can sync the main branch on our local repo with upstream to get the changes we just merged.
- Then we push the changes to the main branch of origin.
- Now we are back where we started and can start working on the next PR.
- It is also possible to have multiple PRs going at the same time.

```
$ git fetch --all
Fetching origin
Fetching upstream
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 1 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 765 bytes | 382.00 KiB/s, done.
From github.com:HarikrishnanBalagopalIITK/my-app
 * [new branch]      main       -> upstream/main
$ x
* commit 53bf8c014873581cdedf5a9933878ca1670f3180 (upstream/main)
| Author: HarikrishnanBalagopal <harikrishmenon@gmail.com>
| Date:   Thu Feb 25 04:33:35 2021 +0530
|
|     add a greeting (#1)
|
|     * add a greeting
|
|     * refactor greeting
|
| * commit e9d585b59ea3cd6232300677f394e55b9885d7fc (HEAD -> myfeature, origin/myfeature)
| | Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
| | Date:   Thu Feb 25 04:26:11 2021 +0530
| |
| |     refactor greeting
| |
| * commit 6992d5cbd6539fd6b41836ca38882563ff9c9cc7
|/  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
|   Date:   Thu Feb 25 03:53:24 2021 +0530
|
|       add a greeting
|
* commit b811e9d958e7411caaf3a209f9ab0843a7a18535 (origin/main, origin/HEAD, main)
  Author: Harikrishnan Balagopal <harikrishmenon@gmail.com>
  Date:   Thu Feb 25 03:23:45 2021 +0530

      add readme and license
$
```

Enterprise workflows	Open source workflows
Workflows tend to depend on company policy and team leads.	Workflows depend on project size and scope, number of contributors, history/project origins, code frequency, etc.
May have access to commit directly to the main/master branch.	Usually have no access to commit directly to any branch. Have to fork and clone.
There may be force pushes to the upstream repo. Especially if sensitive info was added accidentally.	Almost never force push since it requires a flag day, and requires a large number of users to sync the changes.
Changes might be made to the repo with little to no public discussion.	All changes need to be discussed and agreed to in public forums before they can be merged.

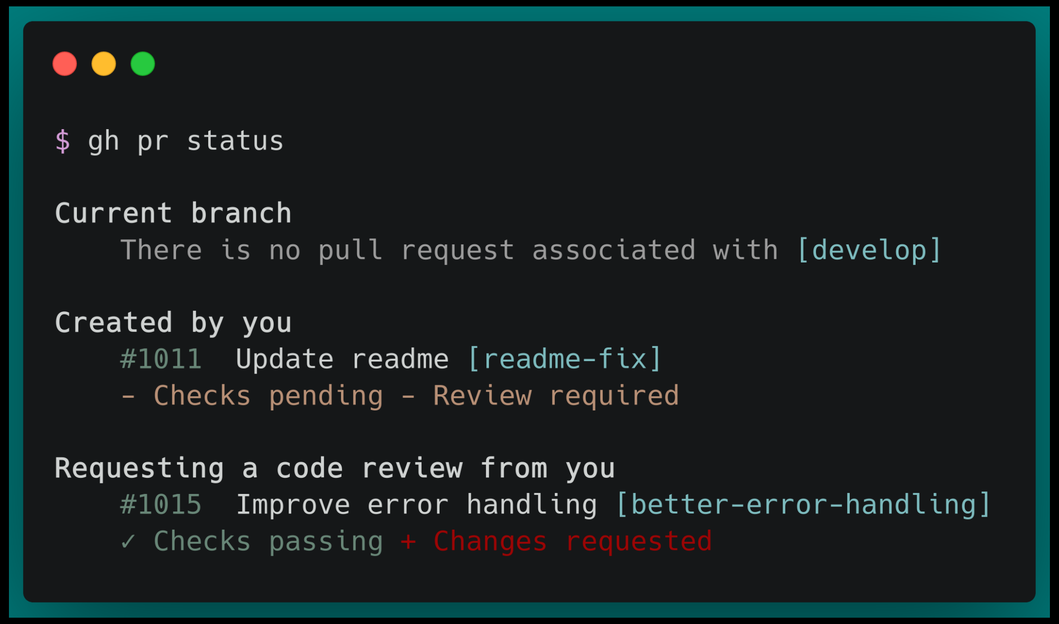
Best practices

- Always fork and clone the repo to start contributing.
- Never commit directly to main/master branch. Always make a new branch.
- Always link your pull request to an existing issue on the Github repo. If there are no appropriate issues then create one.
- Each open source project will have their own set of standards regarding commit messages, issue format, PR titles, PR review process, license agreements, etc.

  Read the `CONTRIBUTING.md` file if it exists. Every large open source project will have one next to the `README.md`, with the details on how to contribute.

GitHub CLI

- `gh` is a tool for interacting with GitHub from the command line: https://github.com/cli/cli
- Available for MacOS, Linux and Windows.
- `gh pr checkout <pr_number>` is an easy way to checkout a PR for review and testing locally.
- The gh tool exposes all of the GitHub specific functionality such as, making comments on issues and PRs, creating new issues and PRs, submitting PR reviews, and a lot more.



Demo

Resources

- Videos:
  - Dives into git internals to give a better understanding.
    Lecture 6: Version Control (git) (2020)

  - Only 7 minutes and straight to the point. Perhaps a bit overwhelming.
    Git Internals - Git Objects

  - Long but goes into much more depth, including a look at the actual files and folders inside.
    Deep Dive into Git - Edward Thomson

- Books:
  - The official git book. https://git-scm.com/book/en/v2

- Interactive guide for fixing mistakes:
  - https://sukima.github.io/GitFixUm/

- Useful info:
  - How to write meaningful commit messages: https://www.conventionalcommits.org/en/v1.0.0/
  - Workflow: https://github.com/konveyor/move2kube/blob/main/git-workflow.md

- Slides:
  https://github.ibm.com/Harikrishnan-Balagopal/git-exercises/blob/master/presentation.pdf

There will a session in April on Best Practices for Open Source Projects

Thank You! 😀